SARAH KAISER, CHRISTOPHER GRANADE

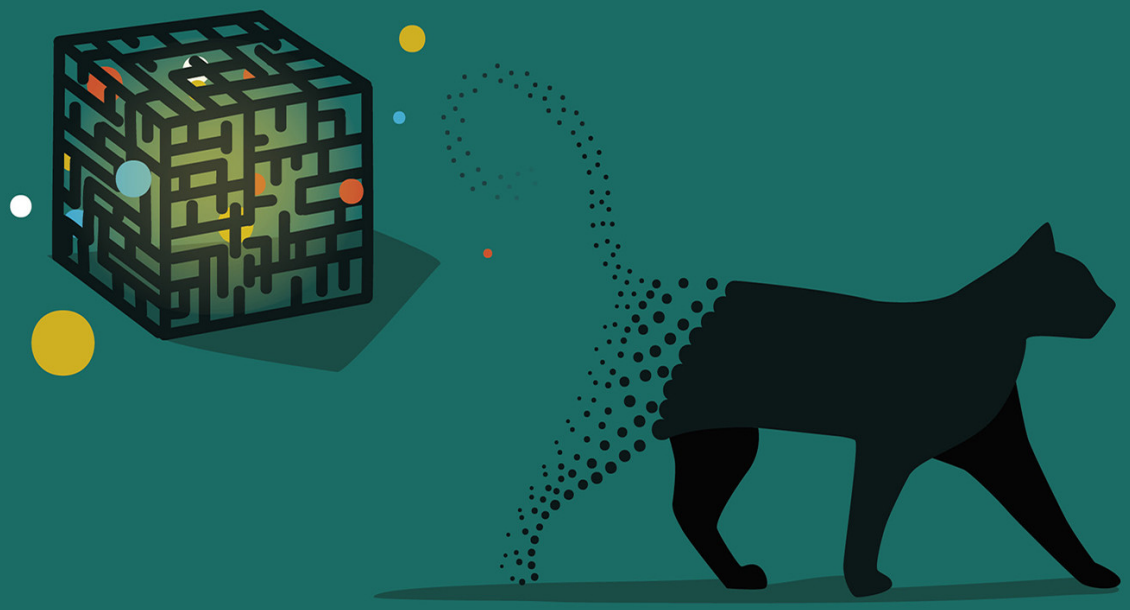
Quantum Computing

Guida alla programmazione con Python e Q#

APOGEO

SARAH KAISER, CHRISTOPHER GRANADE

Quantum Computing



Guida alla programmazione con Python e Q#

APOGEO

Quantum Computing
Guida alla programmazione con Python e Q#

Sarah Kaiser, Christopher Granade

APOGEO

© Apogeo - IF - Idee editoriali Feltrinelli s.r.l.
Socio Unico Giangiacomo Feltrinelli Editore s.r.l.

ISBN ebook: 9788850319183

Original English language edition published by Manning Publications. USA.
Copyright (c) 2020 by Manning Publications. All rights reserved.

Il presente file può essere usato esclusivamente per finalità di carattere personale. Tutti i contenuti sono protetti dalla Legge sul diritto d'autore.

Nomi e marchi citati nel testo sono generalmente depositati o registrati dalle rispettive case produttrici.

L'edizione cartacea è in vendita nelle migliori librerie.

~

Sito web: www.apogeonline.com

Scopri le novità di Apogeo su Facebook

Seguici su Twitter

Collegati con noi su LinkedIn

Guarda cosa stiamo facendo su Instagram

Rimani aggiornato iscrivendoti alla nostra newsletter

~

Sai che ora facciamo anche CORSI? Dai un’occhiata al calendario.

Dedichiamo questo libro alla prossima generazione di sviluppatori quantistici, che stanno lavorando per rendere questo campo uno spazio più sicuro e inclusivo.

Per la maggior parte della sua storia, il quantum computing è stato un campo per i fisici, magari con una certa propensione per la computer science, ma non necessariamente. Il libro *Quantum Computation and Quantum Information*, di Michael A. Nielsen e Isaac L. Chuang, è considerato ancora oggi il testo di riferimento ed è stato scritto da due fisici quantistici. A dire il vero, i programmatori puri sono sempre stati presenti, ma alcuni teorici indossano le poche righe di codice che hanno scritto come una sorta di distintivo. Questo è il mondo quantistico in cui io, Kaiser e Granade siamo cresciuti. Quindi ho tutti i diritti di alzare il pugno alla nuova coorte di studenti e gridare: "Quando io avevo la tua età, non scrivevamo codice, ci strozzavamo alla lavagna con la polvere di gesso!".

Ho conosciuto Chris Granade appena dopo la laurea di entrambi. Allora scrivevamo articoli accademici per riviste di fisica che contenevano righe di codice che erano state rifiutate perché "non riguardavano la Fisica". Ma non ci siamo scoraggiati. E ora, molti anni dopo, questo libro rappresenta per me la vendetta finale! Questo è un libro che insegna tutto ciò che c'è da sapere sul quantum computing, senza però il supporto della fisica – ma se volete conoscere i suoi legami con la fisica, Kaiser e Granade offrono anche questo 😆? Sì, troverete anche le emoji 😉!

Ho fatto molta strada da allora, e sia io sia l'intero campo del quantum computing dobbiamo molto a Granade, per aver mostrato a

molti di noi che tra “calcolo” e “quantistico” c’è ben di più che teoremi e dimostrazioni. Kaiser mi ha anche insegnato più di quanto immaginassi sulla necessità del tocco dell’ingegnere del software nello sviluppo della tecnologia quantistica. Kaiser e Granade hanno trasformato la loro esperienza in parole e righe di codice, e questo a vantaggio di tutti, e anche mio.

Sebbene l’obiettivo fosse quello di creare un “non libro di testo”, questo libro potrebbe certamente essere utilizzato come tale in un corso universitario, come un’introduzione al passaggio del quantum computing dai dipartimenti di Fisica a quelli di Computer science. C’è un interesse crescente per il quantum computing e nella maggior parte dei casi non nel campo della fisica: sviluppatori di software, responsabili di produzione e dirigenti finanziari vogliono tutti sapere di che cosa si tratta e come metterci sopra le mani. Sono finiti i giorni in cui il quantum computing era un’attività puramente accademica. Questo libro soddisfa le esigenze della crescente comunità quantistica.

Sebbene abbia accennato alla proporzione decrescente di fisici nel campo del quantum computing, non per questo voglio sottovalutarli. Questo libro è davvero per chiunque, specialmente per coloro che, già esperti nel campo, vogliono conoscere il lato software del quantum computing in un ambiente familiare.

Avviate il vostro editor di codice preferito e preparatevi al classico “Hello quantum world!”.

Chris Ferrie
Professore associato, Center for Quantum Software and Information
Sydney, Nuovo Galles del Sud, Australia

Il quantum computing è stato l'argomento su cui abbiamo lavorato insieme per più di 20 anni, e siamo ansiosi di prendere quell'esperienza e di usarla per aiutare sempre più persone a lasciarsi coinvolgere dalle tecnologie quantistiche. Abbiamo completato il nostro dottorato insieme e, mentre lo facevamo, abbiamo lottato con bandi di ricerca, gare di giochi di parole e giochi da tavolo, aiutando a spingere i confini di ciò che era possibile fare con i qubit. Per la maggior parte, ciò significava sviluppare nuovi software e strumenti per aiutare noi e i nostri team a eseguire ricerche migliori, il che è stato un ottimo ponte tra le parti "computer science" e "quantistica". Tuttavia, durante lo sviluppo di vari progetti software, dovevamo spiegare ai nostri colleghi sviluppatori su che cosa stavamo lavorando e così continuavamo a chiederci: "Perché non esiste un buon libro dedicato al quantum computing che sia, sì, tecnico, ma non un libro di testo?". Il risultato sono esattamente le pagine che avete in mano. 💔

Abbiamo scritto il libro perché fosse accessibile agli sviluppatori, piuttosto che scriverlo nello stile dei libri di testo, così tipico in altri libri di quantum computing. Imparare da soli il quantum computing è stato molto eccitante ma anche un po' spaventoso e pesante. Questo argomento però non deve essere visto così, perché ciò che rende davvero complessi gli argomenti quantistici è principalmente il *modo* in cui sono presentati, non il loro contenuto.

Sfortunatamente, il quantum computing è spesso descritto come "strano", "inquietante" o al di là della nostra comprensione, quando la verità è che è ormai abbastanza comprensibile, dopo 35 anni di storia. Impiegando una combinazione di sviluppo software e matematica, *potete* acquisire i concetti di base necessari per comprendere il senso del quantum computing ed esplorare questo fantastico nuovo campo.

Il nostro obiettivo con questo libro è di aiutarvi ad apprendere le basi della tecnologia e fornirvi strumenti pratici per costruire le soluzioni quantistiche di domani. Ci concentriamo soprattutto sull'esperienza pratica nello sviluppo del codice per il quantum computing. Nella Parte 1, potrete costruire un simulatore di dispositivi quantistici in Python; nella Parte 2, imparerete ad applicare queste nuove abilità alla scrittura di applicazioni quantistiche con Q# e il Quantum Development Kit; nella Parte 3, imparerete a implementare un algoritmo che calcola gli interi in modo esponenziale più velocemente del più noto algoritmo convenzionale. E sarete *voi* a farlo, e questo è il *vostro* viaggio quantistico.

Abbiamo incluso quante più applicazioni pratiche possibile, ma la verità è che è qui che entrate in gioco voi! Il quantum computing è giunto a un punto in cui per andare avanti abbiamo bisogno di gettare un ponte tra l'immensa quantità di ciò che è noto su quello che i computer quantistici possono e non possono fare e i problemi che le persone devono risolvere. La costruzione di quel ponte ci porta dagli algoritmi quantistici di ricerca, teorici, a nuovi algoritmi quantistici che possono avere un impatto su tutta la società. Potete certamente aiutare a costruire quel ponte. Benvenuti quindi nel vostro viaggio quantistico; siamo qui per aiutarvi a renderlo anche divertente!

All'inizio non sapevamo a che cosa saremmo andati incontro, con questo libro; tutto quello che sapevamo era che una risorsa come questa doveva esistere. Scrivere questo libro ci ha dato un'enorme opportunità per affinare e sviluppare le nostre capacità di spiegare e insegnare i contenuti che conoscevamo. Tutti coloro con cui abbiamo lavorato in Manning sono stati meravigliosi: Deirdre Hiam, la nostra editor di produzione, Tiffany Taylor, la nostra copy-editor, Katie Tennant, la nostra correttrice di bozze, Ivan Martinović, il nostro revisore, ci hanno aiutato a fare in modo che questo fosse il miglior libro possibile, per i nostri lettori.

Ringraziamo Olivia Di Matteo e Chris Ferrie per tutti i loro preziosi feedback e note, che hanno contribuito a mantenere le nostre spiegazioni sempre accurate e chiare.

Ringraziamo anche tutti i revisori del manoscritto, che lo hanno esaminato nelle varie fasi di sviluppo e il cui attento feedback ha reso questo libro migliore: Alain Couniot, Clive Harber, David Raymond, Debmalya Jash, Dimitri Denisjonok, Domingo Salazar, Emmanuel Medina Lopez, Geoff Clark, Javier, Karthikeyarajan Rajendran, Krzysztof Kamyczek, Kumar Unnikrishnan, Pasquale Zirpoli, Patrick Regan, Paul Otto, Raffaella Ventaglio, Ronald Tischliar, Sander Zegveld, Steve Sussman, Tom Heiman, Tuan A. Tran, Walter Alexander Mata Lopez e William E. Wheeler.

I nostri ringraziamenti vanno a tutti gli abbonati al Manning Early Access Program (MEAP), che ci hanno aiutato a trovare bug, errori di battitura e i punti in cui era meglio migliorare le spiegazioni. Molte persone hanno anche fornito i loro feedback indicando i problemi sul nostro archivio di codice di esempio: i nostri più sentiti ringraziamenti!

Vorremmo ringraziare i molti ottimi locali nell'area di Seattle (in particolare Caffe Ladro, Miir, Milstead & Co. e Downpour Coffee Bar) che ci hanno tollerato mentre trangugiavamo caffè dopo caffè e parlavamo animatamente di qubit, così come il meraviglioso personale di Fremont Brewing, sempre presente quando avevamo bisogno di una pinta. È sempre stata una gradita interruzione quando un passante casuale ci poneva domande su ciò su cui stavamo lavorando!

Vorremmo anche ringraziare i membri di talento del team Quantum Systems di Microsoft, per aver lavorato per fornire agli sviluppatori i migliori strumenti possibili per l'accesso al quantum computing. In particolare, i nostri ringraziamenti vanno a Bettina Heim, per essere riuscita a rendere Q# un linguaggio straordinario, oltre che per essere una nostra buona amica.

Infine, ringraziamo il nostro pastore tedesco, Chewie, che ha fornito distrazioni davvero necessarie, e per aver rappresentato una scusa per fare una pausa lontano dal computer.

Sarah Kaiser

La mia famiglia è sempre stata lì per me e ringrazio tutti per la loro pazienza e il loro incoraggiamento mentre lavoravo a questo progetto. Vorrei ringraziare il mio terapista, senza il quale questo libro non sarebbe mai esistito. Soprattutto, voglio ringraziare il mio coautore e partner, Chris. Sono stati lì, con me, nel bene e nel male, e mi hanno sempre incoraggiato e ispirato a fare ciò che sapevano che potevo fare.

Christopher Granade

Questo libro non sarebbe stato possibile senza lo straordinario amore e il sostegno della mia partner e coautrice, Sarah Kaiser. Insieme, ne abbiamo passate di tutti i colori e abbiamo ottenuto ben più di quanto avrei mai immaginato. La nostra storia ha sempre riguardato la creazione di una comunità quantistica migliore, più sicura e più inclusiva, e questo libro è un'opportunità meravigliosa per compiere un altro passo in questo viaggio. Grazie per aver reso possibile tutto questo, Sarah.

Né questo sarebbe stato possibile senza il sostegno della mia famiglia e dei miei amici. Grazie per essere sempre lì, a condividere foto di cuccioli, a discutere delle ultime notizie o a partecipare all'osservazione dei meteoriti a tarda notte in Animal Crossing. Infine, ringrazio anche la fantastica comunità online su cui ho fatto affidamento negli anni per aiutarmi a capire il mondo da sempre nuove prospettive.

Sarah Kaiser ha completato il dottorato in Fisica (Informazione quantistica) presso l'Institute for Quantum Computing dell'Università di Waterloo. Ha trascorso gran parte della sua carriera sviluppando nuovo hardware quantistico in laboratorio, dalla costruzione di satelliti all'hacking dell'hardware di crittografia quantistica. La sua passione è comunicare ciò che c'è di eccitante nel quantum computing e adora costruire nuove demo e strumenti per favorire la crescita della comunità quantistica. Quando non è alla sua tastiera, ama andare in kayak e scrivere libri scientifici per tutte le età.

Christopher Granade ha completato il dottorato in Fisica (Informazione quantistica) presso l'Institute for Quantum Computing dell'Università di Waterloo e ora lavora nel gruppo Quantum Systems di Microsoft, che si occupa dello sviluppo di librerie standard per Q# e della caratterizzazione statistica di dispositivi quantistici da dati classici. Chris ha anche aiutato Scott Aaronson a raccogliere le sue lezioni in un libro, *Quantum Computing Since Democritus* (Cambridge University Press, 2013).

Benvenuti in *Quantum Computing. Guida alla programmazione con Python e Q#*. Questo libro vi introdurrà al mondo del quantum computing utilizzando Python come comodo punto di partenza e sviluppando soluzioni scritte in Q#, un linguaggio di programmazione specifico sviluppato da Microsoft. Adotteremo un approccio basato su esempi e sul gioco per introdurre il quantum computing e i concetti di sviluppo che vi consentano di iniziare subito a scrivere codice.

Approfondimento: va bene fare snorkeling!
Il quantum computing è un'area di studio sommamente interdisciplinare, che riunisce concetti di programmazione, fisica, matematica, ingegneria e computer science. Di tanto in tanto, nel corso del libro, dedicheremo un momento a sottolineare come il quantum computing attinga idee da questi altri campi per introdurre in un contesto più ricco i concetti che stiamo trattando.
Mentre questi "inserti" hanno lo scopo di suscitare curiosità e suggerire ulteriori esplorazioni, hanno una natura "tangenziale". Nel libro potrete trovare tutto il necessario per apprendere la programmazione quantistica in Python e Q#, indipendentemente dal fatto che vi immergiate in queste profonde digressioni. Fare un'immersione profonda può essere divertente e illuminante, ma se le immersioni profonde non fanno per voi, va bene; va benissimo anche lo "snorkeling".

A chi è rivolto questo libro

Questo libro è rivolto a persone interessate al quantum computing ma che hanno poca o addirittura nessuna esperienza con la meccanica

quantistica, ma con un background di programmazione. Mentre imparerete a scrivere simulatori quantistici in Python e programmi quantistici in Q#, il linguaggio Microsoft dedicato al quantum computing, utilizzeremo idee e tecniche di programmazione tradizionali, per semplificare le cose. Sarà quindi utile avere una comprensione generale dei concetti base della programmazione come i cicli, le funzioni e gli assegnamenti di variabili.

Allo stesso modo, trarremo alcuni concetti matematici dall'algebra lineare, come i vettori e le matrici, per aiutarci a descrivere i concetti quantistici; se avete familiarità con la computer grafica o il machine learning, troverete che molti dei concetti sono simili. Useremo Python per rivedere i concetti matematici più importanti, ma una certa familiarità con l'algebra lineare sarà certamente utile.

Come è organizzato questo libro: una tabella di marcia

Questo testo mira a consentirvi di iniziare a esplorare e utilizzare alcuni strumenti pratici per il quantum computing. Il libro è suddiviso in tre parti, da leggere in successione.

- La Parte I introduce senza troppi traumi i concetti necessari per descrivere i qubit, l'unità fondamentale di un computer quantistico. Questa parte descrive come simulare i qubit in Python, semplificando la scrittura di semplici programmi quantistici.
- La Parte II spiega come utilizzare il Quantum Development Kit e il linguaggio di programmazione Q# per comporre qubit ed eseguire algoritmi quantistici che funzionano in modo diverso da ogni algoritmo classico noto.

- Nella Parte III applichiamo gli strumenti e i metodi delle due parti precedenti per scoprire come i computer quantistici possono essere applicati ai problemi del mondo reale, come la simulazione delle proprietà chimiche.

Sono previste anche quattro appendici. L'Appendice A contiene tutte le istruzioni di installazione per configurare gli strumenti che utilizziamo nel libro. L'Appendice B è una sezione di riferimento rapido con un glossario quantistico, un promemoria di notazioni e alcuni frammenti di codice che possono essere utili man mano che procederete nel libro. L'Appendice C è un ripasso dell'algebra lineare e l'Appendice D è un'immersione profonda in uno degli algoritmi che verranno implementati.

Il codice

Tutto il codice utilizzato in questo libro può essere trovato su `https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp`. Le istruzioni complete per l'installazione sono disponibili nell'archivio di questo libro e nell'Appendice A.

Gli esempi del libro possono anche essere eseguiti online senza installare nulla, utilizzando il servizio `mybinder.org`. Per iniziare, andate su `https://bit.ly/qsharp-book-binder`.

Altre risorse online

Mentre iniziate il vostro viaggio nel quantum computing con questo libro e lavorando sul codice di esempio, potreste trovare utili le seguenti risorse online.

- La documentazione del *Quantum Development Kit* (`https://docs.microsoft.com/azure/quantum`): documentazione

concettuale e guida di riferimento completa a tutto ciò che riguarda Q#, comprese le modifiche e le nuove aggiunte successive alla stampa di questo libro.

- Esempi del *Quantum Development Kit* (https://github.com/microsoft/quantum): esempi completi per l'utilizzo di Q#, sia da solo sia con programmi host in Python e .NET, coprendo un'ampia gamma di applicazioni.
- *QuTiP.org* (http://qutip.org): guida completa per l'utente per il package QuTiP che abbiamo usato per la matematica presente in questo libro.

Ci sono anche alcune ottime comunità che raggruppano esperti di quantum computing e principianti. Entrare a far parte di una comunità di sviluppo quantistico come le seguenti può aiutarvi a risolvere le domande che vi sorgeranno lungo il percorso e vi consentirà anche di assistere gli altri nel loro viaggio.

- *qsharp.community* (https://qsharp.community): una comunità di utenti e sviluppatori Q#, dotata di chatroom, blog e repository di progetti.
- *Quantum Computing Stack Exchange* (https://quantumcomputing.stackexchange.com): un ottimo luogo per porre domande sul quantum computing, comprese eventuali domande su Q#.
- *Women in Quantum Computing and Applications* (https://wiqca.dev): una comunità inclusiva per persone di tutti i sessi per celebrare il quantum computing e coloro che lo rendono possibile.
- *Quantum Open Source Foundation* (https://qosf.org): una comunità che supporta lo sviluppo e la standardizzazione di strumenti aperti per il quantum computing

- *Unitary Fund* (https://unitary.fund): un'organizzazione non profit che lavora per creare un ecosistema tecnologico quantistico a vantaggio della maggior parte delle persone.

Per approfondire

Il quantum computing è un campo nuovo e affascinante, che offre nuovi modi per pensare al calcolo e nuovi strumenti per risolvere problemi difficili. Questo libro può aiutarvi a iniziare a conoscere il quantum computing, in modo da consentirvi poi di continuare a esplorare e imparare. Ciò detto, questo non è un libro di testo e non ha lo scopo di prepararvi da soli all'argomento del quantum computing. Come per gli algoritmi classici, anche lo sviluppo di nuovi algoritmi quantistici è un'arte matematica, tanto quanto qualsiasi altra cosa; dal momento che impieghiamo la matematica per spiegare gli algoritmi, troverete vari di libri di testo che possono aiutarvi a formarvi le idee di base.

Dopo aver letto questo libro e aver iniziato a impiegare il quantum computing, se volete continuare il viaggio nella fisica o nella matematica, vi suggeriamo una delle seguenti risorse.

- *The Complexity Zoo* (https://complexityzoo.net/Complexity_Zoo).
- *The Quantum Algorithm Zoo* (http://quantumalgorithmzoo.org).
- *Complexity Theory: A Modern Approach*, di Sanjeev Arora e Boaz Barak (Cambridge University Press, 2009).
- *Quantum Computing: A Gentle Introduction*, di Eleanor G. Rieffel e Wolfgang *H*. Polak (MIT Press, 2011).
- *Quantum Computing since Democritus*, di Scott Aaronson (Cambridge University Press, 2013).
- *Quantum Computation and Quantum Information*, di Michael A. Nielsen e Isaac L. Chuang (Cambridge University Press, 2000).

- *Quantum Processes Systems, and Information*, di Benjamin Schumacher and Michael Westmoreland (Cambridge University Press, 2010)

Primi passi con il quantum computing

Questa parte del libro aiuta a preparare il terreno per il resto del nostro viaggio quantistico. Il Capitolo 1 presenta il contesto del quantum computing, l'approccio all'apprendimento del quantum computing scelto in questo libro e dove potete aspettarvi di utilizzare le abilità che apprenderete. Nel Capitolo 2, inizieremo a scrivere del codice sviluppando un simulatore quantistico in Python. Quindi utilizzeremo il simulatore per programmare un generatore quantistico di numeri casuali. Successivamente, nel Capitolo 3, estenderemo il simulatore per programmare applicazioni crittografiche della tecnologia quantistica, come il protocollo di scambio quantistica delle chiavi BB84. Nel Capitolo 4, utilizzeremo giochi non locali per parlare di entanglement ed estendere ancora una volta il simulatore per supportare più qubit. Nel Capitolo 5, impareremo a utilizzare un nuovo package Python per implementare strategie quantistiche per giocare ai giochi non locali del Capitolo 4. Infine, nel Capitolo 6, estenderemo il simulatore un'ultima volta, aggiungendo nuove operazioni quantistiche per poter simulare tecniche come il *teletrasporto quantistico* e la pratica del trasferimento dei dati nei nostri dispositivi quantistici.

Introduzione al quantum computing

Il quantum computing è stato un campo di ricerca sempre più vigoroso e fonte di clamore negli ultimi anni. Utilizzando la fisica quantistica per eseguire calcoli in nuovi e straordinari modi, i computer quantistici possono avere un impatto sulla società, e pertanto questo è un momento entusiasmante per scoprire e imparare a programmare i computer quantistici e applicare risorse quantistiche alla soluzione di problemi importanti.

Con tutto il rumore sui vantaggi offerti dal quantum computing, tuttavia, è facile perdere di vista la reale portata di tali vantaggi. Abbiamo alcuni interessanti precedenti storici di ciò che può accadere quando le promesse di una tecnologia superano la realtà. Negli anni Settanta, il machine learning e l'intelligenza artificiale hanno sofferto di finanziamenti drasticamente ridotti, poiché l'attenzione dei media e l'entusiasmo per l'intelligenza artificiale aveva superato i risultati; questo periodo sarebbe stato poi chiamato "l'inverno dell'AI". Allo stesso modo, le società basate su Internet hanno affrontato lo stesso pericolo quando hanno cercato di superare il fallimento delle dot-com.

Un modo di procedere consiste nel comprendere in modo critico le promesse del quantum computing, come funzionano i computer quantistici e anche, semplicemente, che cos'è e non è il quantum computing. In questo capitolo, vi aiutiamo a sviluppare tali

conoscenze, in modo che possiate mettervi a scrivere i vostri programmi quantistici, con l'aiuto del libro.

Ma, a parte tutto questo, la verità è che è davvero fantastico conoscere un modello di elaborazione interamente nuovo! Mentre leggerete questo libro, scoprirete come funzionano i computer quantistici, programmando simulazioni che potrete eseguire anche su un computer portatile. Queste simulazioni vi mostreranno molti elementi essenziali del modo in cui ci aspettiamo che sia la vera programmazione quantistica, quando sarà disponibile il relativo hardware commerciale. Questo libro è rivolto a chi abbia qualche esperienza di programmazione di base e algebra lineare, ma nessuna conoscenza preliminare di fisica quantistica o computer science. Se avete già una certa familiarità con i concetti quantistici, potete saltare alle Parti 2 e 3, dove entreremo più in dettaglio nelle attività di programmazione e negli algoritmi quantistici.

Perché il quantum computing è importante?

La tecnologia informatica sta avanzando a un ritmo davvero sbalorditivo. Solo tre decenni fa, il processore 80486 consentiva agli utenti di eseguire 50 MIPS (milioni di istruzioni al secondo). Oggi, anche piccoli computer come il Raspberry Pi possono raggiungere i 5.000 MIPS, mentre i sistemi desktop possono facilmente raggiungere i 50.000-300.000 MIPS. Quando abbiamo un problema di eccezionale difficoltà computazionale, una strategia molto ragionevole per risolverlo potrebbe essere, semplicemente, aspettare la prossima generazione di processori, per semplificarsi la vita, scaricare video in streaming più velocemente e avere giochi più vivaci.

Per molti problemi a cui teniamo, tuttavia, non siamo così fortunati. Potremmo sperare che avendo una CPU due volte più veloce potremo risolvere problemi due volte più complessi, ma come accade sempre nella vita, "di più è anche diverso". Supponiamo di dover ordinare un elenco di 10 milioni di numeri e di scoprire che l'operazione richiede circa 1 secondo. Successivamente, se vorremo ordinare un elenco di 1 miliardo di numeri in 1 secondo, avremo bisogno di una CPU 130 volte più veloce, non solo 100 volte. In alcuni tipi di problemi, la situazione è anche peggiore: per alcuni problemi grafici, passare da 10 milioni a 1 miliardo di punti richiederebbe 13.000 volte più tempo.

Problemi del tutto differenti, come il controllo del traffico in una città e la previsione delle reazioni chimiche, diventano sempre più complessi, *molto* rapidamente, aumentando la scala. Se il quantum computing consistesse nel creare un computer 1.000 volte più veloce, potremmo a malapena intaccare le grandi sfide che intendiamo risolvere. Fortunatamente, i computer quantistici sono *molto* più interessanti. Ci aspettiamo che i computer quantistici saranno molto più *lenti* dei computer classici, ma che le risorse necessarie per risolvere molti problemi *varieranno di scala* in modo diverso, in modo tale che, guardando ai giusti tipi di problemi, possiamo andare oltre al "di più è diverso". Allo stesso tempo, i computer quantistici non sono una bacchetta magica: alcuni problemi rimarranno difficili. Per esempio, sebbene sia probabile che i computer quantistici possano aiutarci immensamente nella previsione delle reazioni chimiche, potrebbero non essere di grande aiuto con altri problemi difficili.

Scoprire esattamente in quali problemi possiamo ottenere un reale vantaggio e sviluppare algoritmi quantistici per risolverli è stato un grande obiettivo della ricerca sul quantum computing. Finora, è stato molto difficile valutare gli approcci quantistici in questo modo, poiché ciò richiede una grande abilità matematica, necessaria per scrivere gli

algoritmi quantistici e per comprendere tutte le sottigliezze della meccanica quantistica.

Tuttavia, dal momento in cui l'industria ha iniziato a sviluppare piattaforme in grado di aiutare a connettere gli sviluppatori al quantum computing, questa situazione ha iniziato a cambiare. Utilizzando il Quantum Development Kit Microsoft, possiamo astrarre la maggior parte delle complessità matematiche del quantum computing e iniziare a *comprendere* e *utilizzare* effettivamente i computer quantistici. Gli strumenti e le tecniche insegnati in questo libro consentono agli sviluppatori di esplorare e comprendere come sarà lo "scrivere programmi" per questa nuova piattaforma hardware.

In altre parole, il quantum computing non scomparirà, quindi è davvero molto importante capire a quali problemi possiamo applicarlo! Indipendentemente dal fatto che si verifichi una "rivoluzione" quantistica, il quantum computing ha preso pesantemente in considerazione – e continuerà a farlo – le decisioni su come sviluppare le risorse di calcolo nei prossimi decenni. Decisioni come le seguenti sono fortemente influenzate dal quantum computing:

- Che cosa si intende per "ipotesi ragionevoli" nella sicurezza delle informazioni?
- Quali competenze sono utili nei corsi di laurea?
- Come possiamo valutare il mercato delle soluzioni informatiche?

Coloro, fra noi, che lavorano in settori tecnologici o a essi correlati, si trovano sempre più spesso a prendere decisioni di questo tipo o a fornire input per tali decisioni. Abbiamo la responsabilità di capire che cos'è il quantum computing e, cosa forse più importante, che cosa non è. In questo modo saremo preparati al meglio per intensificare gli sforzi e contribuire a queste nuove decisioni.

A parte questo, un altro motivo per cui il quantum computing è un argomento così affascinante è che è allo stesso tempo così simile e così

diverso dalla computer science classica. Comprendere sia le somiglianze sia le differenze tra la computer science classica e quella quantistica ci aiuta a capire che cosa è fondamentale in tutta la computer science, in generale. Il calcolo, sia classico sia quantistico, deriva da diverse descrizioni delle leggi fisiche, così che la comprensione del calcolo potrà aiutarci a comprendere l'universo in un modo nuovo.

Ciò che è assolutamente fondamentale, tuttavia, è che non esiste una ragione giusta o addirittura migliore per essere interessati al quantum computing. Qualunque cosa vi abbia avvicinato alla ricerca o alle applicazioni del quantum computing, imparete qualcosa di interessante, lungo il percorso.

Che cos'è un computer quantistico?

Cerchiamo di scoprire che cosa rende "quantistico" un computer. Per introdurre questa discussione, parliamo brevemente del significato del termine *computer*.

DEFINIZIONE

Un *computer* è un dispositivo che prende dei dati in input ed esegue qualche tipo di operazione su quei dati.

Ci sono molti esempi di quello che abbiamo appena chiamato *computer*, come potete vedere nella Figura 1.1.

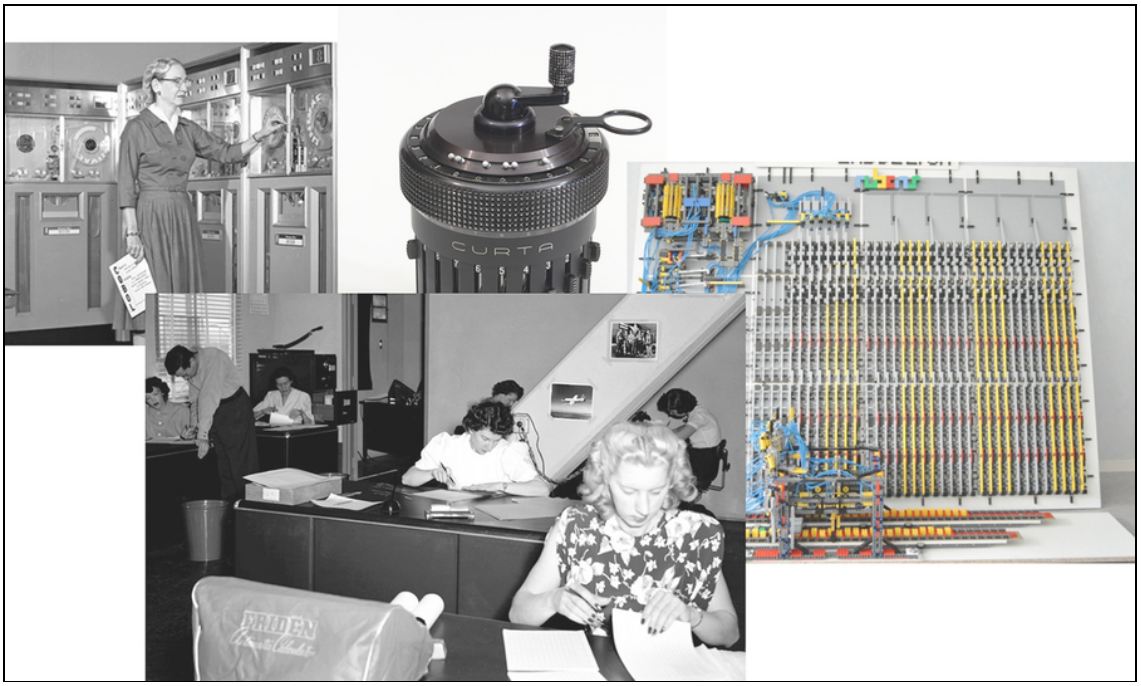


Figura 1.1 Diversi esempi di diversi tipi di computer, incluso il mainframe UNIVAC del contrammiraglio Hopper, una stanza di "computer umani" che lavorano per risolvere i dati di volo, un calcolatore meccanico e una macchina di Turing basata su LEGO. Ogni computer può essere descritto dallo stesso modello matematico: telefoni cellulari, laptop e server. Fonti: foto di "computer umani" della NASA. Foto della macchina di Turing LEGO di Projet Rubens, utilizzata sotto CC BY 3.0 (https://creativecommons.org/licenses/by/3.0).

Tutti questi elementi hanno in comune il fatto che possiamo modellarli con la fisica classica, cioè in termini di leggi del moto di Newton, gravità newtoniana ed elettromagnetismo.

Questo ci aiuterà a distinguere i tipi di computer cui siamo abituati (per esempio laptop, telefoni, tostapane, case, automobili e pacemaker) dai computer che stiamo trattando in questo libro. Per distinguerli, chiameremo i computer che possono essere descritti impiegando la fisica classica con il nome *computer classici*. Un aspetto interessante di questa scelta è che se sostituiamo il termine *fisica classica* con *fisica*

quantistica, abbiamo un'ottima definizione di che cosa sia un computer quantistico.

DEFINIZIONE

Un *computer quantistico* è un dispositivo che prende dei dati in input ed esegue qualche tipo di operazione su quei dati con un processo che può essere descritto solo impiegando la fisica quantistica.

In altre parole, la distinzione tra computer classici e computer quantistici è esattamente quella tra fisica classica e fisica quantistica. Ne parleremo più avanti nel libro, ma la differenza principale è di scala: la nostra esperienza quotidiana riguarda in gran parte oggetti aventi dimensioni e temperature tali per cui gli effetti quantistici, pur esistenti, in generale non agiscono molto. Anche se la meccanica quantistica funziona anche alla scala degli oggetti di uso quotidiano, come le tazzine del caffè, i sacchi di farina e le mazze da baseball, possiamo fare un ottimo lavoro nel descrivere le interazioni fra questi oggetti impiegando solo la fisica classica.

Approfondimento: che cosa è successo alla relatività?

La fisica quantistica si applica a oggetti molto piccoli e molto freddi o ben isolati. Allo stesso modo, un altro ramo della fisica, la *relatività*, descrive oggetti che sono abbastanza grandi da consentire alla gravità di svolgere un ruolo importante o che si muovono molto velocemente, quasi alla velocità della luce. Molti computer si basano su effetti relativistici; infatti, i satelliti di posizionamento globale dipendono in modo critico dalla relatività. Finora abbiamo confrontato principalmente la fisica classica e quella quantistica, quindi… perché parlare di relatività?

Come vedremo, tutto il calcolo implementato utilizzando effetti relativistici può essere descritto utilizzando modelli di calcolo puramente classici, come le macchine di Turing. Al contrario, il quantum computing non può essere descritto come un calcolo classico più veloce, ma richiede un diverso modello matematico. Non è ancora stato proposto per un "computer gravitazionale" che usi la relatività allo stesso modo, quindi possiamo tranquillamente mettere da parte la relatività, in questo libro.

Se ci concentriamo su una scala molto più piccola, in cui è necessaria la meccanica quantistica per descrivere i nostri sistemi, il quantum computing è l'arte di utilizzare dispositivi piccoli e ben isolati per trasformare i dati in modi che non possono essere descritti in termini di pura fisica classica. Un modo per costruire dispositivi quantistici consiste nell'usare piccoli computer classici come i processori di segnali digitali (DSP) per controllare le proprietà di materiali esotici.

Fisica e quantum computing

I materiali esotici utilizzati per costruire computer quantistici hanno nomi che possono anche spaventare, come *superconduttori* e *isolanti topologici*. Possiamo però alleggerire questa preoccupazione pensando a come impariamo a comprendere e utilizzare i computer classici.

Possiamo programmare i computer classici senza sapere che cosa sia un semiconduttore. Allo stesso modo, la fisica in base alla quale costruiamo i computer quantistici è un argomento affascinante, ma è non è necessario conoscerla per imparare a programmare e impiegare i dispositivi quantistici.

I dispositivi quantistici possono differire da quelli classici per i dettagli del modo in cui vengono controllati, ma alla fine tutti i dispositivi quantistici sono controllati e gestiti da computer classici e da un'elettronica di controllo. Dopotutto, siamo interessati a dati classici, quindi, alla fine, deve esistere un'interfaccia fra mondo classico e mondo quantistico.

NOTA

Molti dispositivi quantistici devono essere mantenuti molto freddi e ben isolati, poiché possono essere estremamente sensibili al rumore.

Se applichiamo operazioni quantistiche su un hardware classico embedded, possiamo manipolare e trasformare dati quantistici. La potenza del quantum computing deriva quindi da una scelta attenta di quali operazioni applicare, al fine di attuare una trasformazione utile che risolva il problema in questione.

Come utilizzeremo i computer quantistici?

Figura 1.2 Modi in cui vorremmo poter utilizzare i computer quantistici.Fumetto utilizzato con il permesso di xkcd.com.

È importante comprendere sia le potenzialità sia i limiti dei computer quantistici, soprattutto dato il clamore che circonda il quantum computing. Molti degli equivoci alla base di questo clamore derivano dall'estrapolazione di analogie ben oltre il punto in cui perdono senso: tutte le analogie hanno i loro limiti e il quantum computing non sfugge alla regola. Simulare il modo in cui un programma quantistico agisce nella pratica può essere un ottimo modo per sottoporre a test e perfezionare la comprensione fornita dalle analogie. Tuttavia, in questo libro utilizzeremo ancora varie analogie, in quanto possono aiutare a fornire idee su come funziona il quantum computing.

SUGGERIMENTO

Se avete letto descrizioni dei nuovi risultati nel quantum computing che dicono qualcosa come "Possiamo teletrasportare gatti che esistono contemporaneamente in due posti utilizzando la potenza di infiniti universi paralleli che collaborano, tutti insieme, per curare il cancro", allora avete toccato con mano il pericolo di eseguire estrapolazioni ben lontano da dove le analogie possono essere utili.

Un punto di confusione particolarmente comune riguardo al quantum computing riguarda il modo in cui gli utenti useranno i computer quantistici. Come società, siamo arrivati a capire che cos'è un *computer*: qualcosa che potete usare per eseguire applicazioni web, scrivere documenti ed eseguire simulazioni. In effetti, i computer classici svolgono talmente tante attività nella nostra vita che non sempre è facile capire che cosa sia e non sia un computer. Cory Doctorow ha compiuto questa osservazione notando che "La tua automobile è un computer in cui ti siedi" (DrupalCon Amsterdam 2014, www.youtube.com/watch?v=iaf3Sl2r3jE).

I computer quantistici, al contrario, probabilmente saranno molto più specializzati: ci aspettiamo che i computer quantistici siano inutili per alcune attività. Un ottimo modello per il modo in cui il quantum computing si adatterà ai nostri problemi di elaborazione classici sono le GPU. Le GPU sono dispositivi hardware specializzati, progettati per velocizzare determinati tipi di calcoli come il disegno grafico, le attività di machine learning e qualsiasi cosa sia facilmente parallelizzabile. Certamente vorrete avere una GPU per quelle attività specifiche, ma probabilmente non vorrete usarla per tutto, poiché abbiamo CPU molto più flessibili per le attività generali, come il controllo della posta elettronica. I computer quantistici saranno esattamente la stessa cosa: saranno ottimi per accelerare determinati tipi di attività, ma non saranno appropriati per un uso più generale.

NOTA

La programmazione di un computer quantistico prevede alcune limitazioni, quindi sarà preferibile impiegare computer classici quando non è possibile trovare un determinato vantaggio con quelli quantistici.

La computer science classica sarà ancora in circolazione e sarà il modo con cui comunicheremo e interagiremo fra noi, ma sarà affiancata da hardware quantistico. Anche per far sì che le risorse

classiche si interfaccino con i dispositivi quantistici, nella maggior parte dei casi avremo anche bisogno di un processore di segnale da digitale ad analogico, come illustrato nella Figura 1.3.

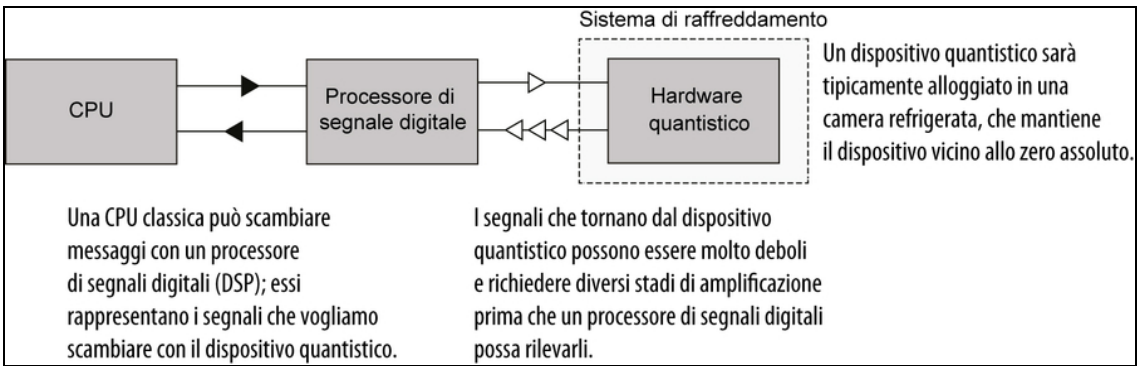


Figura 1.3 Un esempio del modo in cui un dispositivo quantistico potrebbe interagire con un computer classico attraverso l'uso di un processore di segnali digitali (DSP). Il DSP invia segnali a bassa potenza al dispositivo quantistico e amplifica i segnali a bassissima potenza che tornano dal dispositivo.

Inoltre, la fisica quantistica descrive entità esistenti su scale molto piccole (sia per dimensioni sia per energia) e ben isolate dall'ambiente circostante. Questo pone alcune forti limitazioni agli ambienti in cui possiamo collocare un computer quantistico. Una possibile soluzione consiste nel mantenere i dispositivi quantistici in apparecchi criogenici, spesso vicino a zero assoluto (-273,15°C). Mentre questo può non rappresentare un problema in un data center, la manutenzione di un refrigeratore a diluizione non è davvero sensata in un computer desktop, tanto meno in un laptop o in un telefono cellulare. Per questo motivo, i computer quantistici verranno probabilmente utilizzati a lungo attraverso il cloud, almeno per un bel po' di tempo dopo che saranno disponibili per la prima volta in commercio.

L'utilizzo di computer quantistici come servizio cloud somiglia ad altri progressi nell'hardware di calcolo specializzato. Centralizzando le risorse di calcolo più costose nei data center, è possibile esplorare

modelli di elaborazione che sarebbero difficili da distribuire in utilizzo locale, se non per gli utenti più grandi.

- Hardware di gioco specializzato (PlayStation Now, Xbox One).
- Cluster di elaborazione a latenza estremamente bassa e ad alte prestazioni (per esempio Infiniband) per problemi scientifici.
- Enormi cluster di GPU.
- Hardware riprogrammabile (per esempio Catapult/Brainwave).
- Cluster di unità di elaborazione tensoriale (TPU).
- Archiviazione a elevata permanenza e latenza (per esempio Amazon Glacier).

In futuro, i servizi cloud come Azure Quantum (https://azure.com/quantum) renderanno disponibile la potenza del quantum computing più o meno allo stesso modo.

Proprio come le connessioni Internet ad alta velocità e ad alta disponibilità hanno reso il cloud computing accessibile a un gran numero di utenti, saremo in grado di utilizzare computer quantistici da una sdraio in spiaggia o da un caffè, grazie al WiFi, o anche da un treno, mentre ammiriamo maestose catene montuose, in lontananza.

Che cosa possono fare i computer quantistici?

Come programmatori quantistici, se abbiamo un determinato problema, *come facciamo a sapere che ha senso risolverlo con un computer quantistico?*

Stiamo ancora scoprendo la portata esatta di ciò di cui sono capaci i computer quantistici, e quindi non abbiamo ancora regole concrete per rispondere a questa domanda. Finora, abbiamo trovato alcuni esempi di problemi in cui i computer quantistici offrono vantaggi significativi rispetto agli approcci classici più noti. In tutti i casi, tutti gli algoritmi quantistici che sono stati sviluppati per risolvere questi problemi

sfruttano gli effetti quantistici per ottenere tali vantaggi, motivo per cui si parla di un *vantaggio quantistico*. Ecco due esempi di algoritmi quantistici.

- L'algoritmo di Grover (discusso nel Capitolo 11) esegue ricerche in una lista di N elementi in $\sqrt{N}$ passaggi.
- L'algoritmo di Shor (Capitolo 12) calcola rapidamente numeri interi di grandi dimensioni, come quelli utilizzati dalla crittografia per proteggere i dati privati.

Ne vedremo molti altri in questo libro, ma gli algoritmi di Grover e Shor sono buoni esempi di come lavorano gli algoritmi quantistici: ognuno di essi impiega effetti quantistici per separare le risposte corrette dei problemi computazionali dalle soluzioni non valide. Un modo per realizzare un vantaggio quantistico consiste nel trovare modi per sfruttare gli effetti quantistici per separare le soluzioni corrette da quelle errate nei problemi classici.

Quali sono i vantaggi quantistici?

Gli algoritmi di Grover e Shor illustrano due tipi distinti di vantaggi quantistici. La fattorizzazione di numeri interi potrebbe essere più semplice di quanto sospettiamo. Sono molti ad aver tentato di scomporre rapidamente i numeri interi, e non ci sono riusciti, ma ciò non significa che possiamo *dimostrare* che la fattorizzazione sia difficile. D'altra parte, possiamo dimostrare che l'algoritmo di Grover è più veloce di *qualsiasi* algoritmo classico; il problema è che utilizza un diverso tipo di input.

Trovare un vantaggio *dimostrabile* per un problema pratico è un'area di ricerca attiva nel quantum computing. Detto questo, i computer quantistici possono essere strumenti potenti per risolvere i problemi, anche se non possiamo dimostrare che verrà mai trovato un algoritmo classico migliore. Dopotutto, l'algoritmo di Shor sfida le ipotesi che sono alla base di molte applicazioni di sicurezza delle informazioni: è necessaria una dimostrazione matematica solo perché non abbiamo ancora costruito un computer quantistico abbastanza grande da eseguire l'algoritmo di Shor.

I computer quantistici offrono vantaggi significativi anche per la simulazione delle proprietà dei sistemi quantistici, con applicazioni nei campi della chimica quantistica e della scienza dei materiali. Per esempio, i computer quantistici potrebbero rendere molto più facile lo studio delle energie dello stato fondamentale dei sistemi chimici. Queste energie dello stato fondamentale forniscono quindi informazioni su tassi di reazione, configurazioni degli elettroni, proprietà termodinamiche e altre proprietà di grande interesse per la chimica.

Lungo la strada per lo sviluppo di queste applicazioni, abbiamo anche intravisto vantaggi significativi nelle tecnologie derivate, come la distribuzione quantistica delle chiavi e la metrologia quantistica, alcuni dei quali saranno esaminati nei prossimi capitoli. Nell'imparare a controllare e comprendere i dispositivi quantistici per scopi di calcolo, abbiamo anche appreso tecniche preziose per l'imaging, la stima dei parametri, la sicurezza e altro ancora. Sebbene queste non siano applicazioni di quantum computing in senso stretto, sono molto utili per mostrare i valori del *ragionamento* in termini di calcolo quantistico.

Naturalmente, diventa molto più facile scoprire nuove applicazioni dei computer quantistici quando abbiamo una comprensione concreta di come funzionano gli algoritmi quantistici e di come costruire nuovi algoritmi a partire dai principi di base. Da quel punto di vista, la programmazione quantistica è un'ottima risorsa per imparare a scoprire applicazioni interamente nuove.

Che cosa non possono fare i computer quantistici?

Come altre forme di hardware specializzato, i computer quantistici non sono "buoni per tutto". Per alcuni problemi, i computer classici saranno, semplicemente, più adatti al compito. Nello sviluppo di

applicazioni per dispositivi quantistici, è utile parlare di quali attività e problemi non rientrano nell'ambito del quantum computing.

La "versione breve" è che non abbiamo regole rigide per decidere rapidamente quali attività sono più adatte ai computer classici e quali possono trarre vantaggio dai computer quantistici. Per esempio, i requisiti di archiviazione e larghezza di banda per le applicazioni in stile big data sono molto difficili da mappare su dispositivi quantistici, dove potremmo avere a disposizione un sistema quantistico relativamente piccolo. Gli attuali computer quantistici possono registrare solo input di poche decine di bit e questa limitazione diventerà più rilevante man mano che i dispositivi quantistici verranno utilizzati per compiti sempre più impegnativi. Sebbene ci aspettiamo di costruire in futuro sistemi quantistici molto più grandi di quanto possiamo ora, i computer classici saranno probabilmente sempre preferibili per quei problemi la cui risoluzione richiede grandi quantità di input/output.

Allo stesso modo, le applicazioni di machine learning che dipendono fortemente dall'accesso casuale a grandi serie di input classici sono concettualmente difficili da risolvere con il quantum computing. Detto questo, *potrebbero* esserci altre applicazioni di machine learning che si mappano in modo molto più naturale sul quantum computing. Gli sforzi di ricerca per trovare i modi migliori per applicare le risorse quantistiche a risolvere attività di machine learning sono ancora in corso. In generale, i problemi che hanno piccoli dati di input e output ma richiedono grandi quantità di calcolo per passare dall'input all'output sono buoni candidati per i computer quantistici.

Alla luce di queste sfide, si potrebbe essere tentati di concludere che i computer quantistici eccellono sempre in compiti che hanno piccoli input e output ma molto calcolo intenso per passare dall'uno all'altro. Concetti come il *parallelismo quantistico* sono ampiamente presenti

nei media e talvolta i computer quantistici sono persino descritti come se utilizzassero universi paralleli per i loro calcoli.

> **NOTA**
>
> Il concetto di "universi paralleli" è un ottimo esempio di analogia che può aiutare a rendere comprensibili i concetti quantistici, ma che può anche portare a vere sciocchezze, se portata al suo estremo. A volte può essere utile pensare che le diverse parti di un calcolo quantistico si trovino in "universi" differenti, che non possono influenzarsi a vicenda, ma questa descrizione non aiuta a pensare ad alcuni degli effetti che tratteremo in questo libro, come l'interferenza. Se portata troppo lontano, l'analogia degli universi paralleli porta a pensare al quantum computing in modi che sono più vicini a un episodio particolarmente intrigante e divertente di una serie di fantascienza come Star Trek che alla realtà.

Ciò che questo non riesce a comunicare, tuttavia, è che non sempre è ovvio capire come utilizzare gli effetti quantistici per estrarre risposte utili da un dispositivo quantistico, anche se lo stato del dispositivo quantistico sembra contenere l'output desiderato. Per esempio, un modo per fattorizzare un intero N utilizzando un computer classico è elencare ogni fattore *potenziale* e verificare se è effettivamente un fattore o meno:

1. Esegui l'assegnamento $i = 2$.
2. Verifica se il resto di N / i è 0.
   - In tal caso, restituisci che i è un fattore di N.
   - In caso contrario, incrementa i e ripeti il ciclo.

Possiamo accelerare questo algoritmo classico utilizzando una grande quantità di computer classici, uno per ogni fattore potenziale che vogliamo provare. Cioè, questo problema può essere facilmente parallelizzato. Un computer quantistico può provare ogni fattore potenziale all'interno dello stesso dispositivo, ma, a quanto pare, questo non è ancora sufficiente per fattorizzare gli interi più velocemente dell'approccio classico. Se utilizziamo questo approccio

su un computer quantistico, l'output sarà uno dei potenziali fattori scelti a caso. I fattori effettivamente corretti si verificheranno con probabilità di circa $1 / \sqrt{N}$, un risultato non migliore dell'algoritmo classico.

Come vedremo nel Capitolo 12, però, possiamo usare altri effetti quantistici per fattorizzare i numeri interi con un computer quantistico più velocemente rispetto ai più noti algoritmi classici. Gran parte del lavoro pesante svolto dall'algoritmo di Shor consiste nell'assicurarsi che la probabilità di misurare un fattore corretto alla fine sia molto maggiore della probabilità di misurare un fattore errato. Cancellare le risposte errate in questo modo è il punto in cui entra in gioco gran parte dell'arte della programmazione quantistica; è non è facile o addirittura nemmeno possibile farlo per tutti i problemi che potremmo voler risolvere.

Per capire più concretamente che cosa possono e non possono fare i computer quantistici e come fare cose interessanti con i computer quantistici nonostante queste sfide, è utile adottare un approccio più concreto. Quindi, cerchiamo di prendere in considerazione ciò che un programma quantistico *effettivamente è*, in modo da poter iniziare a scrivere i nostri programmi.

Che cos'è un programma?

In tutto il libro, troveremo spesso utile spiegare un concetto quantistico riesaminando prima l'analogo concetto classico. In particolare, facciamo un passo indietro ed esaminiamo che cos'è un programma classico.

DEFINIZIONE

Un *programma* è una sequenza di istruzioni che può essere interpretata da un computer classico per eseguire il compito desiderato. I moduli fiscali, le

indicazioni stradali, le ricette e gli script Python sono tutti esempi di programmi.

Possiamo scrivere programmi classici per suddividere un'ampia varietà di compiti per consentirne l'interpretazione da parte di tutti i tipi di computer. La Figura 1.4 mostra alcuni esempi di programmi.

Vediamo l'aspetto di un semplice programma "hello world" in Python:

```
>>> def hello():
...     print ("Hello, world!")
...
>>> hello()
Hello, world!
```

Nella sua forma più elementare, questo programma può essere considerato come una sequenza di istruzioni fornite all'interprete Python, che quindi lo esegue un'istruzione per volta per ottenere qualche effetto, in questo caso, mostrare un messaggio sullo schermo. Il programma è quindi una descrizione di un'attività che viene poi *interpretata* da Python e, a sua volta, dalla CPU per raggiungere l'obiettivo. Questa interazione tra la descrizione e l'interpretazione spinge a chiamare Python, C e altri strumenti di programmazione simili con il nome di *linguaggi*, enfatizzando la programmazione è il modo in cui comunichiamo con i nostri computer.

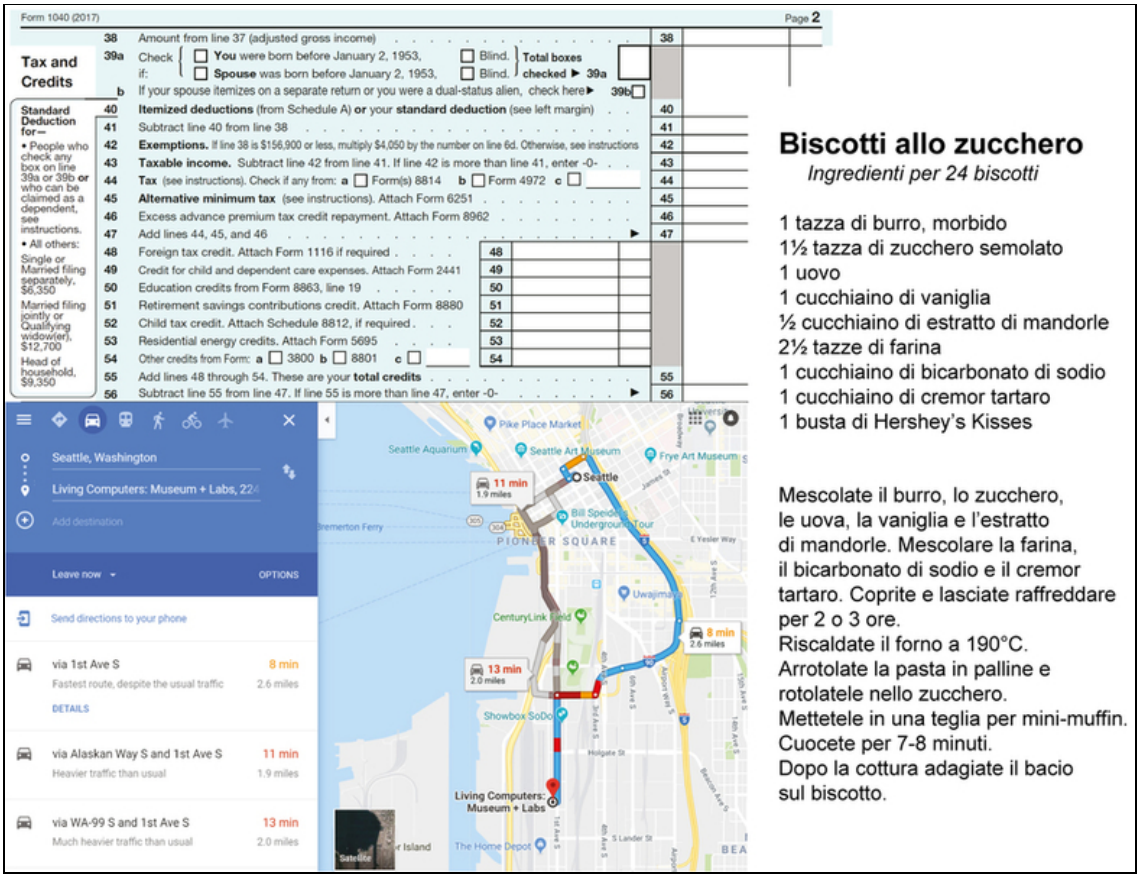


Figura 1.4 Esempi di programmi classici. Moduli fiscali, indicazioni stradali e ricette sono tutti esempi in cui una sequenza di istruzioni deve essere interpretata da quel "calcolatore" classico, che è una persona. Possono sembrare programmi molto diversi, ma ognuno di essi utilizza un elenco di passaggi per comunicare una procedura.

Nell'esempio dell'utilizzo di Python per mostrare "Hello, world!" stiamo effettivamente comunicando con Guido van Rossum, il progettista fondatore del linguaggio Python. Guido, a sua volta, comunica in pratica per nostro conto con i progettisti del sistema operativo che stiamo utilizzando. Questi progettisti a loro volta comunicano per nostro conto con Intel, AMD, ARM o qualsiasi altra azienda abbia progettato la CPU che stiamo utilizzando e così via.

Che cos'è un programma quantistico?

Come i programmi classici, i programmi quantistici sono costituiti da sequenze di istruzioni che vengono interpretate dai computer classici per eseguire un determinato compito. La differenza, tuttavia, è che in un programma quantistico, il compito che desideriamo svolgere viene svolto da un sistema quantistico.

Di conseguenza, anche le istruzioni utilizzate nei programmi classici e quantistici differiscono. Un programma classico può descrivere un'attività come il caricamento da Internet di alcune immagini di gatti in termini di istruzioni per uno stack di rete e alla fine in termini di istruzioni assembler, come `mov`. Al contrario, linguaggi quantistici come Q# consentono ai programmatori di esprimere compiti quantistici in termini di istruzioni come `M` (misura). Quando vengono eseguiti utilizzando hardware quantistico, questi programmi possono chiedere a un elaboratore di segnali digitali di inviare microonde, onde radio o laser a un dispositivo quantistico e di amplificare poi i segnali che escono dal dispositivo.

Nel resto di questo libro, vedremo molti esempi dei tipi di compiti che un programma quantistico deve risolvere o, quanto meno, affrontare, e scopriremo quali tipi di strumenti classici possiamo impiegare per agevolare la programmazione quantistica. La Figura 1.5 mostra un esempio di scrittura di un programma quantistico in Visual Studio Code, un classico ambiente di sviluppo integrato (IDE).

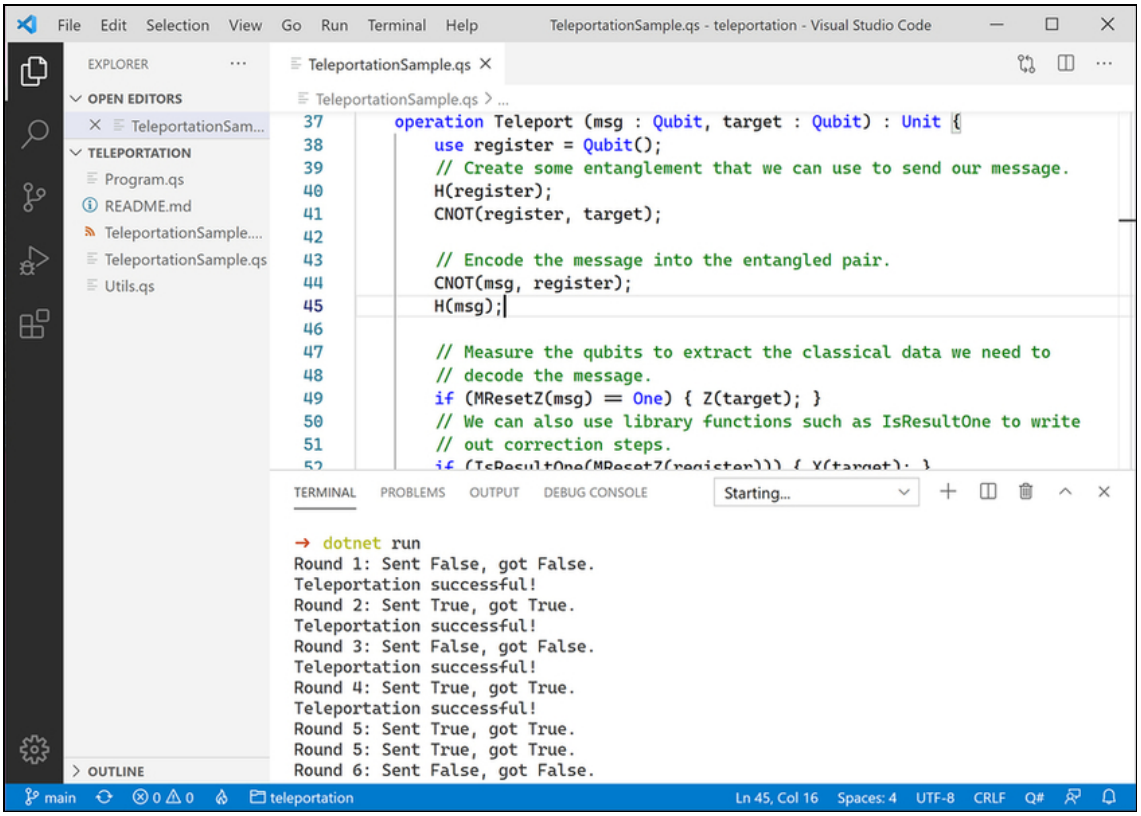


Figura 1.5 Scrittura di un programma quantistico con Quantum Development Kit e Visual Studio Code. Esamineremo il contenuto di questo programma nel Capitolo 7, ma già a questo livello noterete certamente la similitudine con altri progetti software sui quali potreste aver lavorato.

Capitolo per capitolo svilupperemo i concetti di cui abbiamo bisogno per poter scrivere programmi quantistici; la Figura 1.6 mostra una tabella di marcia. Nel prossimo capitolo, scopriremo gli elementi di base che compongono un computer quantistico e li utilizzeremo per scrivere il nostro primo programma quantistico.

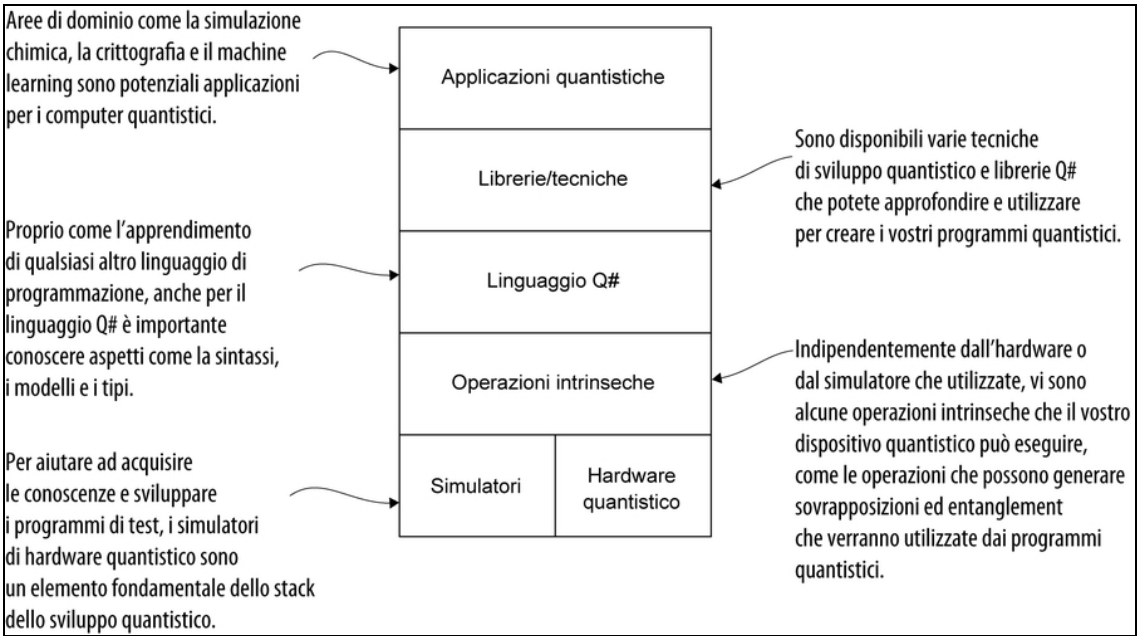


Figura 1.6 Questo libro sviluppa i concetti di cui abbiamo bisogno per scrivere programmi quantistici. La Parte I offre descrizioni di basso livello dei simulatori e delle operazioni intrinseche (l'API hardware) tramite la costruzione di un simulatore in Python. La Parte II esamina il linguaggio Q# e le tecniche di sviluppo quantistico che ci aiuteranno a sviluppare le nostre applicazioni. La Parte III mostra alcune applicazioni note di quantum computing e le sfide e opportunità che ci offre questa nuova tecnologia.

Riepilogo

- Il quantum computing è importante perché i computer quantistici ci consentiranno, potenzialmente, di risolvere problemi troppo difficili per i computer convenzionali.
- I computer quantistici possono fornire vantaggi rispetto ai computer classici per alcune classi di problemi, come la fattorizzazione di grandi numeri.
- I computer quantistici sono dispositivi che utilizzano la fisica quantistica per elaborare i dati.

- I programmi sono sequenze di istruzioni che possono essere interpretate da un computer classico per eseguire determinate attività.
- I programmi quantistici sono programmi che eseguono calcoli inviando istruzioni a dispositivi quantistici.

I qubit: gli elementi costitutivi

In questo capitolo cominceremo almeno a fare la conoscenza di alcuni concetti di programmazione quantistica. Il concetto principale che esploreremo è il *qubit*, l'analogo quantistico di un bit classico. Useremo i qubit come astrazione o modello per descrivere i nuovi tipi di calcolo resi possibili dalla fisica quantistica. La Figura 2.1 mostra un modello di utilizzo di un computer quantistico e la configurazione del simulatore che useremo in questo libro. I qubit, reali o simulati, si troveranno sulla macchina target e interagiranno con i programmi quantistici che scriveremo. Questi programmi quantistici possono essere inviati da vari programmi host, che poi attenderanno di ricevere i risultati dal programma quantistico.

Per scoprire che cosa sono i qubit e come si interagisce con essi, utilizzeremo un esempio del modo in cui vengono utilizzati oggi: per la generazione di numeri casuali. Mentre a partire dai qubit possiamo costruire dispositivi molto più interessanti, il semplice esempio di un generatore quantistico di numeri casuali è un buon modo per familiarizzare con il qubit.

Perché abbiamo bisogno dei numeri casuali?

Agli esseri umani piacciono le certezze. Ci piace che quando premiamo un tasto alla tastiera succeda ogni volta la stessa cosa.

Tuttavia, ci sono alcuni contesti in cui abbiamo bisogno di casualità.

- Nei giochi.
- Nella simulazione di sistemi complessi (come il mercato azionario).
- Nella scelta di segreti sicuri (per esempio, password e chiavi crittografiche).

In tutte queste situazioni in cui cerchiamo la casualità, possiamo descrivere le probabilità di ciascun esito. Per gli eventi casuali, descrivere le probabilità è tutto ciò che possiamo dire sulla situazione, fino a quando viene lanciato il dado (o viene lanciata la moneta o viene riutilizzata la password). Quando descriviamo le probabilità di ciascun esempio, possiamo dire qualcosa come:

- "*Se* lancio questo dado, *allora* otterrò un 6 con *una probabilità di* 1 su 6."
- "*Se* lancio questa moneta, *allora* otterrò testa con *una probabilità di* 1 su 2."

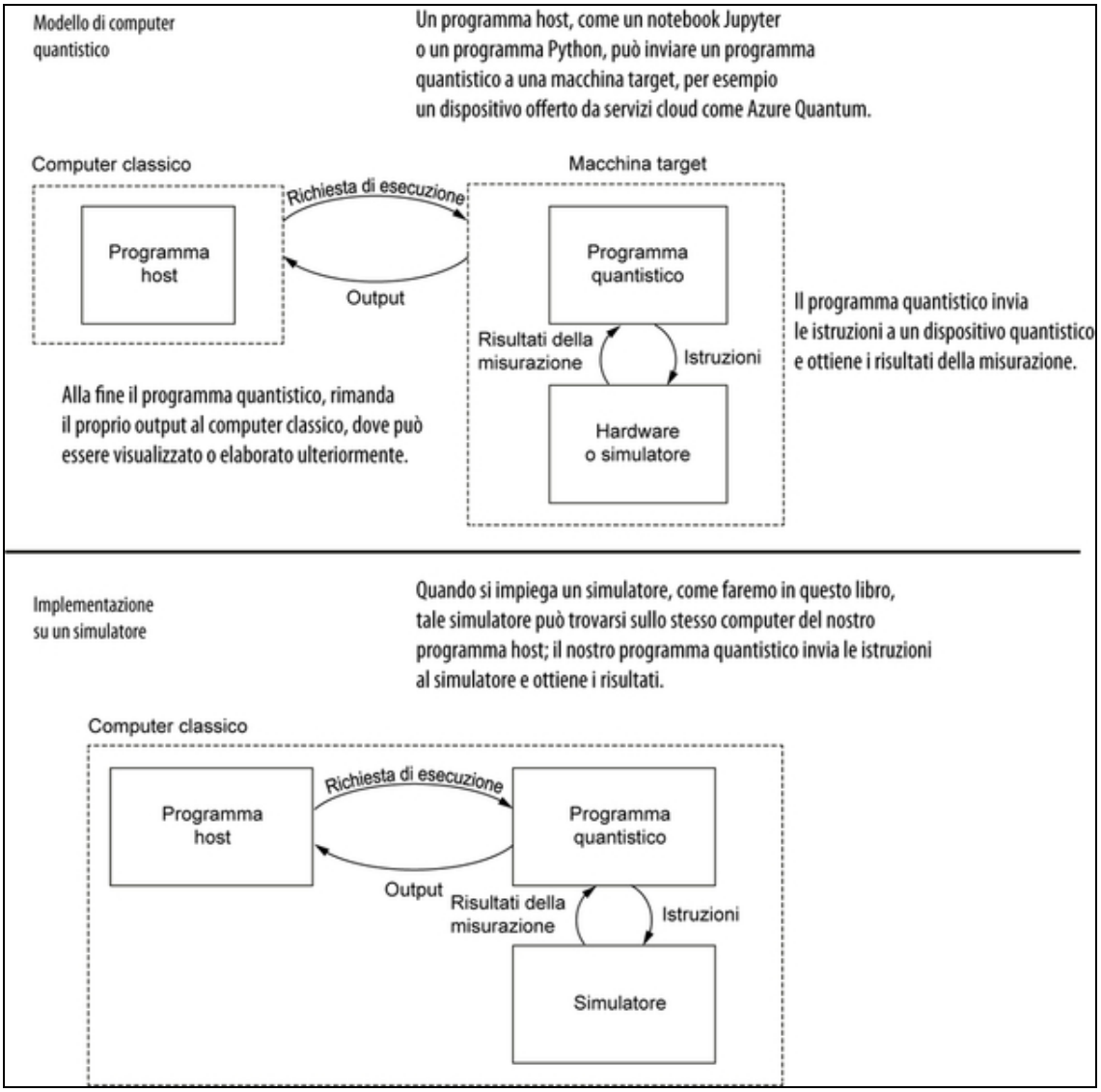


Figura 2.1 Un modello del modo in cui possiamo usare un computer quantistico. La metà superiore della figura rappresenta il modello generale di un computer quantistico. In questo libro impiegheremo simulatori locali e, appunto, la metà inferiore rappresenta ciò che costruiremo e utilizzeremo.

Possiamo anche descrivere casi in cui le probabilità non sono le stesse per ogni risultato. Sulla *Ruota della Fortuna* (Figura 2.2) la probabilità che *se* giriamo la ruota, *allora* otteniamo un premio di un milione di dollari è molto inferiore alla probabilità che *se* giriamo la ruota, *allora* perderemo il denaro.

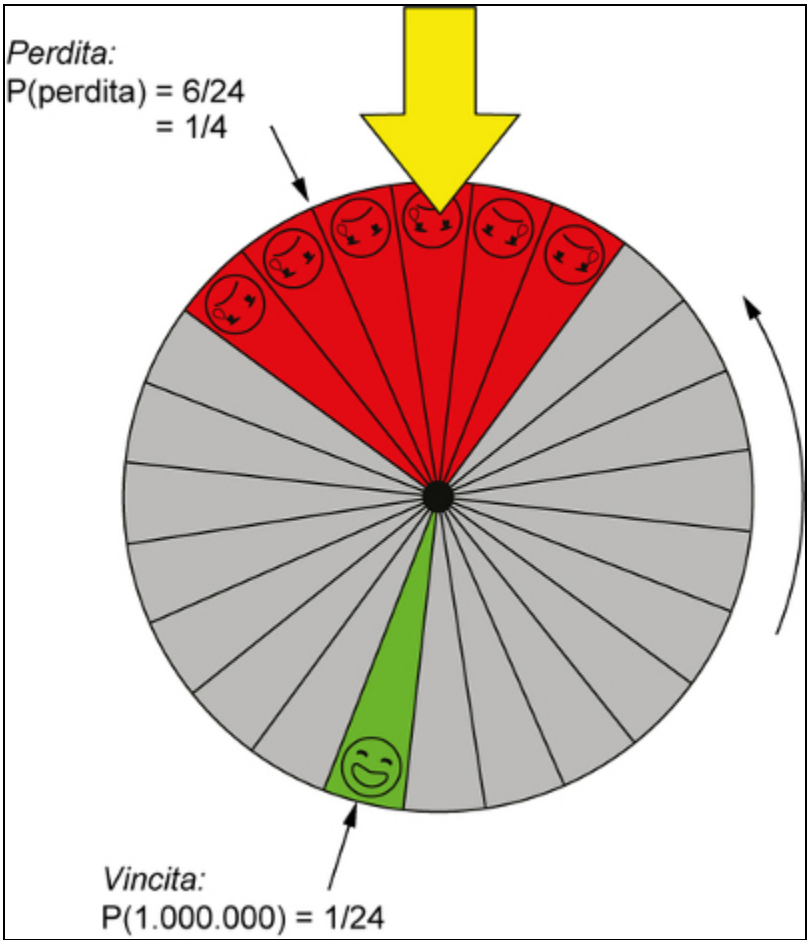


Figura 2.2 Le probabilità di vincere un milione di dollari e di perdere il nostro denaro sulla Ruota della Fortuna. Prima di girare la ruota, non conosciamo l'esito, ma osservando la ruota sappiamo che la probabilità di perdere è molto maggiore della probabilità di vincere.

Come nei game show, anche nella computer science ci sono molti contesti in cui la casualità è fondamentale, specialmente quando è richiesta la sicurezza. Se vogliamo proteggere alcune informazioni, impieghiamo la crittografia, che ci consente di combinare i nostri dati con dei numeri casuali in vari modi. Se il nostro generatore di numeri casuali non è molto efficace – vale a dire, se un malintenzionato può prevedere quali numeri usiamo per proteggere i nostri dati privati – la crittografia non ci aiuta molto. Immaginiamo di utilizzare un

generatore di numeri casuali scadente per gestire una lotteria; un malintenzionato che dovesse scoprire come vengono generati i nostri numeri casuali potrebbe portarci rapidamente alla bancarotta.

Che cosa sono le probabilità?
Possiamo perdere molto denaro impiegando numeri casuali che i nostri avversari possono prevedere. Chiedetelo ai produttori dello show *Press Your Luck!*, molto popolare negli anni Ottanta.
Un concorrente ha scoperto di poter prevedere dove si sarebbe fermata la nuovissima "ruota" elettronica del gioco, il che gli ha permesso di vincere più di 250.000 dollari (in valore attuale). Per saperne di più, vedi *The Man Who Got No Whammies*, di Zachary Crockett su https://priceonomics.com/the-man-who-got-no-whammies.

Come vedremo, la meccanica quantistica ci consente di costruire alcune sorgenti di numeri casuali davvero uniche. Se le costruiamo davvero bene, la casualità dei nostri risultati è garantita dalla fisica, non da un'ipotesi su quanto tempo impiegherebbe un computer a risolvere un problema difficile. Ciò significa che un hacker dovrebbe infrangere le leggi della fisica per poter infrangere la sicurezza. Questo non significa che dovremmo usare numeri casuali quantistici per tutto; gli esseri umani sono ancora l'anello più debole nelle infrastrutture di sicurezza 😈.

Approfondimento: sicurezza computazionale e sicurezza teorica delle informazioni
Alcuni modi per proteggere le informazioni private si basano su ipotesi su quali problemi siano facili o difficili da risolvere per un malintenzionato. Per esempio, RSA è un algoritmo di crittografia comunemente usato e si basa sulla difficoltà di trovare i fattori primi di grandi numeri. L'algoritmo RSA viene utilizzato sul Web e in altri contesti per proteggere i dati degli utenti, partendo dal presupposto che gli avversari non possono calcolare facilmente dei numeri molto grandi. Finora, questo si è dimostrato un buon presupposto, ma è del tutto possibile che venga scoperto un nuovo algoritmo di fattorizzazione, il quale minerà la sicurezza dell'algoritmo RSA. I nuovi modelli di calcolo, come il quantum computing, cambiano la scena

anche in termini di quanto sia ragionevole o irragionevole formulare ipotesi computazionali come "la fattorizzazione è un problema difficile". Come vedremo nel Capitolo 11, un algoritmo quantistico, l'*algoritmo di Shor*, consente di risolvere alcuni problemi di crittografia molto più velocemente rispetto ai computer classici, sfidando le ipotesi comunemente impiegate per promettere la sicurezza computazionale.

Al contrario, se un malintenzionato può solo indovinare casualmente i segreti, magari contando su grandi potenze di calcolo, allora un sistema di sicurezza fornirà garanzie molto migliori in termini di capacità di proteggere le informazioni private. Tali sistemi sono detti *informativamente sicuri*. Più avanti in questo capitolo, vedremo che la generazione di numeri casuali in un modo difficile da prevedere ci consente di implementare una procedura informativamente sicura chiamata *pad monouso*.

Questo ci dà una certa sicurezza: che possiamo usare numeri casuali quantistici per attività vitali come proteggere i dati privati, gestire lotterie e giocare a *Dungeons and Dragons*. Simulare il funzionamento dei generatori quantistici di numeri casuali ci consente di apprendere molti dei concetti che sono alla base della meccanica quantistica, quindi… cominciamo!

Come abbiamo accennato in precedenza, un ottimo modo per iniziare è osservare un esempio di un programma quantistico che genera numeri casuali: un generatore quantistico di numeri casuali (QRNG). Non preoccupatevi se il seguente algoritmo (mostrato anche nella Figura 2.3) non sembra avere molto senso in questo momento: spiegheremo le sue parti nel prosieguo del capitolo.

1. Chiedere al dispositivo quantistico di allocare un qubit.
2. Applicare al qubit un'istruzione chiamata *istruzione di Hadamard*; ne parleremo più avanti nel capitolo.
3. Misurare il qubit e restituire il risultato.

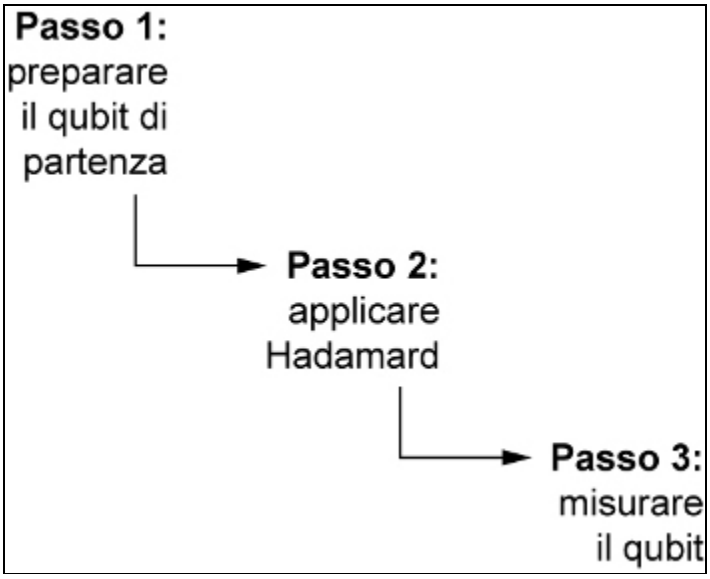


Figura 2.3 Algoritmo del generatore quantistico di numeri casuali. Per campionare numeri casuali con un computer quantistico, il nostro programma preparerà un nuovo qubit e quindi utilizzerà un'istruzione di Hadamard per preparare la sovrapposizione di cui abbiamo bisogno. Infine, possiamo misurare e restituire il risultato casuale che otteniamo.

Nel resto del capitolo, svilupperemo una classe Python chiamata `QuantumDevice` che ci consente di scrivere programmi che implementano questo tipo di algoritmi. Una volta che avremo una classe `QuantumDevice`, saremo in grado di scrivere un generatore quantistico di numeri casuali come un programma Python simile ai programmi classici cui siamo abituati.

> **NOTA**
> Vedi l'Appendice A per le istruzioni su come configurare Python sul dispositivo per eseguire programmi quantistici.

Notate che il seguente esempio non verrà eseguito finché non avrete scritto il simulatore presentato in questo capitolo 😊.

Listato 2.1 qrng.py: un programma quantistico che genera numeri casuali.

```
def qrng(device : QuantumDevice) -> bool: (1)
    with device.using_qubit() as q: (2)
```

```
    qh() (3)
    return q.measure() (4)
```

(1) I programmi quantistici sono scritti proprio come i programmi classici. In questo caso stiamo impiegando Python, quindi il nostro programma quantistico è una funzione Python `qrng` che implementa un QRNG.
(2) I programmi quantistici funzionano chiedendo dei qubit all'hardware di quantum computing: i qubit sono gli analoghi quantistici dei bit che usiamo normalmente per eseguire calcoli.
(3) Una volta che abbiamo un qubit, possiamo eseguire istruzioni su quel qubit. Come nei linguaggi assembly, queste istruzioni possono essere denotate da abbreviazioni; più avanti in questo capitolo vedremo che cosa rappresenta h().
(4) Per recuperare i dati dai nostri qubit, possiamo misurarli. In questo caso, la metà delle volte la nostra misurazione darà `true` e l'altra metà delle volte darà `false`.

Ecco fatto! In quattro passaggi abbiamo creato il nostro primo programma quantistico. Questo QRNG restituisce `true` o `false`. In termini di Python, questo significa che otteniamo un `1` o uno `0` ogni volta che eseguiamo `qrng`. Non è un generatore di numeri casuali molto sofisticato, ma il numero che restituisce è veramente casuale.

Per eseguire il programma `qrng`, abbiamo bisogno di dare alla funzione un `QuantumDevice` che fornisca l'accesso ai qubit e implementi le diverse istruzioni che possiamo inviare ai qubit. Anche se, per cominciare, abbiamo bisogno di un solo qubit, fra poco costruiremo il nostro simulatore di computer quantistico. Potrebbe certamente essere utilizzato l'hardware esistente per questo compito così modesto, ma ciò che vedremo più avanti andrà ben oltre l'ambito dell'hardware attualmente disponibile. Funzionerà localmente su un laptop o desktop ma si comporterà come un vero hardware quantistico. Per tutto il resto

del capitolo, costruiremo i vari componenti di cui abbiamo bisogno per scrivere il nostro simulatore ed eseguire `qrng`.

Che cosa sono i bit classici?

Quando si studiano i concetti della meccanica quantistica, spesso è utile fare un passo indietro e riesaminare i concetti *classici* per stabilire un legame con il modo in cui essi sono espressi nel quantum computing. Detto questo, parliamo ancora una volta di *bit*.

Supponiamo di voler inviare alla nostra amica Eve un messaggio importante, per esempio "💔". Come possiamo rappresentare il nostro messaggio in modo che possa essere inviato facilmente?

Potremmo iniziare facendo un elenco di ogni lettera e simbolo che potremmo usare per scrivere messaggi. Fortunatamente, Unicode Consortium (`https://unicode.org`) lo ha già fatto per noi e ha assegnato un *codice* a un'ampia varietà di caratteri utilizzati per la comunicazione in tutto il mondo. Per esempio, alla lettera *I* è stato assegnato il codice 0049, mentre il simbolo 🏵 è indicato con A66E, ✙ con 2E0E e 💔 con 1F496. Questi codici potrebbero non sembrare utili, a prima vista, ma lo sono se pensiamo di inviare ogni simbolo in un messaggio. Se sappiamo come inviare due messaggi (diamo "0" e "1"), queste ricette ci permettono di costruire messaggi più complicati, come sono "🏵", "✙" e "💔" come sequenze di messaggi "0" e "1":

0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100

5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Ora possiamo inviare quello che vogliamo, se sappiamo come inviare a Eve anche solo due messaggi: un messaggio “0” e un messaggio “1”. Utilizzando queste ricette, il nostro messaggio “💔” diventa “0001 1111 0100 1001 0110” o il codice Unicode 1F496.

SUGGERIMENTO

Fate attenzione a non inviare a Eve, per errore, il messaggio “0001 1111 0100 1001 0100”, altrimenti riceverà il messaggio “💔”!

Chiamiamo entrambi i messaggi, “0” e “1”, un *bit*.

NOTA

Per distinguere i bit dai bit quantistici di cui parliamo in tutto il resto del libro, spesso sottolineeremo che stiamo parlando di *bit classici*.

Quando usiamo la parola *bit*, generalmente possiamo intendere due cose:

- qualsiasi sistema fisico che possa essere descritto completamente rispondendo a una domanda che prevede come risposta solo vero o falso;
- le informazioni memorizzate in tale sistema fisico.

Per esempio, lucchetti, interruttori, transistor, la rotazione a sinistra o a destra di una palla e la presenza del vino in un calice sono tutti fenomeni che possono essere considerati bit, poiché possiamo usarli tutti per inviare o registrare messaggi (vedi la Tabella 2.1).

Tabella 2.1 Esempi di bit.

Etichetta	Lucchetto	Interruttore	Transistor	Calice	Baseball

0	Aperto	Spento	Bassa tensione	Contiene vino bianco	Rotazione a sinistra
1	Chiuso	Acceso	Alta tensione	Contiene vino rosso	Rotazione a destra

Questi esempi sono tutti bit, perché possiamo descriverli completamente a qualcun altro rispondendo a una singola domanda vero/falso. In altre parole, ogni esempio ci consente di inviare un messaggio 0 o 1. Come tutti i modelli concettuali, anche questo ha dei limiti: esiste anche il vino rosé, per esempio. Come lo descriveremmo?

Detto questo, un bit è uno strumento utile, perché ci permette di descrivere dei modi per interagire con un bit, i quali sono indipendenti da come costruiamo effettivamente il bit.

Che cosa possiamo fare con i bit classici?

Ora che abbiamo un modo per descrivere e inviare informazioni classiche, che cosa possiamo fare per elaborarle e modificarle? Descriviamo i modi in cui possiamo elaborare le informazioni in termini di *operazioni*, che definiamo come modi per descrivere come un modello può essere modificato o come agire su di esso.

Per visualizzare l'operazione NOT, immaginiamo di etichettare due punti come 0 e 1, come illustrato nella Figura 2.4. L'operazione NOT è quindi una qualsiasi trasformazione che trasforma i bit 0 in bit 1 e viceversa. Nei dispositivi di archiviazione classici, come i dischi rigidi, un gate NOT inverte il campo magnetico che memorizza il valore del nostro bit. Come illustrato nella Figura 2.5, possiamo pensare di implementare NOT come una rotazione di 180° tra i punti 0 e 1 che abbiamo disegnato nella Figura 2.4.

Figura 2.4 Un bit classico può trovarsi in due diversi stati, tipicamente chiamati 0 e 1. Possiamo rappresentare un bit classico come un punto nero nella posizione 0 o 1.

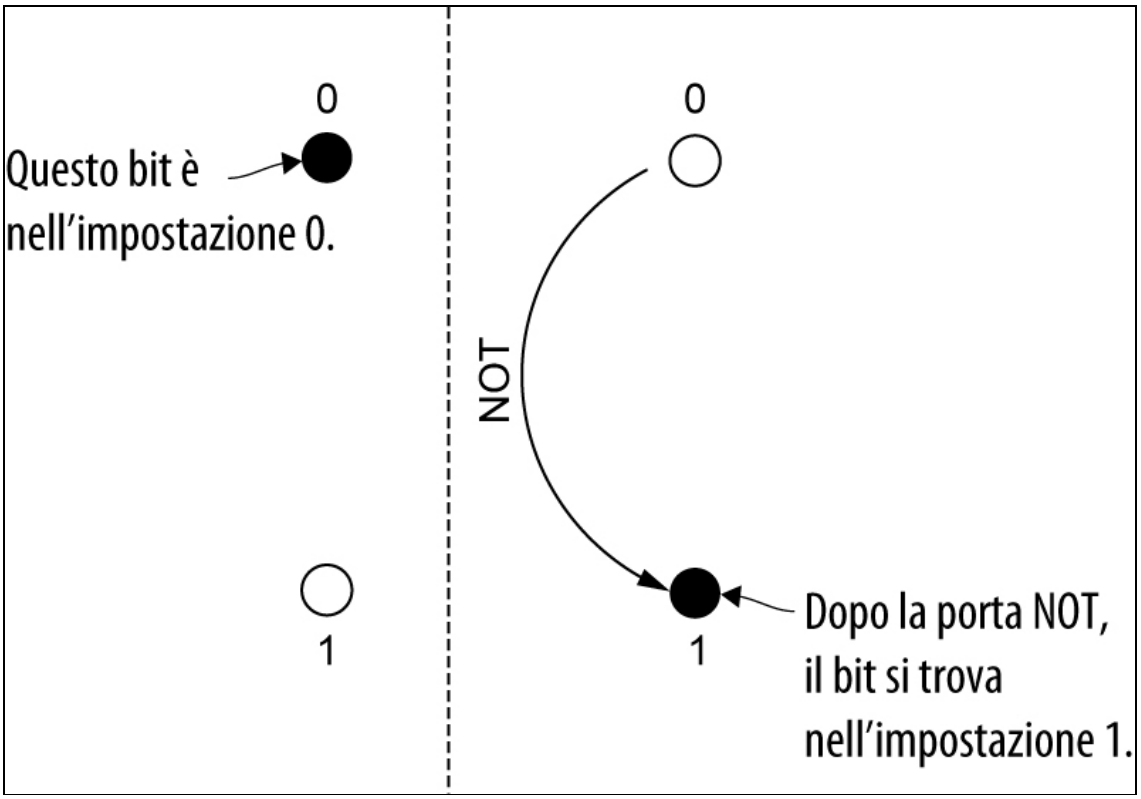


Figura 2.5 L'operazione NOT classica inverte un bit classico tra 0 e 1. Per esempio, se un bit inizia nello stato 0, NOT lo inverte nello stato 1.

Il fatto di rappresentare i bit classici in questo modo ci consente anche di estendere leggermente il concetto di bit per includere un modo per descrivere i *bit casuali* (cosa che ci sarà utile in seguito). Se abbiamo una *moneta equa* (cioè una moneta che dà testa metà delle volte e dà croce l'altra metà), allora non sarebbe corretto chiamare quella moneta uno 0 o un 1. Sappiamo solo quale bit vale la nostra moneta solo con riferimento alla superficie sulla quale poggia; possiamo anche lanciarla per ottenere un valore casuale del bit. Ogni volta che lanciamo una moneta, sappiamo che alla fine atterrerà e otterremo testa o croce. Il fatto che esca testa o croce è governato da una probabilità chiamata *bias* della moneta. Dobbiamo scegliere un lato della moneta per descrivere il bias, cosa che è facile esprimere con una domanda come: "Qual è la probabilità che la moneta atterrerà dando testa?". Quindi una moneta equa avrà un bias del 50% perché atterra dando il valore testa la metà delle volte, risultato che è mappato sul valore 0 del bit nella Figura 2.6.

Impiegando questa rappresentazione, possiamo prendere i nostri due punti precedenti, che indicano i valori 0 e 1 del bit e collegarli con una linea sulla quale possiamo tracciare il bias della nostra moneta. È facile vedere che un'operazione NOT (che funziona ancora sul nostro nuovo bit probabilistico) non fa nulla a una moneta equa. Se 0 e 1 si verificano con la stessa probabilità, non importa se trasformiamo uno 0 in un 1 o un 1 in uno 0: finiremo comunque con 0 e 1 con la stessa probabilità.

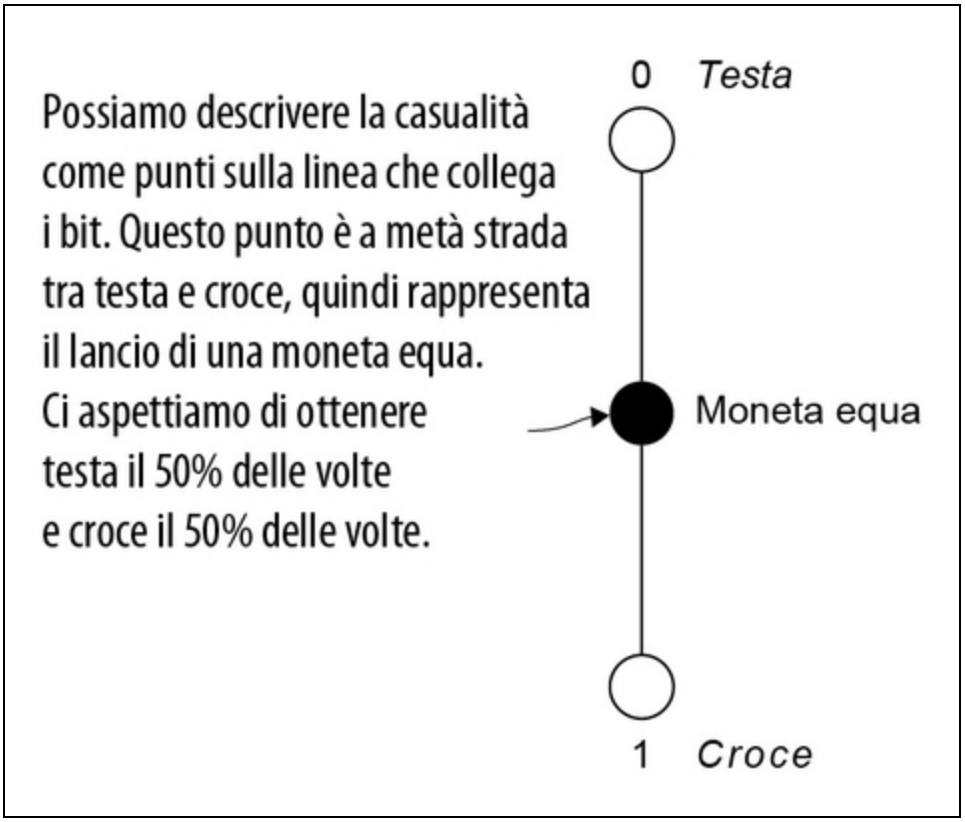


Figura 2.6 Possiamo usare la stessa immagine di prima per estendere il nostro concetto di bit per descrivere una moneta. A differenza di un bit, una moneta ha una certa probabilità di essere 0 o 1 ogni volta che viene lanciata. Rappresentiamo graficamente quella probabilità come un punto compreso tra 0 e 1.

E se il nostro bias non fosse nel mezzo? Se sappiamo che qualcuno sta cercando di imbrogliare impiegando una moneta truccata che dà quasi sempre testa, possiamo dire che il bias della moneta è del 90% e rappresentarlo sulla nostra linea tracciando un punto molto più vicino a 0 che a 1.

DEFINIZIONE

Il punto sulla linea da cui vorremmo trarre ogni bit classico è lo *stato* di quel bit.

Consideriamo uno scenario. Diciamo che voglio inviarvi un gruppo di bit memorizzati impiegando lucchetti. Qual è il modo più

economico per farlo?

Un approccio consiste nello spedirvi una scatola contenente molti lucchetti aperti o chiusi e sperare che arrivino nello stesso stato in cui li ho inviati. Ma possiamo anche convenire sul fatto che tutti i lucchetti inizino nello stato 0 (aperto) e posso inviarvi delle istruzioni riguardanti quali lucchetti chiudere. In questo modo, potete acquistare da soli i vostri lucchetti e ho solo bisogno di inviarvi una *descrizione* del modo in cui preparare i lucchetti impiegando i classici gate NOT. Inviare un foglio di carta o un'e-mail è molto più economico che spedire una scatola di lucchetti!

Questo illustra un principio su cui faremo affidamento in tutto il libro: *lo stato di un sistema fisico può anche essere descritto in termini di istruzioni su come preparare tale stato*. Pertanto, le operazioni consentite su un sistema fisico definiscono anche quali sono gli stati possibili.

Sebbene il discorso possa sembrare assolutamente banale, c'è un'altra cosa che possiamo fare con i bit classici che si rivelerà fondamentale per il modo in cui possiamo affrontare il quantum computing: possiamo guardarli. Se guardo un lucchetto e concludo: "Aha! Quel lucchetto è aperto", allora posso pensare al mio cervello come a una specie di bit particolarmente "morbido". Il messaggio 0 viene memorizzato nel mio cervello dal mio pensiero, "Aha! Quel lucchetto è aperto", mentre il messaggio 1 verrebbe memorizzato dal mio pensiero, "Ah, beh, quel lucchetto è chiuso ☹". In effetti, osservando un bit classico, l'ho *copiato* nel mio cervello. Diciamo che l'atto di *misurare* il bit classico copia quel bit.

Più in generale, la vita moderna si basa sulla facilità con cui copiamo i bit classici, osservandoli. Copiamo bit classici con grande naturalezza, misurando ogni secondo molti miliardi di bit classici, che

magari provengono dalla nostra console per videogiochi che invia il suo output al nostro televisore.

D'altra parte, se un bit viene memorizzato come una moneta, il processo di misurazione implica il suo lancio. La misurazione non copia proprio la moneta, poiché dalla misurazione potrei ottenere un risultato diverso la prossima volta che lancio la stessa moneta. Se ho solo una misurazione di una moneta, non posso conoscere la probabilità di ottenere testa o croce. Non abbiamo questa ambiguità con i bit a lucchetto, perché sappiamo che lo stato dei lucchetti è 0 o 1. Se misurassi un lucchetto e lo trovassi nello stato 0, saprei che sarebbe sempre nello stato 0, a meno di fare qualcosa al lucchetto.

La situazione non è esattamente la stessa nel quantum computing, come vedremo più avanti nel capitolo. Mentre la misurazione di informazioni classiche è abbastanza semplice ed economica, dobbiamo essere molto più attenti nell'effettuare misurazioni quantistiche.

Astrazioni a supporto

Indipendentemente da come costruiamo fisicamente un bit, possiamo (fortunatamente) rappresentarli allo stesso modo sia in matematica sia nel codice. Per esempio, Python fornisce il tipo `bool` (abbreviazione di Boolean, in onore del logico George Boole), che ha due valori validi: `True` e `False`. Possiamo rappresentare le trasformazioni sui bit, come NOT e OR, come operazioni che agiscono su variabili di tipo `bool`. È importante sottolineare che possiamo specificare un'operazione classica descrivendo il modo in cui tale operazione trasforma ogni possibile input, tramite una *tabella di verità*.

DEFINIZIONE

Una *tabella di verità* descrive l'output di un'operazione classica per ogni possibile combinazione di input. Per esempio, la Figura 2.7 mostra la tabella di verità per l'operazione AND.

Possiamo trovare la tabella della verità per l'operazione NAND (abbreviazione di NOT-AND) in Python tramite iterazioni su combinazioni di `True` e `False`.

Listato 2.2 Usare Python per mostrare una tabella di verità per NAND.

```
>>> from itertools import product
>>> for inputs in product([False, True], repeat=2):
...     output = not (inputs[0] and inputs[1])
...     print(f"{inputs[0]}\t{inputs[1]}\t->\t{output}")
False   False   ->      True
False   True    ->      True
True    False   ->      True
True    True    ->      False
```

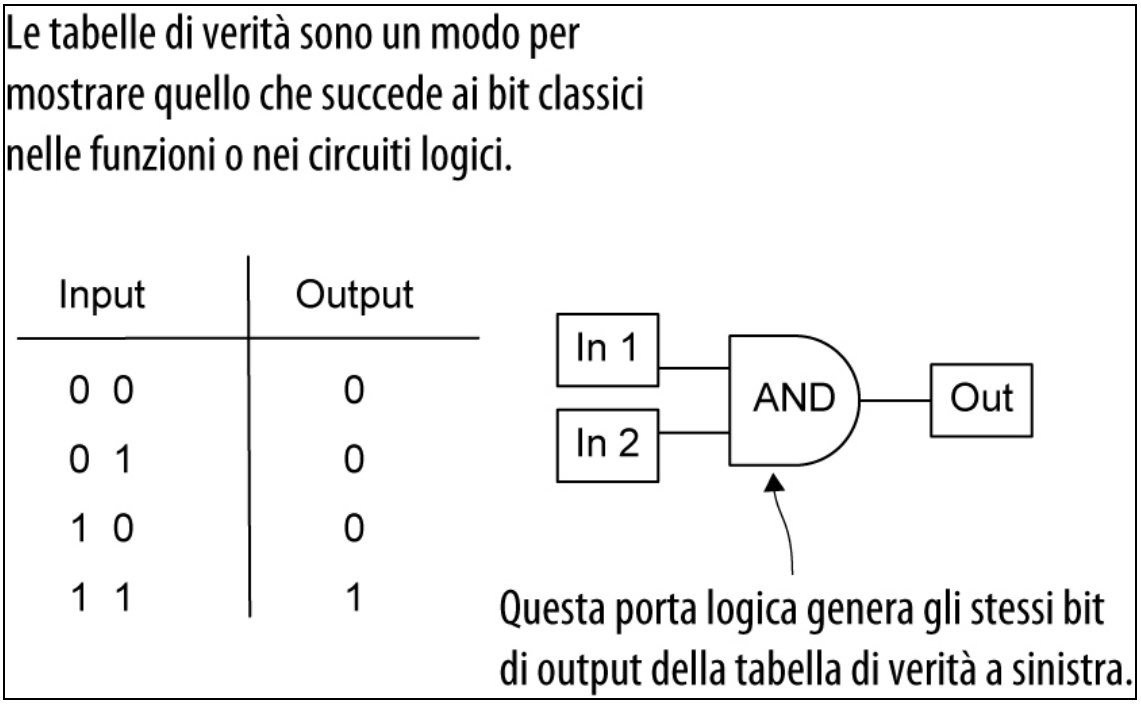


Figura 2.7 Tabella di verità per l'operazione logica AND. Se conosciamo l'intera tabella di verità per un'operazione logica, allora sappiamo che cosa fa quell'operazione per ogni possibile input.

NOTA

Il fatto di descrivere un'operazione come una tabella di verità funziona anche per operazioni più complicate. In linea di principio, anche un'operazione come la somma tra due interi a 64 bit può essere scritta come una tabella di verità.

Questo non sarebbe però molto pratico, poiché una tabella di verità per due input a 64 bit avrebbe $2^{128} \approx \times 10^{38}$ righe e per scriverla sarebbero necessari 10^{40} bit. Per valutarne la mole, stime recenti dicono che l'intera Internet occupi qualcosa come 10^{27} bit.

Gran parte dell'arte della progettazione della logica e dell'hardware classico consiste nel realizzare *circuiti* che possano fornire rappresentazioni molto compatte delle operazioni classiche, invece di fare affidamento su tabelle di verità potenzialmente enormi. Nel quantum computing, usiamo il nome *operatori unitari* per le tabelle di verità per i bit quantistici, concetto che espanderemo man mano che procederemo.

In sintesi:

- i bit classici sono sistemi fisici che possono trovarsi in due diversi *stati*;
- i bit classici possono essere manipolati tramite *operazioni*, per elaborare le informazioni;
- l'atto di *misurare* un bit classico crea una copia delle informazioni contenute nello stato.

NOTA

Nel prossimo paragrafo useremo l'algebra lineare per parlare dei *qubit*, l'unità di base delle informazioni in un computer quantistico. Se avete bisogno di ripassare l'algebra lineare, questo può essere un ottimo momento per compiere una deviazione verso l'Appendice C. Faremo riferimento a un'analogia presente in tale Appendice un po' in tutto il libro, dove considereremo i vettori come indicazioni su una mappa. Tranquilli, andate pure all'Appendice C… vi aspettiamo qui!

Qubit: stati e operazioni

Proprio come i bit classici sono l'unità di informazione più semplice di un computer classico, i qubit sono l'unità di base delle informazioni

di un computer quantistico. Possono essere implementati fisicamente da sistemi che hanno due stati, proprio come i bit classici, ma si comportano secondo le leggi della meccanica quantistica, il che aggiunge loro alcuni comportamenti di cui i bit classici non sono capaci. Trattiamo i qubit come ogni altro nuovo componente di un computer: colleghiamoli e vediamo che cosa succede!

Qubit simulati

In tutto il libro, non useremo veri qubit, ma le loro *simulazioni classiche*. Questo ci permetterà di scoprire come funzionano i computer quantistici e di iniziare a programmare piccole istanze dei tipi di problemi che i computer quantistici possono risolvere, pur non avendo ancora accesso all'hardware quantistico di cui avremmo bisogno per risolvere problemi pratici.

Il problema di questo approccio è che la simulazione dei qubit su computer classici richiede una quantità esponenziale di risorse classiche per ogni qubit. I più potenti servizi di calcolo classici possono simulare solo circa 40 qubit prima di dover semplificare o ridurre i tipi di programmi quantistici in esecuzione. Per fare un confronto, l'attuale hardware commerciale raggiunge al massimo circa 70 qubit reali, al momento della stesura di questo testo. I dispositivi con così tanti qubit sono estremamente difficili da simulare con i computer classici, ma i dispositivi attualmente disponibili sono ancora troppo rumorosi per completare le attività di calcolo più utili.

Immaginate di dover scrivere un programma classico con solo 40 bit classici con cui lavorare! Sebbene 40 bit siano piuttosto pochi rispetto ai gigabyte con cui siamo abituati a lavorare nella programmazione classica, possiamo comunque impiegarli per fare alcune cose davvero interessanti anche con soli 40 qubit, che ci aiuteranno a immaginare con un prototipo come potrebbe essere un effettivo vantaggio quantistico.

Lo stato del qubit

Per implementare il nostro generatore quantistico di numeri casuali (QRNG), dobbiamo capire come rappresentare il nostro qubit.

Abbiamo usato lucchetti, palle da baseball e altri sistemi classici per

rappresentare valori di bit *classici*, 0 o 1. Possiamo usare molti sistemi fisici per simulare il nostro qubit, e gli *stati* sono i "valori" che il nostro qubit può assumere.

Analogamente agli stati 0 e 1 dei bit classici, possiamo assegnare delle etichette agli stati quantistici. Gli stati del qubit più simili ai classici 0 e 1 sono $|0\rangle$ e $|1\rangle$, come illustrato nella Figura 2.8. Questi sono indicati, rispettivamente, come *ket 0* e *ket 1*.

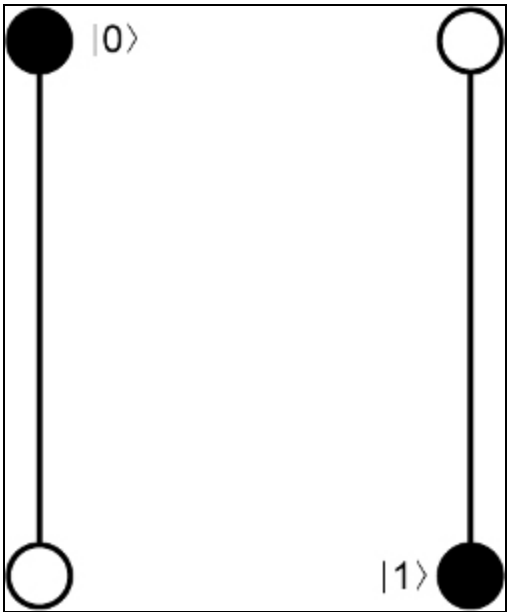


Figura 2.8 Se impieghiamo la notazione Dirac (bra-ket) per i qubit, possiamo rappresentare graficamente gli stati $|0\rangle$ e $|1\rangle$ dei qubit nello stesso modo in cui abbiamo rappresentato gli stati 0 e 1 dei bit classici. In particolare, rappresenteremo gli stati $|0\rangle$ e $|1\rangle$ come punti opposti lungo un asse.

Ket

Il termine *ket* viene da una sorta di denominazione stravagante nel quantum computing, la cui storia deriva da un gioco di parole particolarmente sciocco. Come vedremo quando parleremo di misurazioni, c'è un altro tipo di oggetto, chiamato *bra* e che si scrive $\langle 0|$. Quando mettiamo insieme un *bra* e un *ket*, otteniamo una coppia di parentesi, *bracket*: $\langle\rangle$.

L'uso di *bra* e *ket* per scrivere formule matematiche per la meccanica quantistica è chiamato *notazione di Dirac*, da Paul Dirac, che ha inventato sia la notazione sia il

gioco di parole di cui abbiamo parlato. Nel libro impiegheremo molto questo stile di rappresentazione.

Una cosa da tenere presente, tuttavia, è che uno stato è solo un modello: è comodo per prevedere come si comporta un qubit, ma non è una proprietà intrinseca del qubit. Questa distinzione diventerà particolarmente importante quando considereremo la misurazione, più avanti nel capitolo: come vedremo, non possiamo misurare direttamente lo stato di un qubit.

ATTENZIONE

Nei sistemi reali, non possiamo mai estrarre o conoscere perfettamente lo stato di un qubit dato un numero finito di copie.

Non preoccupatevi se tutto questo non ha ancora molto senso; vedremo molti esempi man mano che procederemo nel libro. Quello che è importante tenere a mente, per ora, è che i qubit non sono stati.

Se vogliamo simulare come si muove una palla da baseball una volta lanciata, potremmo iniziare scrivendo la sua posizione attuale, la sua velocità e la sua direzione, il modo in cui gira e così via. Questo elenco di valori ci aiuta a rappresentare una palla da baseball su un foglio di carta o in un computer, per prevedere che cosa farà quella palla, ma non possiamo certo dire che la palla da baseball *sia* quella lista di numeri. Per avviare la nostra simulazione, dovremmo prendere la palla da baseball che ci interessa e *misurare* dove si trova, a che velocità sta andando e così via.

Diciamo che l'intero set di dati di cui abbiamo bisogno per simulare accuratamente il comportamento di una palla da baseball è lo *stato* di quella palla. Allo stesso modo, lo stato di un qubit è la serie completa di dati che ci servono per simulare e predire i risultati che otteniamo dalla misurazione. Così come abbiamo bisogno di aggiornare lo stato di una palla da baseball mentre procede nel nostro simulatore, dovremo aggiornare lo stato di un qubit quando vi applichiamo delle operazioni.

SUGGERIMENTO

Un modo per ricordare questa sottile distinzione è dire che un qubit *è descritto da* uno stato, mentre non è vero che un qubit *sia uno* stato.

Le cose però si fanno un po' più complesse: mentre possiamo misurare una palla da baseball senza fare nulla su di essa, tranne copiare alcune informazioni classiche, come vedremo in tutto il resto del libro, non possiamo copiare perfettamente le informazioni quantistiche memorizzate in un qubit: quando misuriamo un qubit, ne alteriamo anche lo stato. Questo può creare confusione, in quanto nel simulatore possiamo registrare lo stato completo di un qubit, e possiamo controllare la memoria del nostro simulatore ogni volta che vogliamo. Niente di ciò che possiamo fare con i veri qubit ci permette di conoscerne lo stato, quindi se "imbrogliamo" controllando la memoria di un simulatore, non saremo in grado di eseguire il nostro programma su un vero hardware.

In altre parole, mentre controllare gli stati direttamente può essere utile per il debug dei simulatori classici che realizzeremo, dobbiamo assicurarci di scrivere solo algoritmi basati sulle informazioni che potremmo trarre da un hardware reale.

Barare a occhi chiusi

Quando utilizziamo un simulatore quantistico, esso deve memorizzare internamente lo stato dei nostri qubit: ecco perché la simulazione di sistemi quantistici può essere così difficile. Ogni qubit potrebbe, in linea di principio, essere correlato con ogni altro qubit, quindi abbiamo bisogno di impiegare risorse esponenziali, in generale, per scrivere lo stato nel nostro simulatore (ne riparleremo nel Capitolo 4).

Se "imbrogliamo" osservando direttamente lo stato memorizzato nel simulatore, potremo eseguire il nostro programma solo su un simulatore, non su un hardware reale. Nei prossimi capitoli vedremo come imbrogliare in modo più sicuro, utilizzando asserzioni e rendendo inosservabili gli imbrogli 😈.

Il gioco delle operazioni

Ora che abbiamo i nomi di questi stati, vediamo come rappresentare le informazioni che essi contengono. Con i bit classici, possiamo registrare con facilità le informazioni contenute nel bit come, semplicemente, un valore su una linea: 0 o 1. Funziona perché le uniche operazioni che possiamo fare sono l'inversione (o rotazione di 180°) su questa linea. La meccanica quantistica ci consente di applicare più tipi di operazioni ai qubit, comprese rotazioni di meno di 180°. Pertanto, i qubit differiscono dai bit classici anche in termini di operazioni che possiamo eseguire su di essi.

> **NOTA**
> Mentre le operazioni sui bit classici sono operazioni logiche che possono essere eseguite combinando in vari modi gli operatori NOT, AND e OR, le operazioni quantistiche consistono in *rotazioni*.

Per esempio, se vogliamo trasformare lo stato di un qubit da $|0\rangle$ a $|1\rangle$ e viceversa, ovvero l'analogo quantistico di un'operazione NOT, dobbiamo ruotare il qubit in senso orario di 180°, come illustrato nella Figura 2.9.

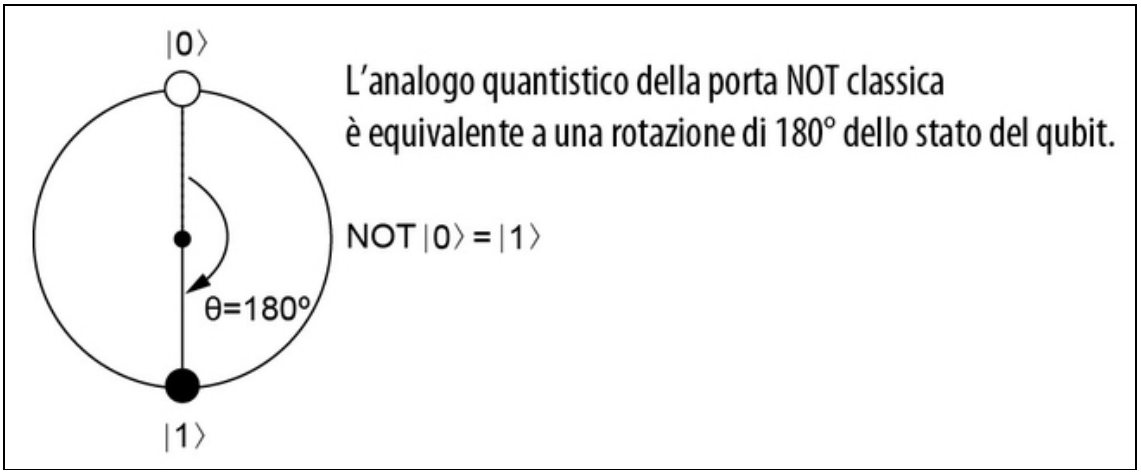


Figura 2.9 Una rappresentazione dell'equivalente quantistico di un'operazione NOT su un qubit nello stato $|0\rangle$, che lascia il qubit nello stato $|1\rangle$. Possiamo immaginare

questa operazione come una rotazione di 180° intorno al centro della linea che collega gli stati |0⟩ e |1⟩.

Abbiamo visto come la rotazione di 180° sia l'analogo di una porta NOT, ma quali altre rotazioni possiamo fare?

Reversibilità

Quando ruotiamo uno stato quantistico, possiamo sempre tornare allo stesso stato di partenza con una rotazione all'indietro. Questa proprietà, nota come reversibilità, risulta essere fondamentale per il quantum computing. Con l'eccezione della misurazione, della quale parleremo più avanti in questo capitolo, tutte le operazioni quantistiche devono essere reversibili.

Tuttavia, non tutte le operazioni classiche cui siamo abituati sono reversibili. Operazioni come AND e OR non sono reversibili, così come sono tipicamente scritte, quindi non possono essere implementate come operazioni quantistiche senza un po' di lavoro. Vedremo come farlo nel Capitolo 8, quando introdurremo il trucco "uncompute" per esprimere altre operazioni classiche in termini di rotazioni.

D'altra parte, operazioni classiche come XOR possono essere facilmente rese reversibili, quindi possiamo scriverle come rotazioni impiegando un'operazione quantistica chiamata *NOT controllato*, come vedremo nel Capitolo 8.

Se ruotiamo un qubit nello stato |0⟩ di 90° in senso orario invece di 180°, otteniamo un'operazione quantistica che possiamo immaginare come la radice quadrata di un'operazione NOT, come illustrato nella Figura 2.10.

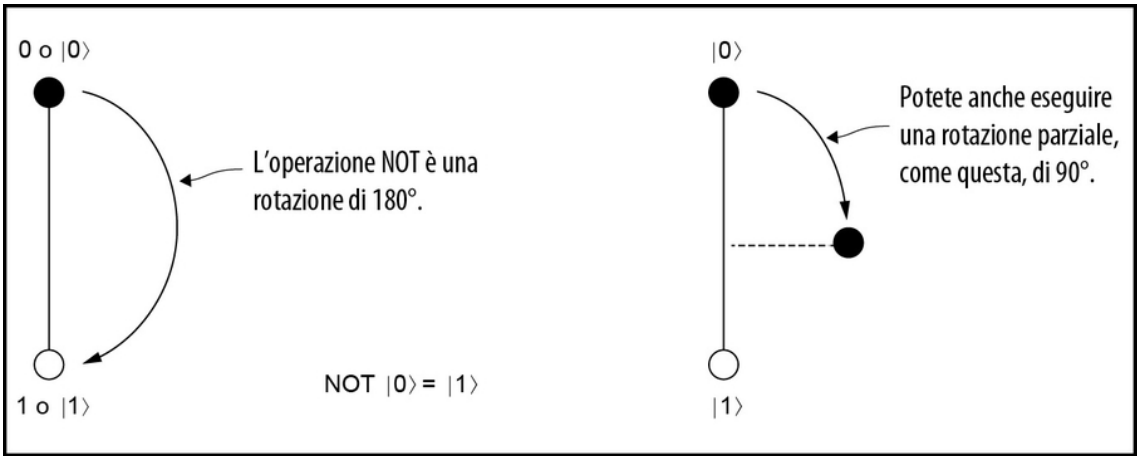


Figura 2.10 Possiamo anche ruotare uno stato di meno di 180 gradi. In questo modo, otteniamo uno stato che non è né |0⟩ né |1⟩, ma che si trova a metà del cerchio tra di essi.

Proprio come abbiamo definito in precedenza la radice quadrata $\sqrt{x}$ di un numero x come il numero y tale che $y^2 = x$, possiamo definire la radice quadrata anche di un'operazione quantistica. Se applichiamo due volte una rotazione di 90°, otteniamo l'operazione NOT, quindi possiamo considerare la rotazione di 90° come la radice quadrata del NOT.

Fattori di 2

Ogni campo ha le sue peculiarità. Chiedete a un programmatore grafico se un y positivo significa "su" o "giù", per esempio. Nel quantum computing, la ricca storia e la natura interdisciplinare del campo a volte si rivelano un'arma a doppio taglio, in quanto ogni diverso modo di immaginare il quantum computing richiede l'impiego di convenzioni e notazioni.

Un modo in cui si manifesta questo problema è che è davvero facile commettere errori su dove mettere i fattori di 2. In questo libro, abbiamo scelto di seguire la convenzione usata dal linguaggio Q# di Microsoft.

Ora abbiamo un nuovo stato che non è né |0⟩ né |1⟩, ma una combinazione "paritaria" di entrambi. Così come possiamo descrivere

il “nordest” sommando le direzioni “nord” ed “est”, possiamo scrivere questo nuovo stato come illustrato nella Figura 2.11.

Lo stato di un qubit può essere rappresentato come un punto su un cerchio che ha due stati etichettati sui poli: $|0\rangle$ e $|1\rangle$. Più in generale, rappresenteremo le rotazioni utilizzando angoli arbitrari q tra gli stati dei qubit, come illustrato nella Figura 2.12.

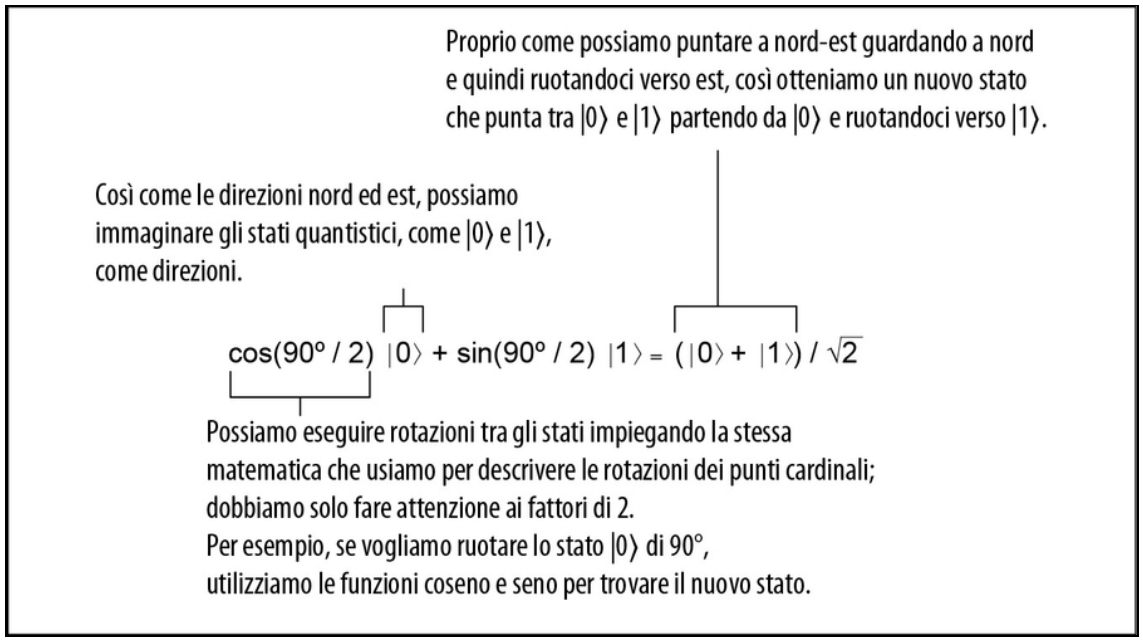


Figura 2.11 Possiamo scrivere lo stato che otteniamo quando eseguiamo una rotazione di 90° pensando agli stati $|0\rangle$ e $|1\rangle$ come a direzioni. In questo modo, e con un po’ di trigonometria, otteniamo che la rotazione di 90° dello stato $|0\rangle$ ci dà un nuovo stato $(|0\rangle + |1\rangle) / \sqrt{2}$. Per maggiori dettagli su come scrivere la matematica per questo tipo di rotazioni, rimandiamo all’Appendice B, che contiene un ripasso di algebra lineare.

$|+\rangle$, $|-\rangle$ e sovrapposizione

Chiamiamo quello stato che è una combinazione uguale di $|0\rangle$ e $|1\rangle$ con il nome $|+\rangle = (|0\rangle + |1\rangle) / \sqrt{2}$ (a causa del segno tra i termini). Diciamo che lo stato $|+\rangle$ è una *sovrapposizione* di $|0\rangle$ e $|1\rangle$.

Se la rotazione è di -90° (in senso antiorario), allora chiamiamo lo stato risultante $|-\rangle = (|0\rangle - |1\rangle) / \sqrt{2}$. Provate a scrivere queste rotazioni impiegando -90° e vedrete che otterrete $|-\rangle$.

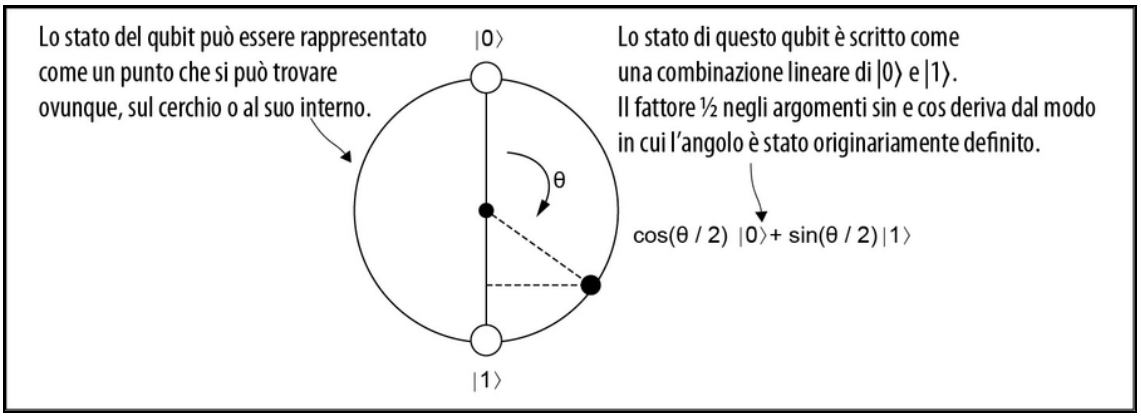


Figura 2.12 Se ruotiamo lo stato |0⟩ di un angolo diverso da 90° o 180°, lo stato risultante può essere rappresentato come un punto su un cerchio che ha |0⟩ e |1⟩ come poli superiore e inferiore. Questo ci dà un modo per visualizzare i possibili stati in cui può trovarsi un singolo qubit.

Matematicamente, possiamo scrivere lo stato di qualsiasi punto del cerchio che rappresenta il nostro qubit come cos(q / 2) |0⟩ + sin(q / 2) |1⟩, dove |0⟩ e |1⟩ sono modi diversi di scrivere, rispettivamente, i vettori [[1], [0]] e [[0], [1]].

SUGGERIMENTO

Un modo per pensare alla notazione ket è dire che dà un nome ai vettori che usiamo comunemente. Quando scriviamo |0⟩ = [[1], [0]], stiamo dicendo che [[1], [0]] è abbastanza importante da denominarlo con lo 0. Allo stesso modo, quando scriviamo |+⟩ = [[1], [1]] / √2, diamo un nome alla rappresentazione vettoriale di uno stato che impiegheremo in tutto il libro.

Un altro modo per dirlo è che un qubit è generalmente la *combinazione lineare* di vettori di |0⟩ e |1⟩ con coefficienti che descrivono l'angolo di cui ruotare |0⟩ per arrivare a tale stato. Per essere utili per la programmazione, possiamo scrivere il modo in cui la rotazione di uno stato influenza ciascuno degli stati |0⟩ e |1⟩, come illustrato nella Figura 2.13.

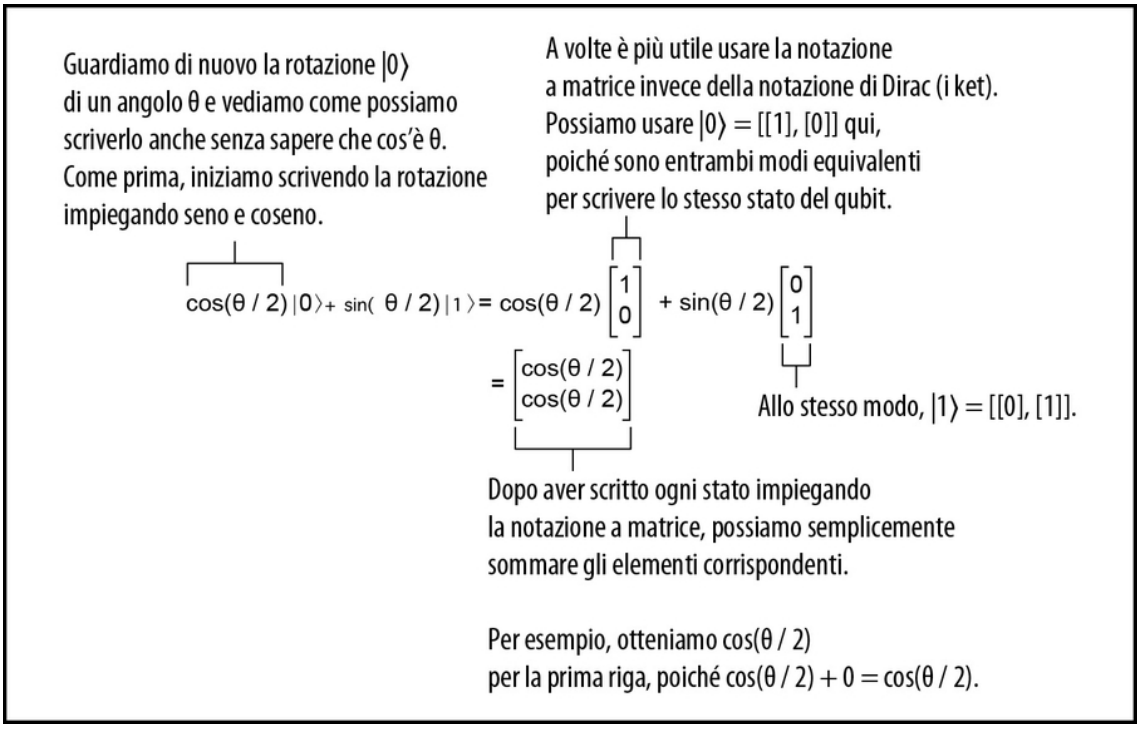


Figura 2.13 Impiegando l'algebra lineare, possiamo descrivere lo stato di un singolo qubit come un vettore di due elementi. In questa equazione, vediamo come quel modo di immaginare gli stati del qubit si riferisce al nostro precedente uso della notazione di Dirac (bra-ket). In particolare, mostriamo lo stato finale dopo aver ruotato lo stato |0⟩ di un angolo arbitrario q utilizzando sia la notazione vettoriale sia la notazione Dirac; entrambe le notazioni saranno utili in diversi punti del nostro viaggio quantistico.

SUGGERIMENTO

Questa è esattamente la stessa cosa che abbiamo usato in precedenza: una base di vettori per rappresentare una funzione lineare come una matrice.

Più avanti impareremo a conoscere altre operazioni quantistiche, che è più facile rappresentare come rotazioni. La Tabella 2.2 riepiloga gli stati che abbiamo imparato a creare da queste rotazioni.

Tabella 2.2 Etichette di stato, espansioni nella notazione di Dirac e rappresentazione vettoriale.

Etichetta di stato	Notazione di Dirac	Rappresentazione vettoriale
|0⟩	|0⟩	[[1], [0]]

$\|1\rangle$	$\|1\rangle$	[[0], [1]]
$\|+\rangle$	$(\|0\rangle + \|1\rangle) / \sqrt{2}$	[[1 / √2], [1 / √2]]
$\|-\rangle$	$(\|0\rangle - \|1\rangle) / \sqrt{2}$	[[1 / √2], [–1 / √2]]

Un boccone di matematica

A prima vista, qualcosa come $|+\rangle = (|0\rangle + |1\rangle) / \sqrt{2}$ sarebbe terribile da pronunciare, impossibile da usare nella conversazione. In pratica, tuttavia, i programmatori quantistici spesso prendono alcune scorciatoie quando parlano o disegnano queste cose su una lavagna.

Per esempio, la parte "√2" deve sempre essere presente, poiché i vettori che rappresentano gli stati quantistici devono essere sempre di lunghezza 1; ciò significa che possiamo anche darla per scontata e scrivere "$|+\rangle = |0\rangle + |1\rangle$", contando sul fatto che chi ci ascolta sa benissimo che dovrà poi dividere per √2. Se stiamo facendo un discorso o stiamo discutendo di quantum computing davanti a una tazza di caffè, possiamo dire, "Ket più è ket 0 più ket 1", ma la ripetizione della parola "più" genera un po' di confusione, senza i bra e i ket. Per enfatizzare verbalmente che la somma ci permette di rappresentare la sovrapposizione, potremmo invece dire: "Lo stato più è una sovrapposizione uguale di 0 e 1".

Misurazione dei qubit

Per recuperare le informazioni memorizzate in un qubit, dobbiamo misurarlo. Idealmente, vorremmo disporre di un dispositivo di misurazione che ci consentisse di leggere direttamente le informazioni sullo stato tutte in una volta. Tuttavia, questo non è possibile in base alle leggi della meccanica quantistica, come vedremo nei Capitoli 3 e 4. Detto questo, la misurazione *può* consentirci di conoscere le informazioni sullo stato relative a specifiche direzioni del sistema. Per esempio, se abbiamo un qubit nello stato $|0\rangle$ e cerchiamo di vedere se è nello stato $|0\rangle$, otterremo sempre che è in quello stato. D'altra parte, se abbiamo un qubit nello stato $|+\rangle$ e vogliamo vedere se è nello stato $|0\rangle$, otterremo un risultato 0 con il 50% di probabilità. Come illustrato nella

Figura 2.14, questo accade perché lo stato $|+\rangle$ si sovrappone allo stesso modo agli stati $|0\rangle$ e $|1\rangle$, in modo tale che avremo entrambi i risultati con la stessa probabilità.

SUGGERIMENTO

I risultati della misurazione dei qubit sono *sempre* valori di bit classici! In altre parole, indipendentemente dal fatto che misuriamo un bit classico o un qubit, il nostro risultato sarà sempre un bit classico.

La maggior parte delle volte sceglieremo di misurare se abbiamo un $|0\rangle$ o un $|1\rangle$, ovvero vorremo misurare lungo la linea tra $|0\rangle$ e $|1\rangle$. Per comodità, diamo un nome a questo asse: l'asse *Z*. Possiamo rappresentarlo visivamente proiettando il nostro vettore di stato sull'asse *Z* (vedi Figura 2.15) impiegando il prodotto interno.

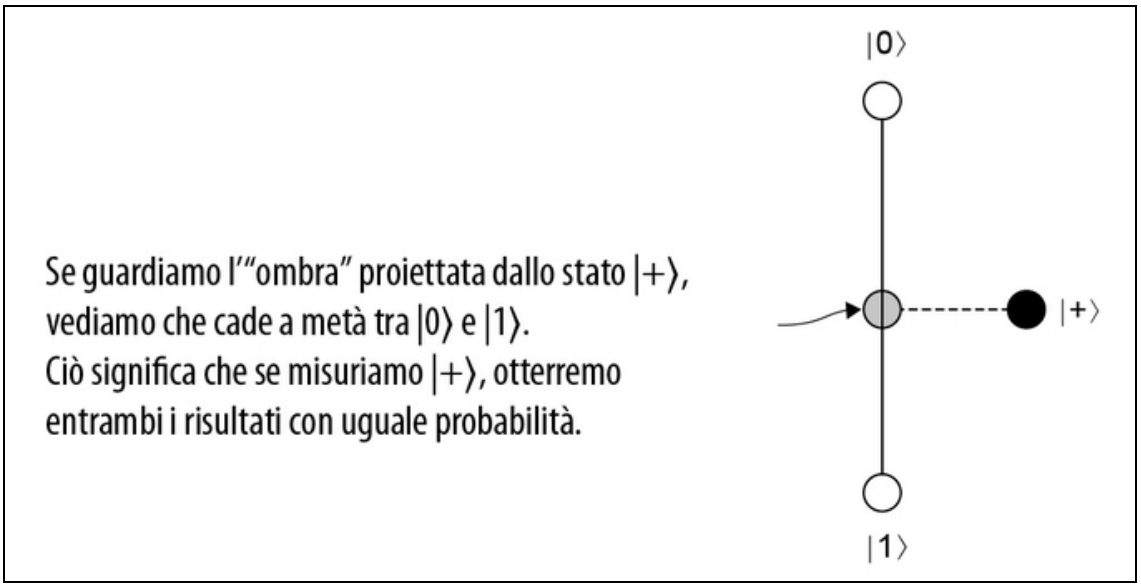


Figura 2.14 Lo stato $|+\rangle$ si sovrappone ugualmente sia a $|0\rangle$ sia a $|1\rangle$, poiché "l'ombra" che proietta cade esattamente nel mezzo. Quindi, quando guardiamo il nostro qubit per vedere se è nello stato $|0\rangle$ o $|1\rangle$, otterremo entrambi i risultati con la stessa probabilità, se il nostro qubit è iniziato nello stato $|+\rangle$. Possiamo immaginare l'ombra proiettata dallo stato $|+\rangle$ sulla linea tra gli stati $|0\rangle$ e $|1\rangle$ come a una specie di lancio di una moneta.

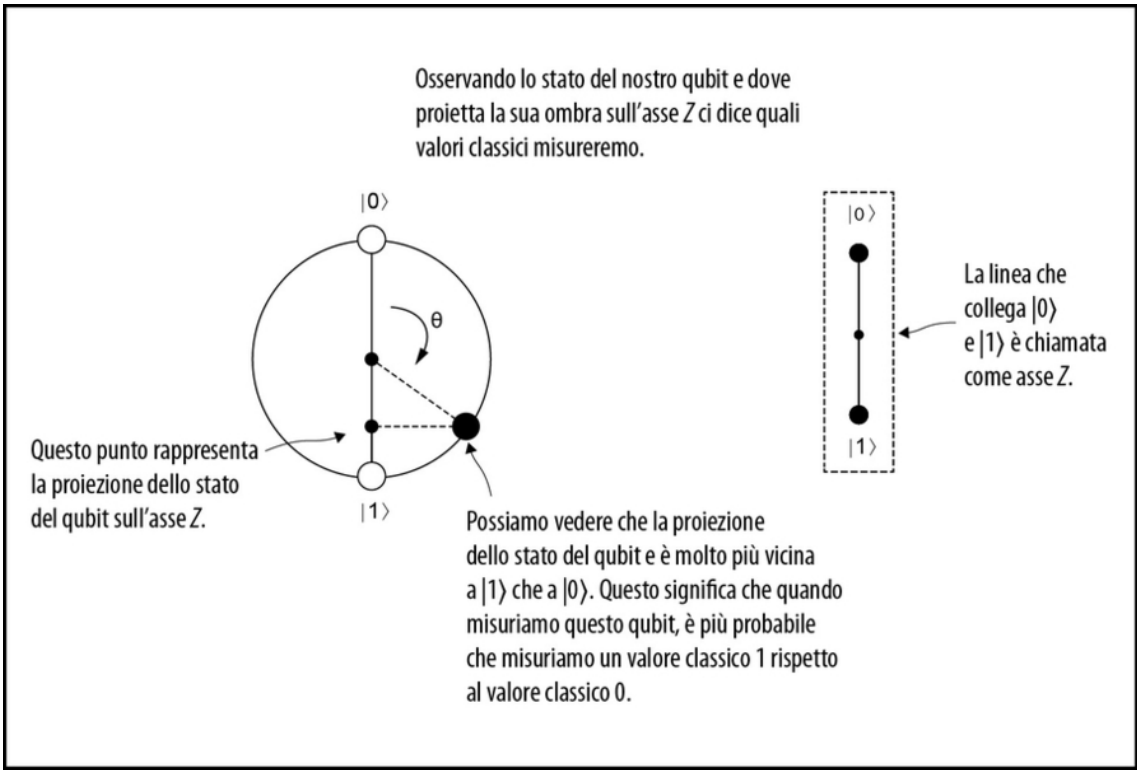


Figura 2.15 Rappresentazione di una misurazione quantistica, che può essere immaginata come una proiezione dello stato in una particolare direzione. Per esempio, se un qubit viene ruotato in modo tale che il suo stato sia vicino allo stato $|1\rangle$, è più probabile che la misurazione restituisca 1 rispetto a 0.

SUGGERIMENTO

Per un ripasso sul prodotto interno, consultate l'Appendice B.

Pensate di illuminare con una torcia elettrica lo stato di un qubit proiettandolo sull'asse *Z*; la probabilità di ottenere un risultato 0 o 1 è determinata dall'ombra proiettata dallo stato sull'asse *Z*.

Approfondimento: perché la misurazione non è lineare?

Dopo aver tanto parlato della linearità della meccanica quantistica, può sembrare strano che ora parliamo di una misurazione non lineare. Se possiamo eseguire operazioni non lineari come la misurazione, possiamo implementare anche altre operazioni non lineari come la clonazione dei qubit?

La "versione breve" è che mentre tutti concordano sulla matematica che sta alla base della misurazione, esistono ancora molte discussioni filosofiche sul modo

migliore per capire perché la misurazione quantistica agisce in questo modo. Queste discussioni cadono sotto il nome di *fondamenti quantistici* e tentano di fare di più che capire semplicemente che cos'è la meccanica quantistica e che cosa prevede, per comprenderne anche il motivo. Per la maggior parte, tali fondamenti esplorano diversi modi di *interpretare* la meccanica quantistica. Allo stesso modo in cui possiamo comprendere la probabilità classica considerando determinati esperimenti mentali controintuitivi come le strategie degli show a premi o il modo in cui i casinò possano guadagnare anche da giochi nei quali sembrano perdere denaro, i fondamenti quantistici sviluppano nuove interpretazioni attraverso piccoli esperimenti mentali che sondano diversi aspetti della meccanica quantistica. Fortunatamente, alcuni dei risultati dei fondamenti quantistici possono aiutarci a dare un senso alla misurazione.

In particolare, un'osservazione fondamentale è che possiamo sempre rendere nuovamente lineari le misurazioni quantistiche includendo nella descrizione lo stato dell'apparato di misurazione; vedremo alcuni degli strumenti matematici necessari per farlo nei Capitoli 4 e 6. Se spinta all'estremo, questa osservazione porta a interpretazioni come quella dei *molti mondi*. L'interpretazione dei molti mondi risolve l'interpretazione della misurazione insistendo sul fatto che consideriamo solo gli stati che includono i dispositivi di misurazione, in modo tale che l'apparente non linearità della misurazione in realtà sparisca.

All'estremo opposto possiamo interpretare la misurazione osservando che la non linearità della misurazione quantistica è esattamente la stessa di un ramo della statistica noto come inferenza bayesiana. Pertanto, la meccanica quantistica appare non lineare solo quando dimentichiamo di includere il fatto che un agente sta eseguendo la misurazione e sta imparando da ogni risultato. Questa osservazione porta a pensare alla meccanica quantistica come a una descrizione non tanto del mondo ma di ciò che sappiamo del mondo.

Sebbene questi due tipi di interpretazioni siano in disaccordo a livello filosofico, entrambi offrono modi diversi per risolvere il modo in cui una teoria lineare come la meccanica quantistica può talvolta sembrare non lineare. Indipendentemente dall'interpretazione impiegata per aiutarci a comprendere l'interazione tra la misurazione e il resto della meccanica quantistica, siamo sicuri che i risultati della misurazione sono sempre descritti dalla stessa matematica e dalle stesse simulazioni. In effetti, fare affidamento sulle simulazioni (a volte chiamate, sarcasticamente, interpretazioni "zitto e calcola") è la più antica e celebrata di tutte le interpretazioni.

La lunghezza al quadrato di ciascuna proiezione *rappresenta* la probabilità che lo stato che stiamo misurando si trovi lungo quella direzione. Se abbiamo un qubit nello stato $|0\rangle$ e proviamo a misurarlo lungo la direzione dello stato $|1\rangle$, otterremo una probabilità pari a 0, perché gli stati sono opposti l'uno all'altro, quando li disegniamo su un cerchio. Ragionando in termini di immagini, lo stato $|0\rangle$ non ha alcuna proiezione sullo stato $|1\rangle$: osservando la Figura 2.15, $|0\rangle$ non proietta nessuna ombra su $|1\rangle$.

SUGGERIMENTO

Se qualcosa accade con probabilità 1, quell'evento si verifica sempre. Se qualcosa accade con probabilità 0, allora quell'evento è impossibile. Per esempio, la probabilità che un dado a sei facce dia un 7 è 0, poiché quel lancio è impossibile. Allo stesso modo, se un qubit è nello stato $|0\rangle$, ottenere un risultato 1 da una misurazione dell'asse Z è impossibile, poiché $|0\rangle$ non ha alcuna proiezione su $|1\rangle$.

Se abbiamo un $|0\rangle$ e proviamo a misurarlo lungo la direzione $|0\rangle$, otterremo una probabilità 1 perché gli stati sono paralleli (e di lunghezza 1, per definizione). Esaminiamo ora la misurazione di uno stato che non è né parallelo né perpendicolare.

Esempio

Supponiamo di avere un qubit nello stato $(|0\rangle + |1\rangle) / \sqrt{2}$ (la stessa cosa di $|+\rangle$, dalla Tabella 2.1), e di volerlo misurare o proiettare lungo l'asse Z. Possiamo quindi trovare la probabilità che il risultato classico sia un 1 proiettando $|+\rangle$ su $|1\rangle$.

Possiamo trovare la proiezione di uno stato su un altro utilizzando il *prodotto interno* delle loro rappresentazioni vettoriali. In questo caso, scriviamo il prodotto interno di $|+\rangle$ e $|1\rangle$ come $\langle 1|+\rangle$, dove $\langle 1|$ è la trasposizione di $|1\rangle$ e dove la sovrapposizione tra le due barre contrapposte indica il calcolo del prodotto interno. Più avanti vedremo che $\langle 1|$ è la trasposta coniugata di $|1\rangle$ ed è chiamata "bra".

Possiamo quindi scrivere così.

Per calcolare la proiezione, iniziamo scrivendo il "bra" su cui vogliamo eseguire la proiezione.

$\langle 1| (|0\rangle + |1\rangle) / \sqrt{2}$

Poi distribuiamo il bra.

= (1 / √2)(⟨1|(|0⟩ + ⟨1|1⟩)
Possiamo scrivere ogni coppia bra-ket come un prodotto interno fra i due valori.
= (1 / √2) ([[0], [1]] · [[1], [0]] + [[0],[1]] · [[0], [1]])
Il calcolo dei prodotti interni semplifica molto le cose.
=(1 / √2)(0 + 1)
Ora abbiamo la sovrapposizione fra |0⟩ e |1⟩.
=(1 / √2)
Per trasformare questa proiezione in una probabilità, la eleviamo al quadrato, ottenendo che la probabilità di osservare un risultato 1 quando prepariamo uno stato |+⟩ è di 1/2.

Spesso eseguiamo la proiezione sull'asse *Z*, perché è comodo farlo in molti esperimenti, ma possiamo effettuarla anche lungo l'asse *X* per capire se abbiamo uno stato |+⟩ o |–⟩. Misurando lungo l'asse *X*, otteniamo |+⟩ con certezza e non otteniamo mai |–⟩, come illustrato nella Figura 2.16.

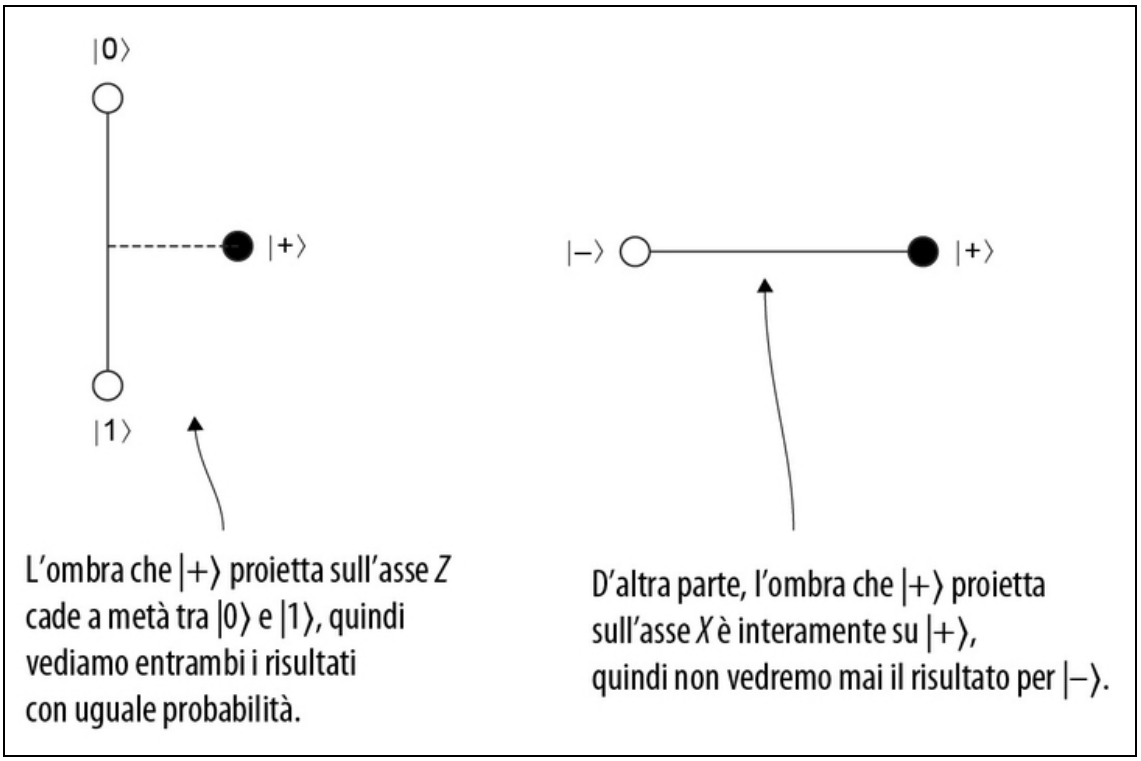


Figura 2.16 Misurando |+⟩ lungo l'asse X otteniamo sempre |+⟩. Per vederlo, notate che l'"ombra" proiettata dallo stato |+⟩ sull'asse X (cioè, la linea tra gli stati |–⟩ e |+⟩)

è precisamente lo stato |+⟩ stesso.

NOTA

Possiamo ottenere dalla misurazione un risultato completamente certo *solo* perché in questo caso conosciamo in anticipo la direzione "giusta" della misurazione. Se ottenessimo semplicemente uno stato, senza alcuna informazione su come sia "giusto" misurarlo, non potremmo prevedere perfettamente alcun risultato della misurazione.

Misurazione generalizzata: indipendenza di base

A volte potremmo non sapere come è stato preparato il nostro qubit, quindi non sapere neanche come misurarlo correttamente. Più in generale, qualsiasi coppia di stati che non si sovrappongono (ovvero che sono poli opposti) definisce un sistema di misurazione. Il risultato effettivo di una misurazione è un valore classico di bit che indica a quale polo è allineato lo stato nel momento in cui eseguiamo la misurazione.

Misure ancora più generali

La meccanica quantistica consente di eseguire tipi di misurazioni molto più generali: ne vedremo alcune man mano che procederemo, ma in questo libro ci concentriamo principalmente sul caso del controllo tra due poli opposti. Questa scelta è un modo abbastanza comodo per controllare la maggior parte dei dispositivi quantistici e può essere utilizzata in quasi tutte le piattaforme commerciali per il quantum computing attualmente disponibili.

Matematicamente, usiamo notazioni come ⟨misurazione | stato⟩ per rappresentare la misurazione di un qubit. Il componente a sinistra, ⟨misurazione |, è chiamato *bra* e, come abbiamo già visto, il componente a destra è chiamato *ket*. Insieme, formano *braket*.

I bra sono molto simili ai ket, tranne per il fatto che per passare dall'uno all'altro dobbiamo prendere la trasposta (trasformare le righe in colonne e viceversa) del bra o del ket che abbiamo:

$$|0\rangle^T = [[1],[0]]^T = [[1,0]].$$

Un altro modo per considerare questo fatto è che la trasposizione trasformi i vettori colonna (i ket) in vettori riga (i bra).

NOTA

Dal momento che per ora stiamo lavorando solo con numeri reali, non avremo bisogno di fare nient'altro per passare dai ket ai bra. Ma quando nel prossimo capitolo lavoreremo con numeri complessi, avremo bisogno anche del coniugato complesso.

I bra ci permettono di scrivere le misurazioni. Ma per vedere che cosa fanno effettivamente le misurazioni, abbiamo bisogno di un'altra cosa: una regola su come usare un bra e un ket insieme per ottenere la probabilità di vedere il risultato della misurazione. In meccanica quantistica, le probabilità della misurazione si trovano osservando la lunghezza della proiezione o dell'ombra che il ket di uno stato lascia su un bra nella misurazione. Sappiamo per esperienza che possiamo trovare le proiezioni e le lunghezze utilizzando i prodotti interni. Nella notazione di Dirac, il prodotto interno di un bra e di un ket è scritto come ⟨misurazione | stato⟩, cosa che ci offre la regola di cui abbiamo bisogno.

Per esempio, se abbiamo preparato uno stato $|+\rangle$ e vogliamo conoscere la probabilità di osservare un 1 quando lo misuriamo in base *Z*, quindi proiettandolo come illustrato nella Figura 2.15, possiamo trovare la lunghezza di cui abbiamo bisogno. La proiezione di $|+\rangle$ su $\langle 1|$ ci dice che vediamo un risultato 1 con probabilità $\Pr(1|+) = |\langle 1|+\rangle|^2 = |\langle 1|0\rangle + \langle 1|1\rangle|^2 / 2 = |0 + 1|^2 / 2 = 1 / 2$. Pertanto, il 50% delle volte, otterremo un risultato 1 e l'altro 50% delle volte, otterremo un risultato 0.

La regola di Born

Se abbiamo uno stato quantistico |stato⟩ ed eseguiamo una misurazione lungo la direzione ⟨misurazione|, possiamo scrivere la probabilità che osserveremo il

risultato *misurazione* come:

$\Pr(\text{misurazione}|\text{stato}) = |\langle \text{misurazione}|\text{stato}\rangle|^2$

In altre parole, la probabilità è il quadrato della grandezza del prodotto interno della misurazione bra e dello stato ket.

Questa espressione è chiamata *regola di Born*.

Nella Tabella 2.3, abbiamo elencato molti altri esempi di utilizzo della regola di Born per prevedere quali bit classici otterremo misurando i qubit.

Tabella 2.3 Esempi di utilizzo della regola di Born per trovare le probabilità della misurazione.

Se prepariamo…	**… e misuriamo…**	**… allora vediamo questo risultato con questa probabilità.**
$\|0\rangle$	$\langle 0\|$	$\|\langle 0\|0\rangle\|^2 = 1$
$\|0\rangle$	$\langle 1\|$	$\|\langle 1\|0\rangle\|^2 = 0$
$\|0\rangle$	$\langle +\|$	$\|\langle +\|0\rangle\|^2 = \|(\langle 0\| + \langle 1\|)\ \|0\rangle / \sqrt{2}\|^2 = (1 / \sqrt{2} + 0)^2 = 1 / 2$
$\|+\rangle$	$\langle +\|$	$\|\langle +\|+\rangle\|^2 = \|(\langle 0\| + \langle 1\|)\ (\|0\rangle + \|1\rangle) / 2\|^2 = \|\langle 0\|0\rangle + \langle 1\|0\rangle + \langle 0\|1\rangle + \langle 1\|1\rangle\|^2 / 4 = \|1 + 0 + 0 + 1\|^2 / 4 = 2^2 / 4 = 1$
$\|+\rangle$	$\langle -\|$	$\|\langle -\|+\rangle\|^2 = 0$
$-\|0\rangle$	$\langle 0\|$	$\|-\langle 0\|0\rangle\|^2 = \|-1\|^2 = 1^2$
$-\|+\rangle$	$\langle -\|$	$\|-\langle +\|-\rangle\|^2 = \|(-\langle 0\| - \langle 1\|)\ (\|0\rangle - \|1\rangle) / 2\|^2 = \|-\langle 0\|0\rangle - \langle 1\|0\rangle + \langle 0\|1\rangle + \langle 1\|1\rangle\|^2 / 4 = \|-1 - 0 + 0 + 1\|^2 / 4 = 0^2 / 4 = 0$

SUGGERIMENTO

Nella Tabella 2.3, usiamo il fatto che $\langle 0|0\rangle = \langle 1|1\rangle = 1$ e che $\langle 0|1\rangle = \langle 1|0\rangle = 0$ (provate a verificarlo). Quando due stati hanno prodotto interno 0, diciamo che

sono *ortogonali* (o *perpendicolari*). Il fatto che $|0\rangle$ e $|1\rangle$ siano ortogonali semplifica e accelera molti calcoli.

Ora abbiamo trattato tutto ciò che occorre sapere sui qubit per poterli simulare! Esaminiamo i requisiti che dobbiamo soddisfare per assicurarci di avere qubit funzionanti.

I qubit
Un qubit è qualsiasi sistema fisico che soddisfi tre proprietà:

- il sistema può essere perfettamente simulato data la conoscenza del vettore dei numeri (lo stato);
- il sistema può essere trasformato impiegando operazioni quantistiche (per esempio, rotazioni);
- qualsiasi misurazione del sistema produce un singolo bit classico, in base alla regola di Born.

Ogni volta che abbiamo un qubit (un sistema con le tre proprietà precedenti), possiamo descriverlo impiegando lo stesso codice matematico o di simulazione, senza ulteriore riferimento al tipo di sistema con cui stiamo lavorando. Questo è come dire che non abbiamo bisogno di sapere se un bit è definito dalla direzione del movimento di un flipper o dalla tensione in un transistor per scrivere delle porte NOT o AND o per scrivere software che utilizza tali porte per fare operazioni interessanti.

NOTA
Analogamente a come usiamo la parola *bit* per indicare sia un sistema fisico che memorizza le informazioni sia le informazioni memorizzate nel bit, usiamo anche la parola *qubit* per indicare sia un dispositivo quantistico sia le informazioni quantistiche in esso memorizzate.

Fase
Nelle ultime due righe della Tabella 2.3, abbiamo visto che la moltiplicazione di uno stato per una fase–1 non influisce sulle probabilità della misurazione. Questa non è una coincidenza, ma indica piuttosto una delle cose più interessanti dei qubit.

Poiché la regola di Born considera solo il valore assoluto al quadrato del prodotto interno di uno stato e una misurazione, la moltiplicazione di un numero per (–1) non influisce sul suo valore assoluto. I numeri come +1 o –1, il cui valore assoluto è uguale a 1, sono chiamati *fasi*. Nel prossimo capitolo, parleremo molto di più delle fasi, dovendo lavorare con i numeri complessi.

Per ora diciamo che moltiplicare un intero vettore per –1 è un esempio di applicazione di una *fase globale*, mentre cambiare da $|+\rangle$ a $|-\rangle$ è un esempio di applicazione di una *fase relativa* tra $|0\rangle$ e $|1\rangle$. Mentre le fasi globali non sempre influenzano i risultati delle misurazioni, c'è una grande differenza tra gli stati $|+\rangle = (|0\rangle + |1\rangle) / \sqrt{2}$ e $|-\rangle = (|0\rangle - |1\rangle) / \sqrt{2}$: i coefficienti davanti a $|0\rangle$ e $|1\rangle$ sono uguali a $|+\rangle$ e sono differenti di una fase di (–1) in $|-\rangle$. Parleremo della differenza tra questi due concetti nei Capitoli 3, 4, 6 e 7.

Simulazione di qubit nel codice

Ora che abbiamo scoperto come descrivere gli stati, le operazioni e le misurazioni dei qubit, è tempo di vedere come rappresentare tutti questi concetti nel codice. Useremo uno scenario con la nostra amica Eve per immaginare il codice che dovremo scrivere.

Supponiamo di voler mantenere segreto il nostro 💔 per Eve, perché nessun altro possa scoprirlo. Come possiamo codificare il nostro messaggio per Eve in modo che solo lei possa leggerlo?

Esploreremo meglio questa applicazione nel prossimo capitolo, ma l'elemento fondamentale di cui abbiamo bisogno per creare un buon algoritmo di *crittografia* è una sorgente di numeri casuali che sia difficile da prevedere. Scriviamo esattamente come vorremmo unire il nostro segreto e dei bit casuali per rendere sicuro il messaggio da inviare a Eve. La Figura 2.17 mostra che se sia noi sia Eve conosciamo la stessa sequenza segreta di bit classici casuali, possiamo usare quella sequenza per comunicare in modo sicuro. All'inizio del capitolo, abbiamo visto come scrivere il messaggio, o *testo in chiaro*, che vogliamo inviare a Eve (in questo caso, "💔") sotto forma di una stringa di bit classici. Il pad monouso è una sequenza di bit classici casuali che

si comporta come un modo per codificare o crittografare il nostro messaggio. Questo rimescolamento viene fatto prendendo lo XOR bit-per-bit del messaggio e dei bit del pad monouso per ogni posizione contenuta nella sequenza. Questo produce una sequenza di bit classici chiamata *testo cifrato*. A chiunque cerchi di leggere il nostro messaggio, il testo cifrato apparirà come una sequenza casuale di bit. Per esempio, è impossibile dire se un bit nel testo cifrato è un 1 a causa del testo in chiaro o del pad monouso.

Potreste chiedervi come otteniamo le stringhe di bit casuali per il nostro pad monouso. Possiamo creare il nostro generatore quantistico di numeri casuali (QRNG) con i qubit! Può sembrare strano, ma simuleremo i qubit con i bit classici per creare il nostro generatore quantistico di numeri casuali. I numeri casuali che genera non saranno più sicuri del computer che useremo per la nostra simulazione, ma questo approccio ci permetterà di iniziare a capire come funzionano i qubit.

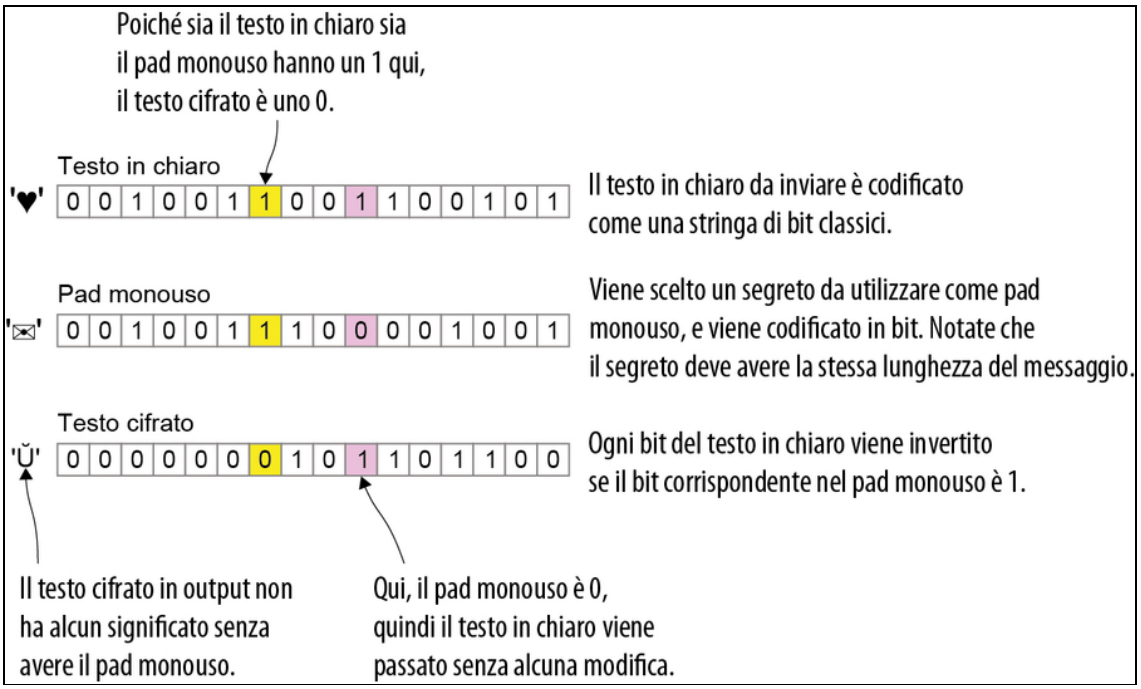


Figura 2.17 Un esempio del modo in cui usare dei bit casuali per crittografare un segreto, anche su Internet o un'altra rete non sicura. In questo caso, stiamo cercando di inviare in modo sicuro il messaggio "♥". Se noi ed Eve iniziamo con un segreto condiviso "✉", allora possiamo utilizzarlo come un pad monouso per proteggere il nostro messaggio.

Mandiamo a Eve il nostro messaggio! Proprio come un bit classico può essere rappresentato nel codice dai valori `True` e `False`, allo stesso modo abbiamo visto che possiamo rappresentare i due stati $|0\rangle$ e $|1\rangle$ di un qubit come vettori. Cioè, gli stati dei qubit sono rappresentati nel codice come liste di liste di numeri.

Listato 2.3 Rappresentare i qubit nel codice con NumPy.

```
>>> import numpy as np (1)
>>> ket0 = np.array( (2)
...     [[1], [0]]
... )
>>> ket0
array([[1], (3)
       [0]])
>>> ket1 = np.array(
...     [[0], [1]]
... )
>>> ket1
array([[0],
       [1]])
```

(1) Usiamo la libreria NumPy per Python per rappresentare i vettori, poiché NumPy è altamente ottimizzata e semplificherà molto le cose.
(2) Chiamiamo la nostra variabile ket0, in base alla notazione $|0\rangle$: etichettiamo gli stati dei qubit con la metà ket delle parentesi $\langle\rangle$.
(3) NumPy mostra i vettori 2×1 come colonne.

Come abbiamo visto in precedenza, possiamo costruire altri stati, come $|+\rangle$, impiegando combinazioni lineari di $|0\rangle$ e $|1\rangle$. Esattamente allo stesso modo, possiamo usare NumPy per sommare le rappresentazioni vettoriali di $|0\rangle$ e $|1\rangle$ per costruire la rappresentazione del vettore $|+\rangle$.

Listato 2.4 La rappresentazione vettoriale di $|+\rangle$.

```
>>> ket_plus = (ket0 + ket1)/np.sqrt(2) (1)
>>> ket_plus
array([[0.70710678 + 0.j], (2)
 [0.70710678 + 0.j]])
```

(1) NumPy utilizza dei vettori per memorizzare lo stato $|+\rangle$, il quale è una combinazione lineare di $|0\rangle$ e $|1\rangle$.

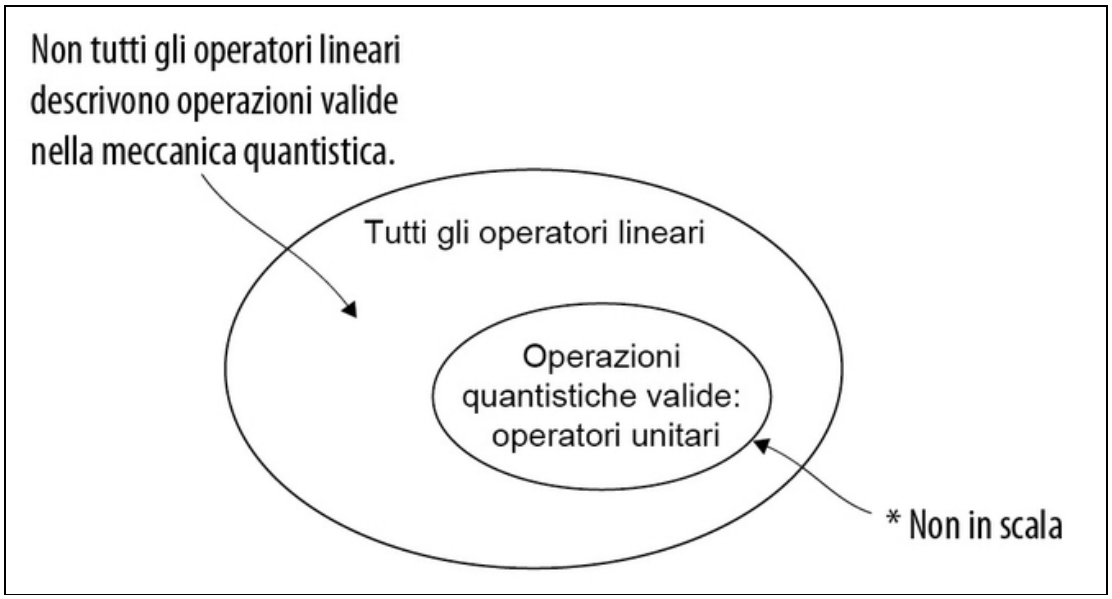
(2) In questo libro parleremo molto del numero 0.70710678, poiché è una buona approssimazione di $\sqrt{2}$, la lunghezza del vettore [[1], [1]].

Nella logica classica, se volessimo simulare come un'operazione trasformerebbe una lista di bit, potremmo usare una tabella di verità. Allo stesso modo, poiché le operazioni quantistiche diverse dalla misurazione sono sempre lineari, per simulare come un'operazione trasformerebbe lo stato di un qubit, possiamo usare una matrice, che ci dice come viene trasformato ogni singolo stato.

Operatori lineari e operazioni quantistiche

Descrivere le operazioni quantistiche tramite operatori lineari è un buon inizio, ma non tutti gli operatori lineari sono operazioni quantistiche valide. Se potessimo implementare un'operazione descritta da un operatore lineare come $2 \times \mathbb{1}$ (cioè il doppio dell'operatore di identità), allora violeremmo il fatto che le probabilità sono sempre numeri compresi tra 0 e 1. Richiediamo anche che tutte le operazioni quantistiche diverse dalla misurazione siano reversibili, poiché questa è una proprietà fondamentale della meccanica quantistica.

Di conseguenza, le operazioni realizzabili in meccanica quantistica sono descritte da matrici U le cui inverse U^{-1} possono essere calcolate prendendo la coniugata trasposta, $U^{-1} = U^{+}$. Tali matrici sono chiamate *matrici unitarie*.



Rappresentazione di tipi di operazioni quantistiche valide. Tutti gli operatori unitari sono lineari, ma non tutti gli operatori lineari sono unitari. Le operazioni quantistiche reversibili (ovvero diverse dalla misurazione) sono rappresentate da operatori non solo lineari ma unitari.

Un'operazione quantistica particolarmente importante è l'*operazione di Hadamard*, che trasforma $|0\rangle$ in $|+\rangle$ e $|1\rangle$ in $|-\rangle$. Come abbiamo già visto, misurando $|+\rangle$ lungo l'asse Z si ottiene un risultato 0 o 1 con uguale probabilità. Poiché vogliamo ottenere bit casuali per inviare messaggi segreti, l'operazione di Hadamard è davvero utile per creare il nostro generatore quantistico di numeri casuali.

Impiegando vettori e matrici, possiamo definire l'operazione di Hadamard creando una tabella del modo in cui essa agisce sugli stati $|0\rangle$ e $|1\rangle$, come mostrato nella Tabella 2.4.

Poiché la meccanica quantistica è lineare, questa è una descrizione pienamente completa dell'operazione di Hadamard.

Tabella 2.4 Rappresentazione dell'operazione di Hadamard sotto forma di tabella.

Stato di input	Stato di output
$\|0\rangle$	$\|+\rangle = (\|0\rangle + \|1\rangle) / \sqrt{2}$

$\|1\rangle$	$\|-\rangle = (\|0\rangle - \|1\rangle) / \sqrt{2}$

In forma matriciale, scriviamo la Tabella 2.4 come `H = np.array([[1, 1], [1, -1]]) / np.sqrt(2)`.

Listato 2.5 Definizione dell'operazione di Hadamard.

```
>>> H = np.array([[1, 1], [1, -1]])/np.sqrt(2) (1)
>>> H @ ket0
array([[0.70710678],
       [0.70710678]])
>>> H @ ket1
array([[ 0.70710678],
       [-0.70710678]])
```

(1) Definiamo una variabile `H` per contenere la rappresentazione matriciale *H* dell'operazione di Hadamard che abbiamo descritto nella Tabella 2.4. Impiegheremo *H* in tutto il resto di questo capitolo, quindi è utile definirla qui.

L'operazione di Hadamard

L'operazione di Hadamard è un'operazione quantistica che può essere simulata da questa trasformazione lineare:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

Qualsiasi operazione su dati quantistici può essere scritta come una matrice in questo modo. Se desideriamo trasformare $|0\rangle$ in $|1\rangle$ e viceversa (la generalizzazione quantistica dell'operazione NOT classica che abbiamo già visto, corrispondente a una rotazione di 180°), facciamo la stessa cosa che abbiamo fatto per definire l'operazione di Hadamard.

Listato 2.6 Rappresentazione della porta NOT quantistica.

```
>>> X = np.array([[0, 1], [1, 0]]) (1)
>>> X @ ket0
array ([[0],
        [1]])
>>> (X @ ket0 == ket1).all() (2)
True
>>> X @ H @ ket0 (3)
array([[0.70710678],
       [0.70710678]])
```

(1) L'operazione quantistica corrispondente all'operazione NOT classica è tipicamente chiamata operazione x; rappresentiamo la matrice per x con la variabile Python `X`.

(2) Possiamo confermare che `X` trasforma |0⟩ in |1⟩. Il metodo NumPy `all()` restituisce `True` se ogni elemento di `X @ ket0 == ket1` è `True`: in pratica, se ogni elemento dell'array `X @ ket0` è uguale all'elemento corrispondente di `ket1`.

(3) L'operatore x non fa nulla a $H|0\rangle$. Possiamo confermarlo impiegando di nuovo l'operatore `@` per moltiplicare `X` per un valore Python che rappresenta lo stato $|+\rangle = H|0\rangle$. Possiamo esprimere quel valore come `H @ ket0`.

L'operazione x non fa nulla sull'ultimo input, perché scambia |0⟩ e |1⟩. Lo stato $H|0\rangle$, chiamato anche |+⟩, è già una somma dei due ket: $(|0\rangle + |1\rangle) / \sqrt{2} = (|1\rangle + |0\rangle) / \sqrt{2}$, così un'azione di scambio dell'operazione x non fa nulla.

Ricordando l'analogia della mappa dell'Appendice C, possiamo pensare alla matrice *H* come a una *riflessione* sulla direzione ↗, come illustrato nella Figura 2.18.

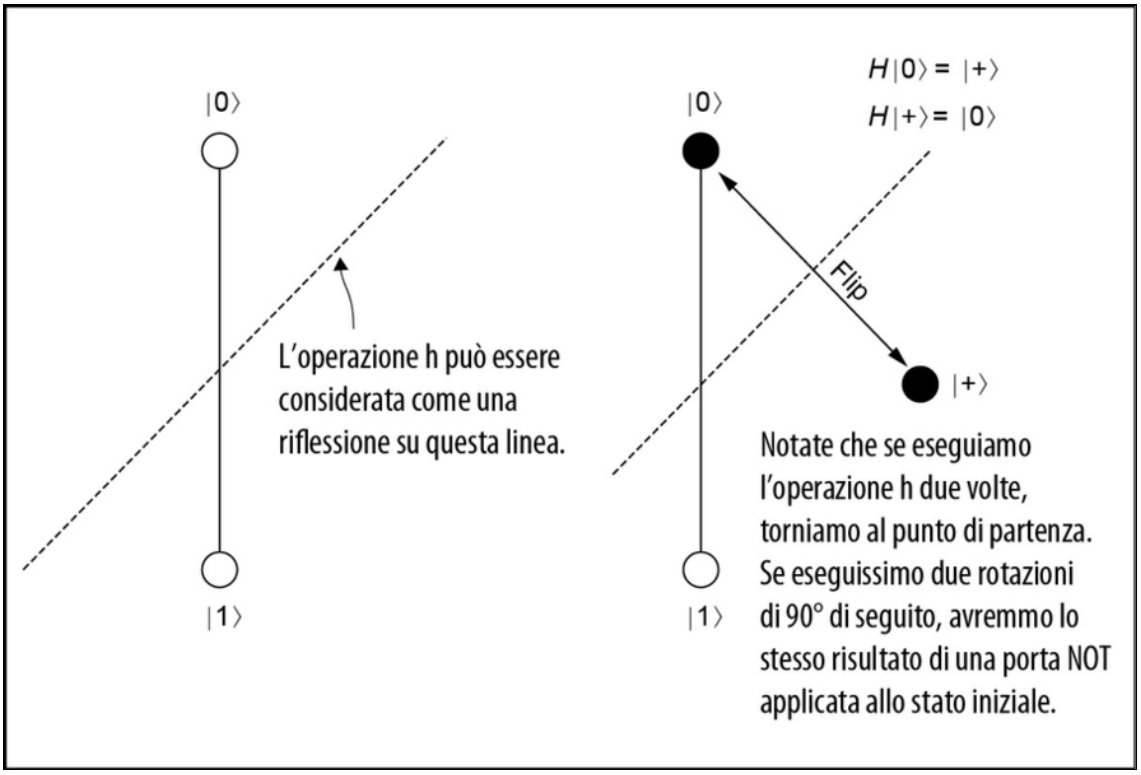
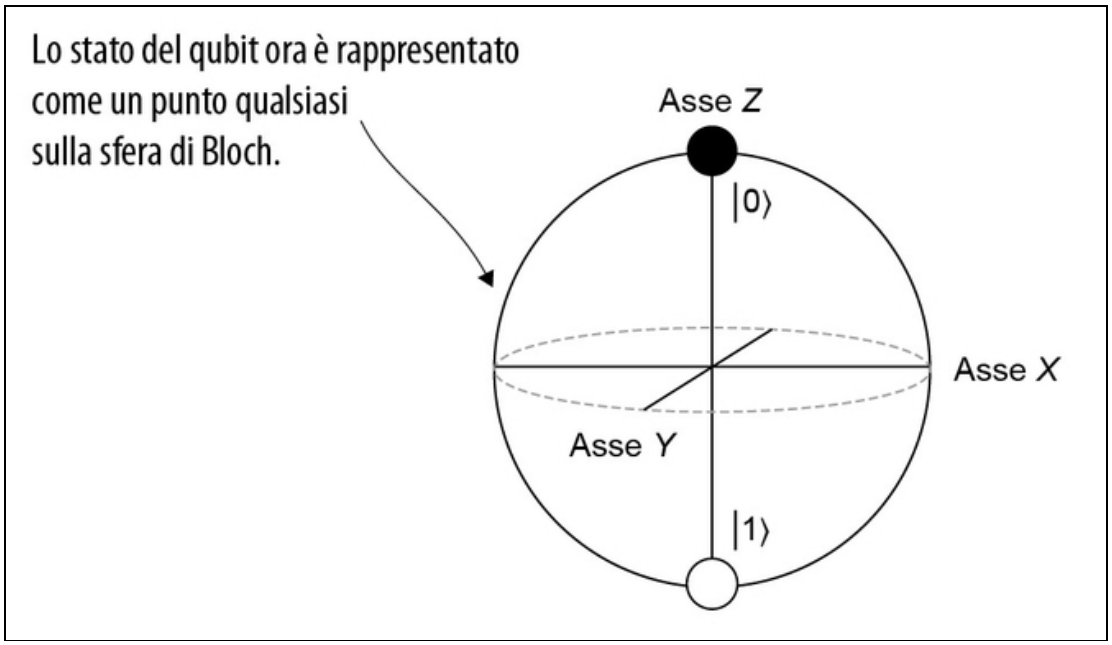


Figura 2.18 L'operazione h come una riflessione o un capovolgimento lungo ⤢. A differenza di una rotazione di 90°, l'applicazione di h due volte restituisce un qubit nello stato di partenza. Un altro modo di immaginare h è che la riflessione su ⤢ scambi il ruolo degli assi X e Z.

Arriva la terza dimensione

Per i qubit, l'analogia con la mappa dell'Appendice C ci aiuta a capire come scrivere e manipolare gli stati dei singoli qubit. Finora, tuttavia, abbiamo esaminato solo gli stati che possono essere scritti utilizzando numeri reali. In generale, gli stati quantistici possono utilizzare numeri complessi. Se riorganizziamo la nostra mappa e la rendiamo tridimensionale, possiamo includere i numeri complessi senza alcun problema. Questo modo di pensare ai qubit è chiamato *sfera di Bloch* e può essere molto utile per considerare le operazioni quantistiche come rotazioni e riflessioni, come vedremo nel Capitolo 6.



Più in generale, possiamo rappresentare gli stati dei qubit come punti su una sfera, non solo su un cerchio. Così facendo dobbiamo però ricorrere ai numeri complessi. Ne riparleremo nel Capitolo 6.

Approfondimento: infiniti stati?

Può sembrare, dalla figura nel riquadro precedente, che un qubit possa assumere infiniti stati differenti. Per due punti distinti su una sfera, possiamo sempre trovare un punto che si trova ”tra” di essi. Questo è vero, ma può anche essere un po' fuorviante. Pensando per un momento alla situazione classica, una moneta che dà testa il 90% delle volte è distinta da una moneta che dà testa il 90,0000000001% delle volte. In effetti, possiamo sempre creare una moneta il cui errore è “tra” l'errore di altre due monete. Lanciare una moneta può darci solo un bit classico, però. In media, ci vorrebbero circa 10^{23} lanci per distinguere una moneta che dà testa il 90% delle volte da una moneta che dà testa il 90,0000000001% delle volte. Così possiamo trattare queste due monete come identiche, se non possiamo trovare un esperimento che le distingua in modo affidabile. Allo stesso modo, per il quantum computing, ci sono limiti alla nostra capacità di distinguere gli infiniti stati quantistici distinti che possiamo riconoscere dall'immagine della sfera di Bloch.

Il fatto che un qubit abbia infiniti stati non è ciò che lo rende unico. A volte si dice che un sistema quantistico può trovarsi “in infiniti stati contemporaneamente”,

motivo per cui si dice che i computer quantistici possono offrire vantaggi prestazionali. *Questo è falso!* Come sottolineato in precedenza, non possiamo distinguere gli stati molto vicini tra loro, quindi il vantaggio del quantum computing non può essere dovuto a questi "infiniti stati". Parleremo meglio della parte "contemporaneamente" nei prossimi capitoli, ma per ora basti sapere che non è il numero di stati in cui può trovarsi il nostro qubit a rendere i computer quantistici così straordinari.

Programmazione di un generatore quantistico di numeri casuali (QRNG)

Ora che abbiamo "in mano" un paio di concetti quantistici con i quali giocare, cerchiamo di applicare ciò che abbiamo imparato alla programmazione di un QRNG, in modo che possiamo inviare i nostri 💔 senza preoccupazioni. Costruiremo un QRNG che restituisca uno 0 o un 1.

Bit casuali o numeri casuali?

Può sembrare limitante il fatto che il nostro generatore quantistico di numeri casuali possa emettere solo due numeri, 0 o 1. Al contrario: questo è sufficiente per generare numeri casuali nell'intervallo da 0 a N per qualsiasi numero intero positivo N. È più facile vederlo a partire dal caso speciale che N sia 2^n - 1 per qualche positivo intero n, nel qual caso scriviamo semplicemente i nostri numeri casuali come stringhe di n bit. Per esempio, possiamo creare numeri casuali compresi tra 0 e 7 generando tre bit casuali r0, r1 e r2 e quindi restituendo 4r2 + 2r1 + r0.

Il caso è leggermente più complicato se N non è dato da una potenza di 2, in quanto dobbiamo gestire alcune possibilità "rimanenti". Per esempio, se dobbiamo tirare un dado a sei facce ma abbiamo solo un dado a otto facce (magari perché eravamo druidi nel gioco di ruolo della sera precedente), dobbiamo decidere che cosa fare quando quel dado dà un 7 o un 8. La cosa migliore che possiamo fare, se vogliamo un dado equo a sei facce, è semplicemente ripetere il tiro. Impiegando questo approccio, possiamo costruire dadi equi arbitrari a partire da lanci di monete, utili

per qualsiasi gioco. Per farla breve, il nostro generatore quantistico di numeri casuali non si limita a due soli risultati.

Come per qualsiasi programma quantistico, il nostro programma QRNG sarà una sequenza di istruzioni per un dispositivo che esegue operazioni su un qubit (vedi Figura 2.19). In pseudocodice, un programma quantistico per l'implementazione di un QRNG è costituito da tre istruzioni.

1. Preparare un qubit nello stato $|0\rangle$.
2. Applicare al nostro qubit l'operazione di Hadamard, in modo che sia nello stato $|+\rangle = H|0\rangle$.
3. Misurare il qubit per ottenere un risultato 0 o 1 con probabilità 50/50.

Cioè, vogliamo un programma che somigli al seguente.

Listato 2.7 Pseudocodice di esempio per un programma QRNG.

```
def qrng():
    q = Qubit()
    H(q)
    return measure(q)
```

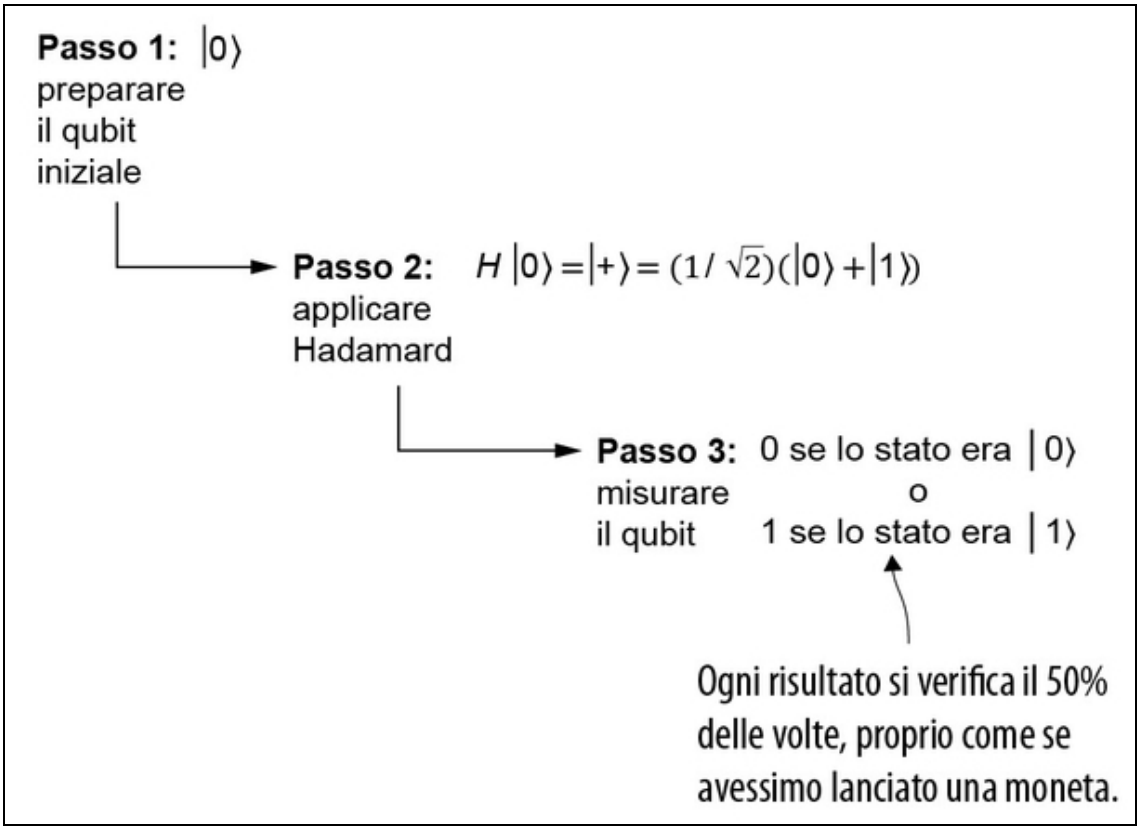


Figura 2.19 Passaggi per scrivere il programma QRNG che vogliamo sottoporre a test. Tornando a osservare la Figura 2.3, possiamo usare ciò che abbiamo imparato finora per scrivere lo stato del nostro qubit dopo ogni passaggio dell'algoritmo QRNG.

Impiegando la moltiplicazione di matrici, possiamo usare un computer classico come un laptop per simulare come `qrng()` agirebbe su un dispositivo quantistico ideale. Il nostro programma `qrng` viene richiamato in uno stack software (vedi Figura 2.20) che astrae il fatto che stiamo impiegando un simulatore classico o un vero e proprio dispositivo quantistico.

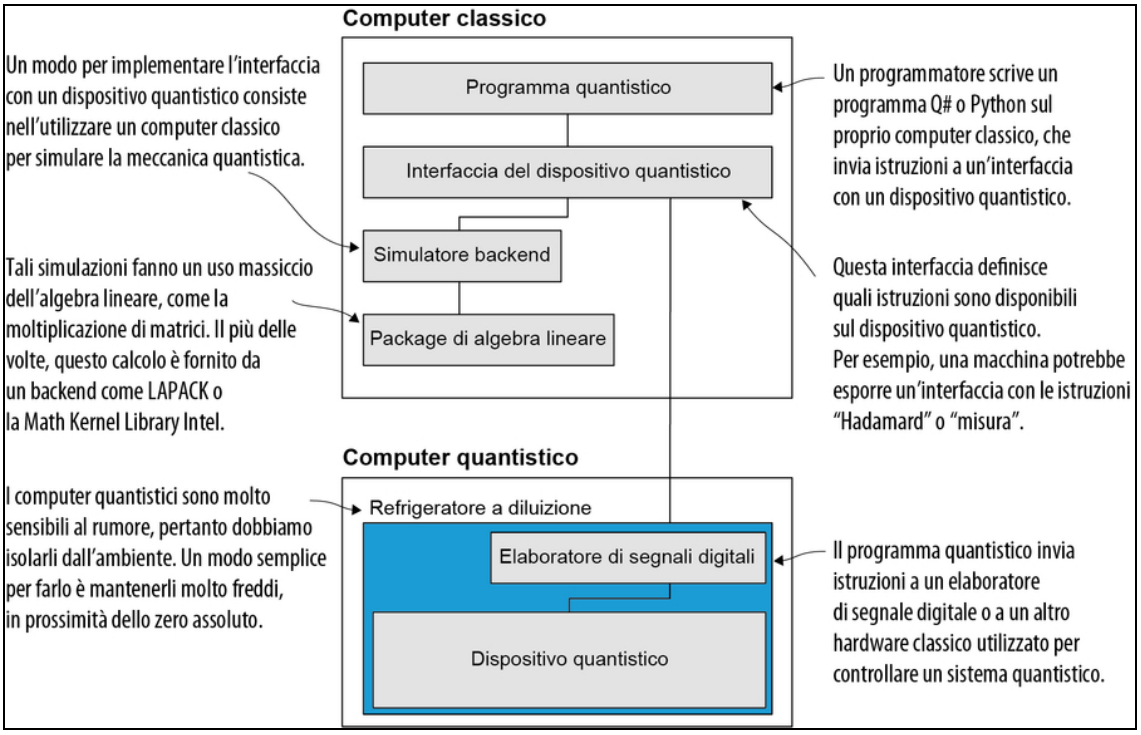


Figura 2.20 Un esempio dell'aspetto di uno stack software per un programma quantistico.

Lo stack è costituito da molte parti, ma non preoccuparvi: le introdurremo man mano che procederemo. Per ora, ci stiamo concentrando sulla sezione superiore (il "Computer classico" nella figura) e inizieremo scrivendo il codice per un programma quantistico e un simulatore backend in Python.

NOTA

Nel Capitolo 7 ci concentreremo sull'utilizzo del simulatore backend fornito con il Quantum Development Kit Microsoft.

Sulla base di questo stack software, possiamo scrivere la nostra simulazione di un generatore quantistico di numeri casuali scrivendo prima una classe `QuantumDevice` con dei metodi astratti per allocare i qubit, per eseguire operazioni e per misurare i qubit. Potremo poi

implementare questa classe con un simulatore e richiamare tale simulatore da `qrng()`.

Per progettare l'interfaccia per il nostro simulatore nel modo rappresentato nella Figura 2.20, elenchiamo quello che deve fare il nostro dispositivo quantistico. Innanzitutto, gli utenti devono essere in grado di allocare e restituire qubit.

Listato 2.8 Interface.py: interfaccia con un dispositivo quantistico, implementata con metodi astratti.

```
class QuantumDevice(metaclass=ABCMeta):
    @abstractmethod
    def allocate_qubit(self) -> Qubit: (1)
        pass
    @abstractmethod
    def deallocate_qubit(self, qubit: Qubit): (2)
        pass
    @contextmanager
    def using_qubit(self): (3)
        qubit = self.allocate_qubit()
        try:
            yield qubit
        finally:
            qubit.reset() (4)
            self.deallocate_qubit(qubit)
```

(1) Qualsiasi implementazione di un dispositivo quantistico deve implementare questo metodo, che consente agli utenti di ottenere i qubit.
(2) Quando gli utenti hanno finito di manipolare un qubit, le implementazioni di `deallocate_qubit` consentiranno loro di restituire il qubit al dispositivo.
(3) Possiamo fornire un gestore del contesto Python per semplificare l'allocazione e la deallocazione dei qubit in modo sicuro.
(4) Il gestore del contesto si assicura che, indipendentemente dalle eccezioni sollevate, ogni qubit venga ripristinato e deallocato prima di essere restituito al computer classico.

I qubit stessi possono quindi esporre le effettive trasformazioni di cui abbiamo bisogno:

- gli utenti devono essere in grado di eseguire operazioni Hadamard sui qubit;
- gli utenti devono essere in grado di misurare i qubit per ottenere i dati classici.

Listato 2.9 interface.py: interfaccia nei qubit su un dispositivo quantistico.

```
from abc import ABCMeta, abstractmethod
from contextlib import contextmanager
class Qubit(metaclass=ABCMeta):
    @abstractmethod
    def h(self): pass (1)
    @abstractmethod (2)
    def measure(self) -> bool: pass
    @abstractmethod
    def reset(self): pass (3)
```

(1) Il metodo `h` trasforma un qubit *in loco* (senza crearne una copia) utilizzando l'operazione di Hadamard `np.array([[1, 1], [1, -1]])/np.sqrt(2)`.

(2) Il metodo `measure` consente agli utenti di misurare i qubit ed estrarvi dati classici.

(3) Il metodo `reset` aiuta a preparare nuovamente il qubit da zero.

Ora, possiamo tornare alla nostra definizione di `qrng` impiegando queste nuove classi.

Listato 2.10 qrng.py: definizione del dispositivo qrng.

```
def qrng(device: QuantumDevice) -> bool:
    with device.using_qubit() as q:
        q.h()
        return q.measure()
```

Se implementiamo l'interfaccia `QuantumDevice` con una classe chiamata `SingleQubitSimulator`, possiamo passarla a `qrng` per eseguire la nostra implementazione del QRNG su un simulatore.

Listato 2.11 qrng.py: definizione di main per qrng.py.

```
if __name__ == "__main__":
    qsim = SingleQubitSimulator()
```

```
    for idx_sample in range(10):
        random_sample = qrng(qsim)
        print(f"Our QRNG returned {random_sample}.")
```

Ora abbiamo tutto il necessario per scrivere `SingleQubitSimulator`. Iniziamo definendo un paio di costanti per il vettore $|0\rangle$ e la rappresentazione matriciale dell'operazione di Hadamard.

Listato 2.12 simulator.py: definizione di costanti utili.

```
KET_0 = np.array([
    [1],
    [0]
], dtype=complex) (1)
H = np.array([
    [1, 1],
    [1, -1]
], dtype=complex) / np.sqrt(2) (2)
```

(1) Dal momento che useremo molto $|0\rangle$ nel nostro simulatore, definiamo una costante apposita.

(2) Allo stesso modo, useremo la matrice `H` di Hadamard per definire come l'operazione di Hadamard trasforma gli stati, quindi definiamo una costante anche per lei.

Successivamente, definiamo l'aspetto di un qubit simulato. Dal punto di vista di un simulatore, un qubit avvolge un vettore che memorizza lo stato corrente del qubit. Usiamo un array NumPy per rappresentare lo stato del nostro qubit.

Listato 2.13 simulator.py: definizione di una classe per rappresentare i qubit nel nostro dispositivo.

```
class SimulatedQubit(Qubit):
    def __init__(self): (1)
        self.reset()
    def h(self):
        self.state = H @ self.state (2)
    def measure(self) -> bool:
        pr0 = np.abs(self.state[0, 0]) ** 2 (3)
        sample = np.random.random() <= pr0 (4)
        return bool(0 if sample else 1) (5)
    def reset(self):
        self.state = KET_0.copy()
```

(1) L'operazione di Hadamard può essere simulata applicando la rappresentazione della matrice `H` allo stato che stiamo memorizzando al momento e quindi aggiornandolo al nuovo stato.

(2) Come parte dell'interfaccia `Qubit`, ci assicuriamo che il metodo `reset` prepari il nostro qubit nello stato |0⟩. Possiamo usarlo quando creiamo il qubit, per assicurarci che tutti i qubit inizino sempre nello stato corretto.

(3) Abbiamo memorizzato lo stato del nostro qubit come un vettore, quindi sappiamo che il prodotto interno con |0⟩ è semplicemente il primo elemento di quel vettore. Per esempio, se lo stato è `np.array([[a], [b]])` per alcuni numeri `a` e `b`, la probabilità di osservare un risultato `0` è `|a|2`. Possiamo trovarlo impiegando `np.abs(a)**2`. Questo ci dà la probabilità che una misurazione del nostro qubit restituisca `0`.

(4) Per trasformare la probabilità di ottenere uno `0` come risultato di una misurazione, generiamo un numero casuale compreso tra `0` e `1` con `np.random.random` e controlliamo se è minore di `pr0`.

(5) Infine, restituiamo `0` al chiamante se abbiamo uno `0` (e *1* se abbiamo un `1`).

Quale numero casuale è arrivato per primo: 0 o 1?

Nel creare questo generatore quantistico di numeri casuali, dobbiamo richiamare un generatore *classico* di numeri casuali. Questa affermazione può sembrare circolare, ma ciò accade perché la nostra simulazione classica è proprio questo: una simulazione. Una simulazione di un QRNG non è più casuale dell'hardware e del software che utilizziamo per implementare tale simulatore.

Detto questo, il programma quantistico `qrng.py` non ha bisogno di richiamare un generatore classico ma lo chiama nel simulatore. Se dovessimo eseguire `qrng.py` su un dispositivo quantistico reale, il simulatore e il generatore classico sarebbero sostituiti dalle operazioni sul vero qubit. A quel punto, avremmo un flusso di

numeri casuali che sarebbe impossibile prevedere, grazie alle leggi della meccanica quantistica.

Eseguendo il nostro programma, otterremo i numeri casuali che ci aspettavamo.

```
$ python qrng.py
Our QRNG returned False.
Our QRNG returned True.
Our QRNG returned True.
Our QRNG returned False.
Our QRNG returned False.
Our QRNG returned True.
Our QRNG returned False.
Our QRNG returned False.
Our QRNG returned False.
Our QRNG returned True.
```

Congratulazioni! Non solo avete scritto il vostro primo programma quantistico, ma avete anche scritto un backend di simulazione e lo avete utilizzato per eseguire il vostro programma quantistico, così come lo avreste eseguito su un vero computer quantistico.

Approfondimento: il gatto di Schrödinger

Potreste aver già visto o sentito parlare di questo programma quantistico con un altro nome. Il programma QRNG è spesso descritto in termini di esperimento mentale del *gatto di Schrödinger*. Supponiamo che un gatto si trovi in una scatola chiusa con una fiala di veleno che verrà rilasciato se si verifica il decadimento di una determinata particella casuale. Prima di aprire la scatola per controllare, come facciamo a sapere se il gatto è vivo o morto?

> Lo [stato] dell'intero sistema esprimerebbe questo concetto contenendo il gatto vivo e morto (scusate l'espressione) mescolati in parti uguali.
> - *Erwin Schrödinger*

Storicamente, Schrödinger propose questa descrizione nel 1935 per esprimere il suo punto di vista secondo cui alcune implicazioni della meccanica quantistica sarebbero "ridicole", per mezzo di un esperimento mentale che evidenzia quanto controintuitive siano le implicazioni. Tali esperimenti mentali, noti come *gedankenexperiment*, rappresentano una grande tradizione della fisica e possono aiutarci a comprendere o criticare diverse teorie, spingendole a limiti estremi o all'assurdo.

Leggendo del gatto di Schrödinger quasi un secolo dopo, tuttavia, è utile ricordare tutto ciò che è accaduto negli anni successivi. Dalla sua lettera originale, il mondo ha visto:

- una guerra su una scala mai sperimentata in precedenza;
- i primi passi dell'umanità al di là del nostro pianeta;
- il boom dei viaggi su jet commerciali;
- la consapevolezza e i primi effetti dei cambiamenti climatici antropogenici;
- un cambiamento fondamentale nel modo in cui comunichiamo (dalla televisione fino a Internet);
- un'ampia disponibilità di dispositivi informatici a basso prezzo;
- la scoperta di una meravigliosa varietà di particelle subatomiche.

Il mondo in cui viviamo non è lo stesso mondo in cui Schrödinger ha cercato di dare un senso alla meccanica quantistica. Oggi abbiamo numerosi vantaggi nel cercare di capire, non ultimo il fatto che possiamo mettere rapidamente le mani sulla meccanica quantistica programmando simulazioni operanti su computer classici. Per esempio, l'istruzione `h` che abbiamo appena visto mette il nostro qubit in una situazione simile al gatto dell'esperimento, ma con il vantaggio che è molto più facile sperimentare con il nostro programma che con un esperimento mentale. Per tutto il resto del libro useremo programmi quantistici per scoprire quelle parti della meccanica quantistica di cui abbiamo bisogno di scrivere algoritmi quantistici.

Riepilogo

- I numeri casuali aiutano in un'ampia gamma di applicazioni, come il gioco, la simulazione di sistemi complessi e la protezione dei dati.
- I bit classici possono trovarsi in due stati, che tradizionalmente chiamiamo 0 e 1.
- Gli analoghi quantistici dei bit classici, chiamati *qubit*, possono essere nello stato $|0\rangle$ o $|1\rangle$ o in sovrapposizioni di $|0\rangle$ e $|1\rangle$; per esempio, $|+\rangle = 1 / \sqrt{2} (|0\rangle + |1\rangle)$.

- Impiegando l’operazione di Hadamard, possiamo preparare qubit nello stato $|+\rangle$; possiamo poi misurare i qubit per generare numeri la cui casualità è garantita dalle leggi della meccanica quantistica.

Condivisione di segreti con la distribuzione quantistica delle chiavi

Nel capitolo precedente, abbiamo iniziato a giocare con i qubit e li abbiamo usati per costruire un generatore quantistico di numeri casuali con un simulatore che abbiamo costruito in Python. In questo capitolo, vedremo che i qubit possono aiutarci con la crittografia (o altre attività crittografiche) permettendoci di *distribuire* in modo sicuro le chiavi segrete. Esistono metodi classici per la condivisione di chiavi casuali (per esempio, RSA), che offrono garanzie differenti sulla sicurezza della condivisione.

Tutto è lecito in amore e crittografia

Ora abbiamo un generatore quantistico di numeri casuali, preparato nel Capitolo 2, che però è solo la metà di ciò di cui abbiamo bisogno per condividere i segreti con i nostri amici. E dobbiamo condividere quei numeri casuali con i nostri amici, se vogliamo utilizzare i numeri casuali quantistici per comunicare in modo sicuro con loro. Quei numeri casuali (normalmente chiamati *chiavi*) possono essere utilizzati dagli *algoritmi di crittografia*, che combinano la casualità della chiave con le informazioni che occorre mantenere segrete, in modo tale che solo chi abbia la chiave possa leggere le informazioni. Nella Figura 3.1

possiamo vedere come due persone potrebbero usare una chiave (qui, una stringa binaria casuale) per crittografare e decrittografare i messaggi che si scambiano.

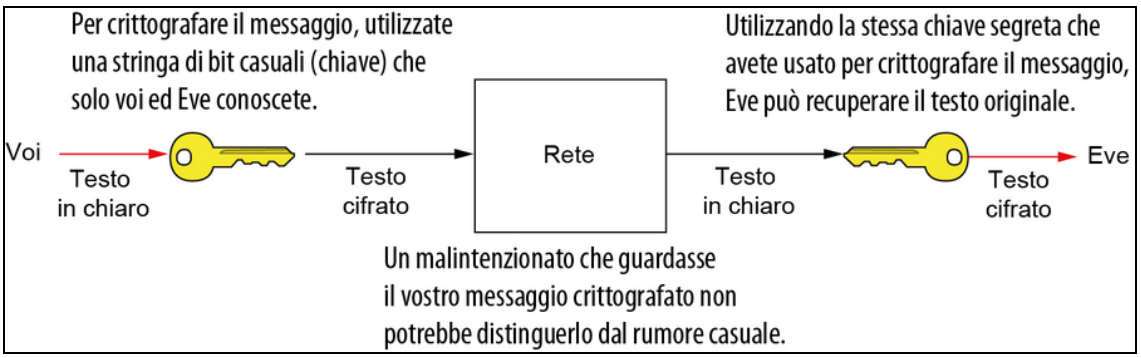


Figura 3.1 Modello mentale del modo in cui noi ed Eve potremmo usare la crittografia per comunicare segretamente, anche su Internet o su un'altra rete non affidabile.

Se vogliamo coinvolgere i qubit, possiamo mostrare che la distribuzione quantistica delle chiavi (QKD) è *dimostrabilmente sicura*, mentre i metodi classici di distribuzione delle chiavi sono, spesso, solo *computazionalmente sicuri*.

DEFINIZIONE

La *distribuzione quantistica delle chiavi* (QKD) è un protocollo di comunicazione che consente agli utenti di condividere numeri casuali quantistici scambiando qubit e informazioni classiche autenticate.

Questa differenza non ha importanza per la maggior parte dei casi d'uso. Ma se siamo un governo, un gruppo di attivisti, una banca, un giornalista, una spia o qualsiasi altra entità in cui la sicurezza delle informazioni è una questione di vita o di morte, questa differenza è enormemente importante.

Sicurezza computazionale o dimostrabile

La sicurezza dimostrabile per i protocolli crittografici è il massimo. Un metodo o un protocollo per un'attività crittografica è *dimostrabilmente sicuro* se possiamo scrivere una dimostrazione del fatto che è sicuro senza fare supposizioni

sull'avversario: potrebbe avere tutto il tempo e la potenza di calcolo dell'universo e il nostro protocollo rimarrebbe ancora sicuro. La maggior parte della nostra attuale infrastruttura crittografica è *computazionalmente sicura*, il che garantisce la sicurezza di un metodo o protocollo con presupposti ragionevoli sulle capacità di calcolo di un malintenzionato. Il progettista o l'utilizzatore del protocollo può impostare apposite soglie in termini di risorse di calcolo (per esempio, il più grande supercomputer attuale o tutti i computer del pianeta) e di tempo a disposizione (100 anni, 10.000 anni, l'età dell'universo).

Quando condividiamo una chiave utilizzando QKD, questo non garantisce che la chiave raggiunga l'altra persona, perché qualcuno può sempre eseguire un attacco *Denial of Service* (per esempio tagliando fisicamente la connessione tra mittente e destinatario), come avviene per qualsiasi altro protocollo classico. Una buona analogia per ciò che QKD può promettere è un sigillo a prova di manomissione sui prodotti alimentari. Quando un produttore di burro di arachidi vuole garantire che quando noi apriamo il barattolo, lo troviamo esattamente come quando ha lasciato la fabbrica, l'azienda appone sul vasetto un sigillo a prova di manomissione. L'azienda promette che se il sigillo che ha fatto per noi (i consumatori) è intatto, il burro di arachidi sarà buono e nessun terzo lo avrà manipolato. Trasmettere una chiave crittografica con un protocollo QKD è come mettere un sigillo a prova di manomissione sui bit in transito. Se qualcuno cercasse di compromettere la chiave durante il transito, il destinatario lo scoprirebbe e così non la utilizzerà. Tuttavia, sigillare i bit in transito non garantisce che i bit arrivino al destinatario.

Possiamo usare molti protocolli per implementare lo schema QKD generale. In questo capitolo lavoreremo con uno dei protocolli QKD più comuni, BB84, ma ce ne sono molti altri che non abbiamo il tempo di trattare. In questo capitolo, ci concentreremo solo su questo; la Figura 3.2 mostra i passaggi svolti dal protocollo BB84.

QKD è un esempio di un programma quantistico che utilizza un singolo qubit e una tecnologia derivata dal quantum computing. Ciò che lo rende interessante da sviluppare è che abbiamo l'hardware per implementarlo già oggi. Diverse aziende vendono hardware QKD da circa 15 anni, ma i prossimi passi della tecnologia riguarderanno la valutazione della sicurezza hardware e software di questi sistemi.

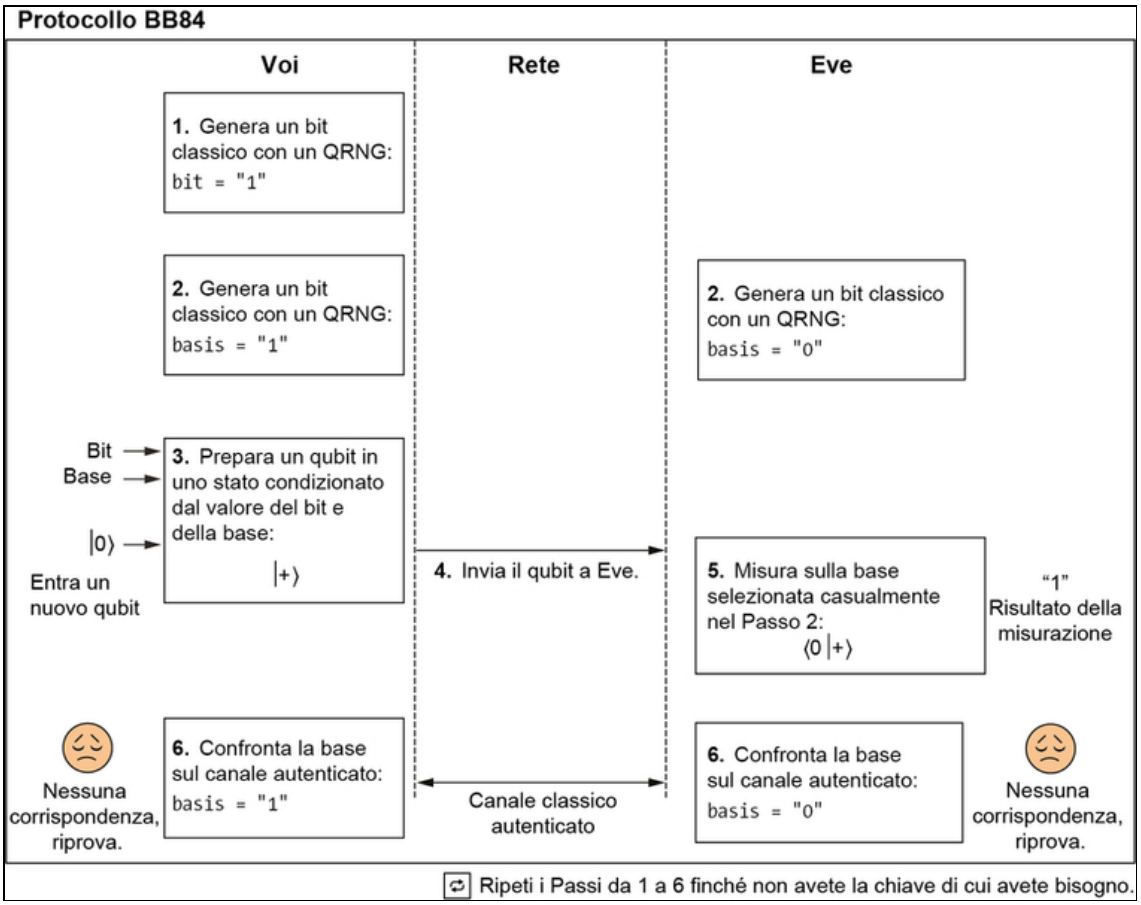


Figura 3.2 Diagramma temporale per il protocollo BB84, una variante di un protocollo QKD.

ATTENZIONE

Gli esempi che stiamo implementando e che utilizziamo in questo libro *simulano* i protocolli dimostrabilmente sicuri. Dal momento che non stiamo eseguendo gli esempi su dispositivi quantistici, non possono essere dimostrabilmente sicuri. Anche quando si implementano questi protocolli su

vero hardware quantistico, queste dimostrazioni di sicurezza non fanno nulla per impedire che si verifichino altri tipi di attacchi (magari di ingegneria sociale) per sottrarci la nostra chiave, 😈. Parleremo di queste dimostrazioni più avanti in questo capitolo, quando discuteremo del teorema di non-clonazione.

Vediamo ora come funziona il protocollo QKD. Per i nostri scopi, diciamo che noi ed Eve vogliamo scambiarci una chiave in modo da poterci inviare dei messaggi segreti 😉. Lo scenario è il seguente:

Vorremmo inviare un messaggio segreto alla nostra amica. Usando il nostro generatore quantistico di numeri casuali del Capitolo 2, il protocollo QKD BB84 e la crittografia a pad monouso, progettiamo un programma per inviare messaggi dimostrabilmente sicuri.

Possiamo rappresentare lo scenario come una sorta di diagramma temporale (Figura 3.3).

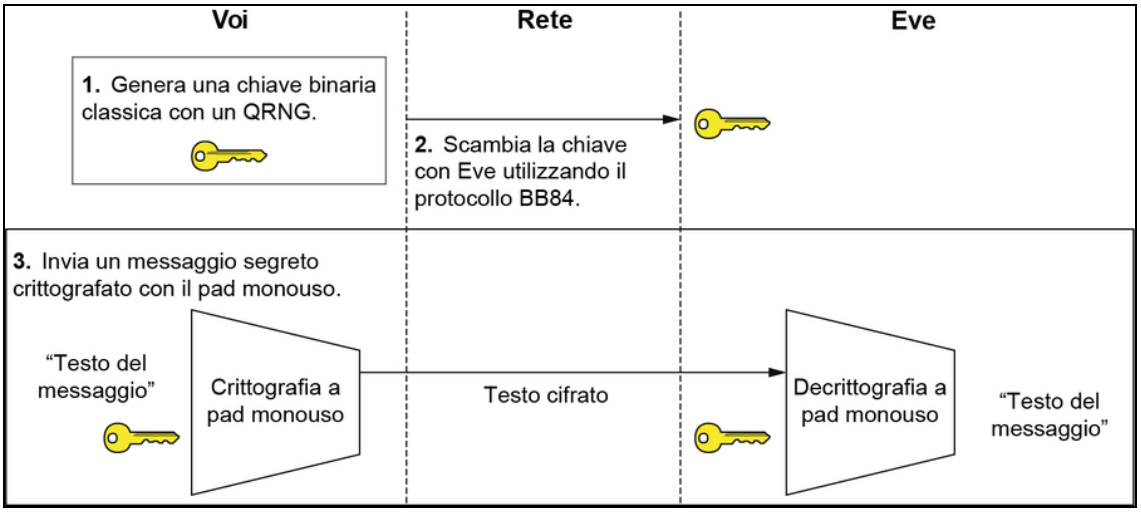


Figura 3.3 Lo scenario di questo capitolo: dobbiamo inviare un messaggio segreto a Eve con BB84 e la crittografia a pad monouso. Prima, però, dobbiamo scambiare una chiave segreta con Eve, in modo da poterla utilizzare per crittografare il messaggio che vogliamo inviare. Possiamo usare i qubit e gli stati di sovrapposizione per facilitare il passaggio di scambio delle chiavi.

Notate che la chiave che dobbiamo inviare è una stringa di bit classici. Come possiamo usare i qubit per inviare quei bit classici?

Cominciamo vedendo come codificare le informazioni classiche in qubit e poi vedremo i passaggi specifici del protocollo BB84. Nel prossimo paragrafo esamineremo una nuova operazione quantistica che ci aiuterà a codificare i bit classici in qubit.

Operazioni NOT quantistiche

Se abbiamo alcune informazioni classiche, diciamo un singolo bit binario, come possiamo codificarle in una risorsa quantistica come un qubit? Il seguente algoritmo invia una stringa di bit classici casuali codificata in qubit.

1. Utilizza un generatore quantistico di numeri casuali per generare un bit casuale da inviare come chiave.
2. Inizia con un qubit nello stato $|0\rangle$ e poi preparalo in uno stato che rappresenti il valore del bit del Passo 1. Qui, usa $|0\rangle$ se il bit classico era 0 e $|1\rangle$ se il bit classico era un 1.
3. Il qubit così preparato viene inviato a Eve, che lo misura e registra il valore del bit classico.
4. Ripeti i Passi da 1 a 3 finché noi ed Eve non abbiamo la chiave che vogliamo (di solito stabilita dal protocollo crittografico che vogliamo usare dopo).

La Figura 3.4 mostra un diagramma temporale per questo algoritmo.

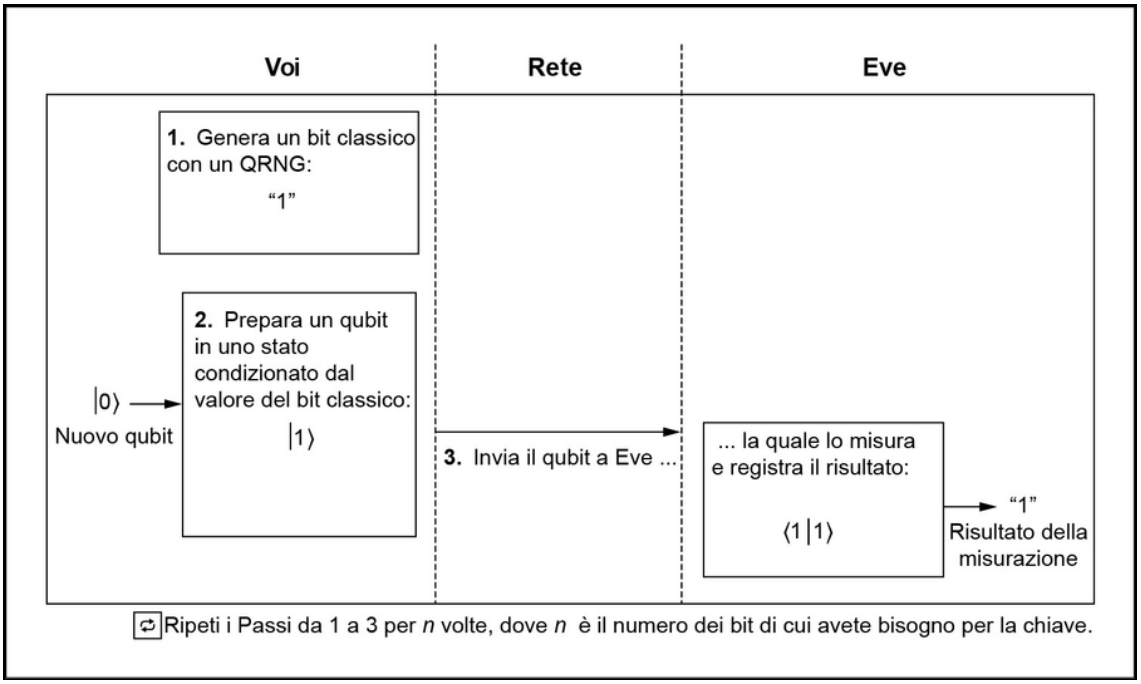


Figura 3.4 Una rappresentazione dell'algoritmo per l'invio di una classica stringa di bit con qubit. Iniziamo utilizzando il nostro QRNG per generare il valore di un bit classico, lo codifichiamo su un nuovo qubit e quindi lo inviamo a Eve. Lei può misurarlo e registrare il risultato classico della misurazione.

Ora, per cambiare il qubit da $|0\rangle$ a $|1\rangle$, abbiamo bisogno di avere un'altra operazione quantistica nella nostra "cassetta degli attrezzi". Nel Passo 2, possiamo usare un'operazione di NOT quantistico – che è simile a una classica operazione NOT – che ruota il qubit da $|0\rangle$ a $|1\rangle$ (vedi Figura 2.9).

Chiamiamo questa operazione di NOT quantistico con il nome di operazione x.

DEFINIZIONE

L'operazione x o NOT quantistico trasforma un qubit dallo stato $|0\rangle$ allo stato $|1\rangle$ e viceversa.

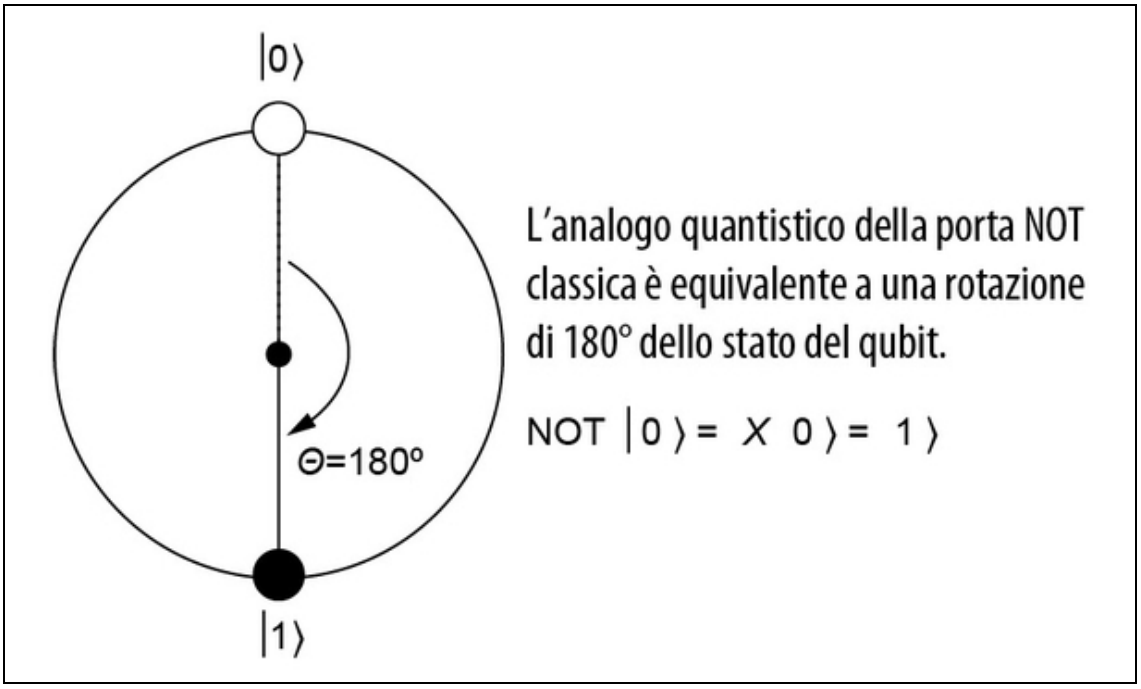


Figura 3.5 Una rappresentazione dell'equivalente quantistico di un NOT operante su un qubit nello stato |0⟩, per lasciarlo nello stato |1⟩.

Il Passo 2 può essere riscritto come segue:

- se il nostro bit classico del Passo 1 era 0, non fare nulla. Se era 1, applica al qubit un'operazione di NOT quantistico (nota anche come operazione $_X$).

Questo algoritmo funziona il 100% delle volte, perché quando Eve misura il qubit che riceve, gli stati |0⟩ e |1⟩ possono essere perfettamente distinti con una misurazione sull'asse *Z*. Può sembrare che noi ed Eve abbiamo fatto fin troppo lavoro solo per condividere solo alcuni bit casuali, ma vedremo come l'aggiunta di alcuni comportamenti quantistici a questo protocollo di base lo renderà molto più utile. Vediamo ora come potremmo implementare questo comportamento nel codice.

Listato 3.1 qkd.py: scambio di bit classico tramite qubit.

```
def prepare_classical_message(bit: bool, q: Qubit) -> None: (1)
    if bit:
        q.x() (2)
def eve_measure(q: Qubit) -> bool:
    return q.measure() (3)
def send_classical_bit(device: QuantumDevice, bit: bool) -> None:
    with device.using_qubit() as q:
        prepare_classical_message(bit, q)
        result = eve_measure(q)
        q.reset()

    assert result == bit (4)
```

(1) Per preparare il nostro qubit con il bit classico che vogliamo inviare, abbiamo bisogno di avere come input il valore del bit e un qubit da utilizzare. Questa funzione non restituisce nulla, perché le conseguenze delle operazioni che applichiamo al qubit sono monitorate nel simulatore del singolo qubit.

(2) Se stiamo inviando un 1, possiamo usare l'operazione NOT `x` per preparare q nello stato $|1\rangle$, perché tale operazione ruoterà $|0\rangle$ in $|1\rangle$ e viceversa.

(3) Sembra sciocco separare la misurazione in un'altra funzione, dato che è di un'unica riga. Ma in futuro cambieremo il modo in cui Eve misura il qubit, quindi questa è una configurazione utile.

(4) Possiamo verificare che la misurazione di q dia lo stesso bit classico che abbiamo inviato.

Il simulatore che abbiamo scritto nel capitolo precedente ha *quasi* ciò di cui abbiamo bisogno per implementarlo. Dobbiamo solo aggiungere un'istruzione corrispondente all'operazione `x`. L'istruzione `x` può essere rappresentata con una matrice X, così come abbiamo rappresentato l'istruzione `h` utilizzando la matrice H. Così come nel Capitolo 2 abbiamo scritto H, possiamo scrivere la matrice X come segue:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

ESERCIZIO 3.1 TABELLE DI VERITÀ E MATRICI

Nel Capitolo 2, abbiamo visto che nel quantum computing le matrici unitarie svolgono lo stesso ruolo delle tabelle di verità nel calcolo classico. Possiamo usare questo fatto per capire come deve essere la matrice X per rappresentare l'operazione di NOT quantistico, `x`. Cominciamo creando una tabella di ciò che la matrice X deve fare a ciascuno stato di input per rappresentare ciò che fa l'istruzione `x`.

Input	Output
$\|0\rangle$	$\|1\rangle$
$\|1\rangle$	$\|0\rangle$

Questa tabella ci dice che se moltiplichiamo la matrice X per il vettore $|0\rangle$, dobbiamo ottenere $|1\rangle$ e, allo stesso modo, $X|1\rangle = |0\rangle$.
Usando NumPy o manualmente, controllate che la matrice

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

corrisponda a ciò che abbiamo scritto nella tabella di verità.

SOLUZIONI DEGLI ESERCIZI

tutte le soluzioni degli esercizi di questo libro si trovano nel suo archivio di codiceall'indirizzo https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Aprite la cartella del capitolo in cui vi trovate e poi il notebook Jupyter con il nome che menziona le soluzioni degli esercizi.

Andiamo avanti e aggiungiamo al nostro simulatore le funzionalità di cui abbiamo bisogno per eseguire il Listato 3.1. Lavoreremo con il

simulatore che abbiamo scritto nel capitolo precedente, ma se avete bisogno di un ripasso, potete trovare il codice nell'archivio GitHub di questo libro: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Innanzitutto, dobbiamo aggiornare l'interfaccia del nostro dispositivo quantistico aggiungendo un nuovo metodo al nostro qubit.

Listato 3.2 interface.py: aggiunta di x all'interfaccia del qubit.

```
class Qubit(metaclass=ABCMeta):
    @abstractmethod
    def h(self): pass
    @abstractmethod
    def x(self): pass (1)
    @abstractmethod
    def measure(self) -> bool: pass
    @abstractmethod

    def reset(self): pass
```

(1) Possiamo modellare l'implementazione dell'operazione di NOT quantistico sulla base dell'operazione `h` del Capitolo 1.

Ora che la nostra interfaccia del qubit sa che vogliamo un'implementazione dell'operazione `x`, aggiungiamo tale implementazione.

Listato 3.3 simulator.py: aggiunta di x al simulatore di qubit.

```
KET_0 = np.array([
    [1],
    [0]
], dtype=complex)
H = np.array([
    [1, 1],
    [1, -1]
], dtype=complex) / np.sqrt(2)
X = np.array([ (1)
    [0, 1],
    [1, 0]
], dtype=complex) / np.sqrt(2)
class SimulatedQubit(Qubit):
    def __init__(self):
        self.reset()
    def h(self):
        self.state = H @ self.state
    def x(self): (2)
        self.state = X @ self.state
```

```
    def measure(self) -> bool:
        pr0 = np.abs(self.state[0, 0]) ** 2
        sample = np.random.random() <= pr0
        return bool(0 if sample else 1)
    def reset(self):
        self.state = KET_0.copy()
```

(1) Aggiunge una variabile `x` per memorizzare la matrice X di cui abbiamo bisogno per rappresentare l'operazione `x`.

(2) Proprio come la funzione `h`, vogliamo implementare l'operazione quantistica `x` applicando al vettore di stato la matrice memorizzata in `x`.

Condivisione di bit classici tramite qubit

Proviamo ora a utilizzare il nostro simulatore di qubit Python per condividere un bit classico segreto tramite un qubit.

> **NOTA**
>
> Nel codice che scriveremo per questo capitolo, i qubit condivisi da noi ed Eve risiedono nello stesso dispositivo simulato. Potrebbe sembrare strano pensare di *inviarci* qubit utilizzando uno stesso computer, ma in realtà, i nostri dispositivi utilizzerebbero *fotoni* (singole particelle di luce) come qubit, che sono davvero facili da inviare su fibra ottica o attraverso telescopi.

Questo non è ancora davvero la stessa cosa di un protocollo per la distribuzione quantistica delle chiavi, ma è una buona base per i tipi di funzioni e passaggi di cui dovrà essere dotato il nostro protocollo QKD.

Aprite la sessione IPython in cui conservate il codice del simulatore eseguendo `ipython` nel Terminale. Dopo aver importato i file Python, create un'istanza del simulatore mono-qubit e generate un bit casuale da utilizzare come bit classico da inviare (ora potete farlo perché avete un generatore quantistico di numeri casuali). Usando un nuovo qubit, preparatelo in base al valore del bit classico che volete inviare a Eve. Eve, da parte sua, misurerà il qubit e a questo punto potrete vedere se entrambi avete lo stesso valore in termini di bit classici.

Listato 3.4 Invio di bit classici con un simulatore mono-qubit.

```
>>> qrng_simulator = SingleQubitSimulator() (1)
>>> key_bit = int(qrng(qrng_simulator)) (2)
>>> qkd_simulator = SingleQubitSimulator() (3)
>>> with qkd_simulator.using_qubit() as q:
...     prepare_classical_message(key_bit, q) (4)
...     print(f"You prepared the classical key bit: {key_bit}")
...     eve_measurement = int(eve_measure(q)) (5)
...     print(f"Eve measured the classical key bit: {eve_measurement}")
...
You prepared the classical key bit: 1

Eve measured the classical key bit: 1
```

(1) Abbiamo bisogno di un qubit simulato da utilizzare per il nostro QRNG.

(2) Reimpiegando la funzione `qrng` che abbiamo scritto nel Capitolo 2, possiamo generare un bit classico casuale da usare per la nostra chiave.

(3) Qui useremo una nuova istanza del simulatore di qubit per lo scambio delle chiavi, ma in realtà non ne avremmo bisogno. Vedremo nel Capitolo 4 come espandere il simulatore per lavorare con più qubit.

(4) Codifichiamo il nostro bit classico nel qubit fornito da `qkd_simulator`. Se il bit classico era uno `0`, non faremo nulla a `qkd_simulator`; se il bit classico era un 1, usiamo il metodo `x` per cambiare il qubit nello stato $|1\rangle$.

(5) Eve misura il qubit ottenuto da `qkd_simulator` e memorizza il suo valore come `eve_measurement`.

Il nostro esempio di condivisione di qubit dovrebbe essere deterministico, vale a dire che ogni volta che prepareremo e invieremo un bit, Eve misurerà correttamente lo stesso valore. È sicuro? Se sospettate che non sia sicuro, avete ragione. Nel prossimo paragrafo, discuteremo della sicurezza del nostro schema prototipo di condivisione dei segreti e cercheremo di migliorarlo.

La storia di due basi

Noi ed Eve ora abbiamo un modo per inviarci bit classici usando qubit, ma che cosa succederebbe se un estraneo dovesse impossessarsi di quel qubit? Potrebbe usare l'istruzione di misurazione per ottenere gli stessi dati classici di Eve. Questo è un problema enorme e ci farebbe ragionevolmente chiedere perché usare i qubit per condividere le chiavi.

Fortunatamente, la meccanica quantistica ci offre un modo per rendere più sicuro questo scambio! Quali modifiche potremmo apportare al protocollo? Per esempio, potremmo rappresentare un messaggio classico "0" con un qubit nello stato |+⟩ e un messaggio "1" con un qubit nello stato |-⟩.

Listato 3.5 qkd.py: codifica del messaggio con gli |+⟩ e |-⟩.

```
def prepare_classical_message_plusminus(bit: bool, q: Qubit) -> None:
    if bit:
        q.x()
    q.h() (1)
def eve_measure_plusminus(q: Qubit) -> bool:
    q.h() (2)
    return q.measure()
def send_classical_bit_plusminus(device: QuantumDevice, bit: bool) ->
None:>    with device.using_qubit() as q:
        prepare_classical_message_plusminus(bit, q)
        result = eve_measure_plusminus(q)
        assert result == bi
```

(1) Tutto ciò che precede questa riga di `prepare_classical_message_plusminus` è uguale a prima, con `prepare_classical_message`. Applicando a questo punto la porta Hadamard ruotiamo gli stati |0⟩ e |1⟩ negli stati |+⟩ e |-⟩.

(2) Usa l'operazione `h` per ruotare gli stati |+⟩ e |-⟩ negli stati |0⟩ e |1⟩ perché la nostra operazione di misurazione è definita per misurare correttamente solo gli stati |0⟩ e |1⟩.

SUGGERIMENTO

Un altro modo di immaginare la misurazione nel Listato 3.5 è che stiamo ruotando la misurazione in modo che corrisponda alla base su cui stiamo attualmente lavorando (|+⟩ / |-⟩). È tutta una questione di punto di vista.

Ora abbiamo due modi diversi di scambiare qubit fra noi ed Eve (vedi la Tabella 3.1). Chiamiamo questi due diversi modi di scambiarsi messaggi *basi*: ognuna di esse contiene due stati ortogonali, perfettamente distinguibili. Nell'Appendice C consideriamo i punti cardinali (per esempio Nord e Ovest) per definire una comoda base per descrivere le direzioni.

Tabella 3.1 Diversi messaggi classici che vogliamo inviare e relativa codifica nelle basi Z e X.

	Messaggio "0"	**Messaggio "1"**
Base "0" (o *Z*)	$\|0\rangle$	$\|1\rangle = X\|0\rangle$
Base "1" (o *X*)	$\|+\rangle = H\|0\rangle$	$\|-\rangle = H\|1\rangle = HX\|0\rangle$

SUGGERIMENTO

Per un ripasso sulle basi, consultate l'Appendice C.

Abbiamo usato gli stati $|0\rangle$ e $|1\rangle$ come una base (chiamata base *Z*) e $|+\rangle$ e $|-\rangle$ come un'altra (chiamata base *X*). I nomi di queste basi si riferiscono all'asse lungo il quale possiamo distinguere perfettamente gli stati (vedi Figura 3.6).

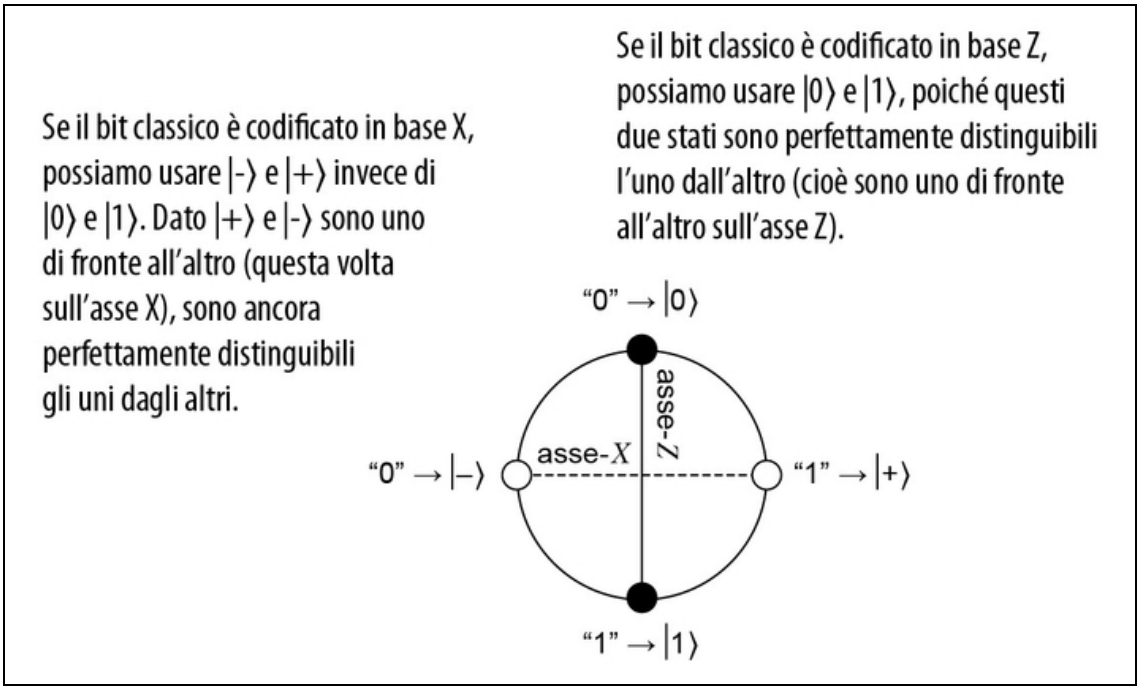


Figura 3.6 Ora, oltre a utilizzare la base Z per codificare un bit classico su un qubit, possiamo usare la base X.

NOTA

Nel quantum computing, non esiste mai una *base corretta*, ma solo basi comode che scegliamo di usare per convenzione.

Se né noi né Eve sappiamo quale modo stiamo usando per inviare un determinato bit, abbiamo entrambi un problema. Che cosa succede se mescoliamo i nostri messaggi in base *Z* e in base *X*? Buone notizie: possiamo usare il nostro simulatore per provare e vedere che cosa succede.

Listato 3.6 Scambiare bit ma senza usare la stessa base.

```
def prepare_classical_message(bit: bool, q: Qubit) -> None: (1)
    if bit:
        q.x()
def eve_measure_plusminus(q: Qubit) -> bool:
    q.h() (2)
    return q.measure()
def prepare_classical_message(bit: bool, q: Qubit) -> None: (1)
    if bit:
        q.x()
```

```
def eve_measure_plusminus(q: Qubit) -> bool:
    q.h() (2)
    return q.measure()
def send_classical_bit_wrong_basis(device: QuantumDevice, bit: bool) ->
None:
    with device.using_qubit() as q:
        prepare_classical_message(bit, q)
        result = eve_measure_plusminus(q)

        assert result == bit, "Two parties do not have the same bit
value" (3)
```

(1) Utilizza il metodo che abbiamo già visto per preparare il nostro qubit in base *Z* utilizzando il metodo `h`.

(2) Eve misura in base *X*, perché applica una porta Hadamard sul suo qubit prima di misurarlo.

(3) La funzione non restituisce nulla, quindi se noi ed Eve otteniamo bit chiave che non corrispondono, verrà generato un errore.

Eseguendo il codice precedente, possiamo vedere che se noi inviamo i dati in base *Z* ed Eve li misura in base *X*, potremmo non ritrovarci con bit classici differenti.

Listato 3.7 Invio in base Z; misurazione in base X.

```
>>> qsim = SingleQubitSimulator()
>>> send_classical_bit_wrong_basis(qsim, 0) (1)

AssertionError: Two parties do not have the same bit value
```

(1) Abbiamo scelto come valore del nostro bit 0. Potrebbe essere necessario eseguire questa riga più volte prima di ottenere l'errore.

Potete verificarlo sperimentalmente. Otterrete `AssertionError` (scambio delle chiavi non riuscito) circa la metà delle volte. Perché? Per cominciare, Eve sta misurando in base *X*, quindi può distinguere perfettamente solo gli stati $|+\rangle$ e $|-\rangle$. Che cosa misurerà se le viene dato uno stato non perfettamente distinguibile nella sua base (come in questo caso, in cui riceve un $|0\rangle$)? Possiamo scrivere lo stato $|0\rangle$ nella base *X* come

$$|0\rangle = (|+\rangle+|-\rangle)/\sqrt{2}$$

Nel Capitolo 2 abbiamo definito $|+\rangle$ in modo simile, sommando $|0\rangle$ e $|1\rangle$. Lo stato $|+\rangle$ è anche chiamato *sovrapposizione* degli stati $|0\rangle$ e $|1\rangle$.

NOTA

Ogni volta che uno stato può essere scritto come una combinazione lineare di stati, come in questo caso, viene considerato una sovrapposizione degli stati che vengono sommati.

ESERCIZIO 3.2 VERIFICARE CHE $|0\rangle$ È UNA SOVRAPPOSIZIONE DI $|+\rangle$ E $|-\rangle$

Provate a usare ciò che avete imparato sui vettori nel capitolo precedente per verificare, a mano o usando Python, che $|0\rangle = (|+\rangle + |-\rangle) / \sqrt{2}$. Suggerimento: ricordate che $|+\rangle = (|0\rangle + |1\rangle) / \sqrt{2}$ e che $|-\rangle = (|0\rangle - |1\rangle) / \sqrt{2}$.

Ora per calcolare la misurazione effettiva con la regola di Born (Capitolo 2), ricordate che possiamo calcolare la probabilità del risultato di una misurazione valutando un determinato stato con la seguente espressione:

$$\Pr(\text{misurazione}|\text{stato}) = |\langle \text{misurazione} \mid \text{stato} \rangle|^2$$

Scrivendo la misurazione dello stato $|0\rangle$ in base X, possiamo vedere che otterremo 0 (o $|+\rangle$) metà delle volte e 1 (o $|-\rangle$) l'altra metà:

$$\Pr(\langle +|0\rangle) = |\langle +|0\rangle|^2 = ((\langle +|+\rangle) + \langle +|-\rangle)/\sqrt{2})^2 = (1 + 0)^2/2 = 1/2$$

ESERCIZIO 3.3 MISURAZIONE DEI QUBIT SU BASI DIFFERENTI

Sulla base dell'esempio precedente:

- calcolare la probabilità di ottenere il risultato $|-\rangle$ misurando lo stato $|0\rangle$ nella direzione$|-\rangle$;
- calcolare anche la probabilità di ottenere dalla misurazione il risultato $|-\rangle$ se lo stato di input è $|1\rangle$.

Questo ci dice che se Eve non conosce la base corretta in cui eseguire la misurazione, allora le sue misurazioni decadono al livello di ipotesi casuali. Questo perché, nella base sbagliata, il qubit è in una sovrapposizione di due stati che definiscono la base. Una "chiave" per il funzionamento del QKD è che senza le corrette informazioni

aggiuntive (la base in cui è codificato il qubit), qualsiasi misurazione di un qubit è praticamente inutile. Per garantire la sicurezza, dobbiamo rendere difficile per un malintenzionato apprendere quelle informazioni aggiuntive necessarie per conoscere la corretta base in cui eseguire le misurazioni. Il protocollo QKD che esamineremo più avanti ha una soluzione per questo problema e anche una dimostrazione (che qui non vedremo) che descrive la possibilità che l'aggressore abbia informazioni sulla chiave.

Distribuzione quantistica delle chiavi: BB84

Abbiamo visto come condividere le chiavi in due basi differenti e che cosa succede se noi ed Eve non usiamo la stessa base. Ancora una volta, potreste chiedervi perché utilizziamo questo approccio per rendere più sicura la condivisione delle nostre chiavi segrete. Esiste un'ampia varietà di protocolli QKD, ciascuno con vantaggi e casi d'uso specifici. Il protocollo QKD *più comune* è BB84, dal nome di una codifica a dire il vero non molto criptica delle iniziali dei due autori e dell'anno in cui il protocollo è stato pubblicato (Bennet e Brassard, 1984).

BB84 è molto simile a ciò che abbiamo impiegato finora per condividere le chiavi, ma ha una differenza fondamentale nel modo in cui noi ed Eve scegliamo le nostre basi. In BB84, entrambe le parti scelgono la propria base in modo casuale (e indipendente), il che significa che finiranno per utilizzare la stessa base il 50% delle volte. La Figura 3.7 mostra le varie fasi del protocollo BB84.

Come conseguenza della scelta casuale delle basi, noi ed Eve dobbiamo anche comunicare su canali classici autenticati (come Internet) per prendere la nostra chiave e trasformarla in una chiave che

riteniamo identica a quella del nostro partner. Questo perché questa è la vita e quando vengono scambiati dei qubit, sia l'ambiente sia persone estranee potranno manipolare o modificare lo stato dei qubit.

Espansione della chiave

Abbiamo sorvolato su un dettaglio nella nostra descrizione del canale di comunicazione classico che noi ed Eve stiamo usando: deve essere autenticato. Cioè, quando scambiamo messaggi classici con Eve via BB84, non c'è alcun problema se qualcun altro può leggerli, ma dobbiamo assicurarci che sia davvero Eve la persona con cui stiamo dialogando. Per dimostrare che qualcuno ha scritto e inviato un determinato messaggio, in realtà abbiamo già bisogno di una qualche forma di segreto condiviso che possiamo usare per convalidare l'identità dell'altro. Quindi, in BB84, dobbiamo già condividere un segreto con l'altra persona. Questo segreto può essere più piccolo del messaggio che stiamo cercando di inviare, quindi BB84 è tecnicamente più un protocollo di *espansione della chiave*.

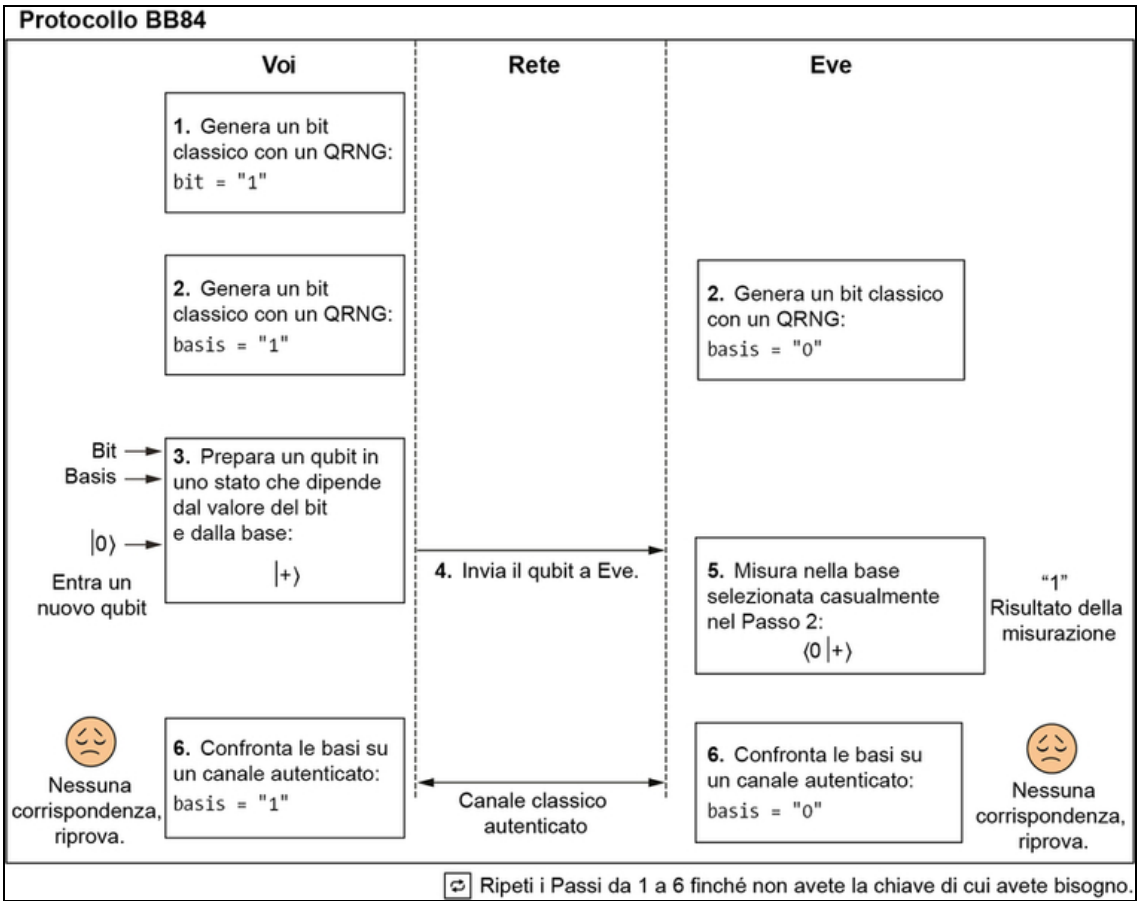


Figura 3.7 I passaggi nel protocollo BB84, una versione particolare di un protocollo QKD.

I passaggi del protocollo BB84 sono i seguenti.

1. Scegli un messaggio casuale di un bit da inviare, campionando il nostro QRNG.
2. Noi ed Eve scegliamo ciascuno una base casuale con il nostro rispettivo QRNG (senza nessuna comunicazione).
3. Prepara un qubit nella base selezionata casualmente che rappresenti il nostro messaggio selezionato casualmente (vedi Tabella 3.2).
4. Invia a Eve il qubit così preparato, attraverso il canale quantistico.
5. Eve misura il qubit al suo arrivo, eseguendo la misurazione nella base da lei selezionata casualmente e registra il risultato come bit classico.
6. Comunica con Eve su un canale classico autenticato e condividi le basi che usate per preparare e misurare il qubit. Se corrispondono, mantieni il bit e aggiungilo alla chiave. Ripeti i Passi da 1 a 6 finché non abbiamo la chiave di cui abbiamo bisogno.

Tabella 3.2 Quale stato dovremmo inviare per ciascun messaggio casuale e scelta della base.

	Messaggio "0"	**Messaggio "1"**
Base "0" (o Z)	$\|0\rangle$	$\|1\rangle = X\|0\rangle$
Base "1" (o X)	$\|+\rangle$	$= H\|0\rangle$ $\|-\rangle = H\|1\rangle = HX\|0\rangle$

Un mondo senza errori

Poiché stiamo simulando il protocollo BB84, sappiamo che il qubit ricevuto da Eve sarà esattamente identito a quello che abbiamo inviato. BB84, più realisticamente, verrà eseguito su gruppi di dati, in cui prima vengono scambiati n qubit, seguiti da una fase di condivisione dei valori della base (con correzione degli errori). Alla fine, dobbiamo ridurre ulteriormente la chiave tramite algoritmi di amplificazione

della privacy, per tenere conto del fatto che un estraneo potrebbe aver ottenuto informazioni parziali dagli errori che abbiamo rilevato. Nella nostra implementazione di BB84, per semplificare le cose abbiamo omesso questi passaggi, che però *sono* fondamentali per la sicurezza, nel mondo reale 😉.

Implementiamo ora il protocollo QKD BB84 in Python. Inizieremo scrivendo una funzione che eseguirà il protocollo BB84 (supponendo di poter contare su una trasmissione senza perdita di dati) per la trasmissione di un singolo bit. Ciò non garantisce che otterremo da questa esecuzione un bit della chiave. Tuttavia, se noi ed Eve scegliamo basi differenti, questo scambio dovrà essere buttato via.

Innanzitutto, è utile predisporre alcune funzioni che semplificheranno il modo in cui scriviamo il protocollo BB84, completo. Noi ed Eve dobbiamo svolgere operazioni come campionare bit casuali e preparare e misurare il qubit del messaggio, che qui abbiamo separato, per maggiore chiarezza.

Listato 3.8 bb84.py: funzioni di supporto prima dello scambio delle chiavi.

```
def sample_random_bit(device: QuantumDevice) -> bool:
    with device.using_qubit() as q:
        q.h()
        result = q.measure()
        q.reset() (1)
    return result
def prepare_message_qubit(message: bool, basis: bool, q: Qubit) -> None:
(2)
    if message:
        q.x()
    if basis:
        q.h()
def measure_message_qubit(basis: bool, q: Qubit) -> bool:
    if basis:
        q.h()
    result = q.measure()
    q.reset() (3)
    return result
def convert_to_hex(bits: List[bool]) -> str: (4)
    return hex(int(
        "".join(["1" if bit else "0" for bit in bits]),
        2

    ))
```

(1) `sample_random_bit` è quasi identica alla nostra precedente funzione `qrng`, tranne per il fatto che qui eseguiremo il reset del qubit dopo la misurazione, poiché sappiamo che vogliamo essere in grado di usarlo più volte.
(2) Il qubit è codificato con il valore del bit della chiave nella base selezionata casualmente.
(3) Come in `sample_random_bit`, dopo che Eve misura il qubit del messaggio, dovrebbe eseguirne il reset, perché nel simulatore lo riutilizzeremo per il prossimo scambio.
(4) Per aiutare a condensare la rappresentazione di lunghe chiavi binarie, una funzione di supporto converte la rappresentazione in una stringa esadecimale, più corta.

Listato 3.9 bb84.py: protocollo BB84 per inviare un bit classico.

```
def send_single_bit_with_bb84(
    your_device: QuantumDevice,
    eve_device: QuantumDevice
    ) -> tuple:
    [your_message, your_basis] = [
        sample_random_bit(your_device) for _ in range(2) (1)
    ]
    eve_basis = sample_random_bit(eve_device) (2)
    with your_device.using_qubit() as q:
        prepare_message_qubit(your_message, your_basis, q) (3)
        # QUBIT SENDING... (4)
        eve_result = measure_message_qubit(eve_basis, q) (5)

    return ((your_message, your_basis), (eve_result, eve_basis)) (6)
```

(1) Possiamo scegliere casualmente il valore di un bit e la base usando il nostro QRNG modificato: qui, la funzione è `sample_random_bit`.
(2) Eve deve scegliere casualmente una base per il proprio qubit, motivo per cui utilizza un `QuantumDevice` distinto.
(3) Dopo le attività preliminari, prepara il qubit da inviare a Eve.
(4) Poiché tutto il calcolo si svolge all'interno di un simulatore sul nostro computer, non dobbiamo fare nulla per "inviare" il qubit a Eve.

(5) Ora Eve ha il nostro qubit e lo misura nella base selezionata casualmente che ha scelto in precedenza.
(6) Restituisce i valori dei bit della chiave e le basi che noi ed Eve avremo alla fine di questo passaggio.

Lo scambio di un bit classico della chiave non sarà sufficiente per inviare un'intera chiave, quindi ora è necessario utilizzare la tecnica precedente per inviare più bit.

Listato 3.10 bb84.py: protocollo BB84 per lo scambio di una chiave con Eve.

```
def simulate_bb84(n_bits: int) -> tuple:
    your_device = SingleQubitSimulator()
    eve_device = SingleQubitSimulator()
    key = []
    n_rounds = 0
    while len(key) < n_bits:
        n_rounds += 1
        ((your_message, your_basis), (eve_result, eve_basis)) =
            ➥ send_single_bit_with_bb84(your_device, eve_device)
        if your_basis == eve_basis: (1)
            assert your_message == eve_result
            key.append(your_message)
    print(f"Took {n_rounds} rounds to generate a {n_bits}-bit key.")

    return key
```

(1) A questo punto, noi ed Eve possiamo annunciare in modo pubblico le basi che ognuno di noi ha usato per misurare questo bit. Se tutto ha funzionato, i nostri risultati dovrebbero concordare ogni volta che le nostre basi concordano. Lo controlliamo qui con un `assert`.

Ora la chiave è acquisita, quindi possiamo procedere e utilizzare la chiave e l'algoritmo di crittografia a pad monouso per inviare il nostro messaggio segreto.

Qubit e non-clonazione

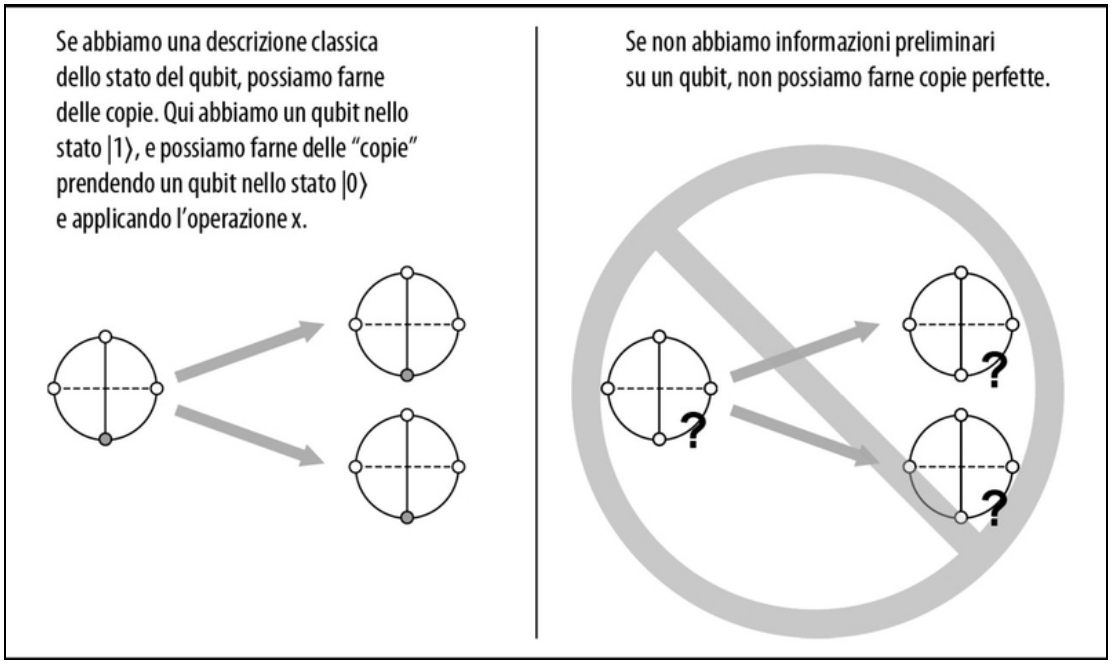
Da quello che abbiamo visto finora, sembrerebbe che un estraneo potrebbe "origliare" i qubit nel canale quantistico e farne delle copie. Ecco come: l'estraneo (che qui si chiama Bob) dovrebbe (senza farsi scoprire)…

1. Copiare i qubit che noi inviamo a Eve, e memorizzarli.

2. Mentre noi ed Eve eseguiamo la parte “classica” del protocollo, ascoltare le basi che entrambi annunciamo e scoprire quelle che abbiamo scelto entrambi.
3. Per i qubit corrispondenti ai bit in cui noi ed Eve usavamo la stessa base, misurare anche le copie dei qubit con la stessa base.

Ed ecco fatto: noi, Eve e Bob avremmo tutti la stessa chiave! Se pensate che questo sia un problema, avete ragione. Ma non preoccupatevi: la meccanica quantistica ha la soluzione. Il problema, per Bob, è nel Passo 1, dove ha bisogno di creare copie *identiche* dei qubit che noi ed Eve ci stiamo scambiando. La buona notizia è che fare una copia esatta di un qubit senza sapere in anticipo di che cosa si tratta è vietato dalla meccanica quantistica. La regola secondo cui i qubit non possono essere copiati in modo identico senza alcuna conoscenza preliminare dello stato è chiamata *teorema di non-clonazione* e afferma quanto segue:

> Nessuna operazione quantistica può copiare perfettamente lo stato di un qubit arbitrario su un altro qubit.



Applicazione del teorema di non-clonazione.

Potremo vedere una semplice dimostrazione di ciò nel prossimo capitolo, una volta che avremo imparato a descrivere lo stato di più di un qubit 😊.

Ecco un modo alternativo per pensare al teorema di non-clonazione: se Bob potesse misurare un qubit senza disturbarlo, potrebbe aggirare la necessità di fare una copia dei qubit che intercetta. Questo non è possibile, perché una volta misurato un qubit, questo "collassa" o cambia, in un modo che Eve potrebbe rilevare come rumore aggiuntivo nei risultati della misurazione dei suoi qubit. Pertanto, la misurazione durante il trasporto non è un'operazione che Bob possa fare senza essere rilevato, quindi le sue intercettazioni sono destinate a fallire.

Utilizzo di una chiave segreta per inviare messaggi segreti

Noi ed Eve abbiamo deciso come utilizzare il protocollo BB84 per condividere una chiave binaria casuale generata da un generatore quantistico di numeri casuali. L'ultimo passaggio consiste nell'usare questa chiave per condividere un messaggio segreto con Eve. Noi ed Eve avevamo precedentemente deciso che il miglior protocollo di crittografia da utilizzare è il pad monouso per inviare i nostri messaggi segreti. Questo risulta essere uno dei protocolli di crittografia più sicuri e, dal momento che condividiamo le chiavi in uno dei modi più sicuri possibili, ha senso mantenere questo standard.

Per esempio, per dire a Eve che ci piace Python, il messaggio che vogliamo inviare è "💝🐍💻". Dal momento che stiamo usando una chiave binaria, dobbiamo convertire la rappresentazione del nostro messaggio Unicode in codice binario, che è il seguente lungo elenco di bit:

```
"1101100000111101 1101110010010110 1101100000111101
 1101110000001101 1101100000111101 1101110010111011"
```

Questa rappresentazione binaria del messaggio è il *testo del nostro messaggio*, e ora vogliamo combinarlo con una chiave per ottenere un testo cifrato che possa essere inviato in sicurezza sulla rete. Una volta che abbiamo la chiave, dal protocollo BB84, (lunga almeno quanto il

nostro messaggio), dobbiamo utilizzare uno schema di crittografia a pad monouso per codificare il nostro messaggio. Abbiamo visto questa tecnica di crittografia nel Capitolo 2; potete usare la Figura 3.8 per un rapido aggiornamento.

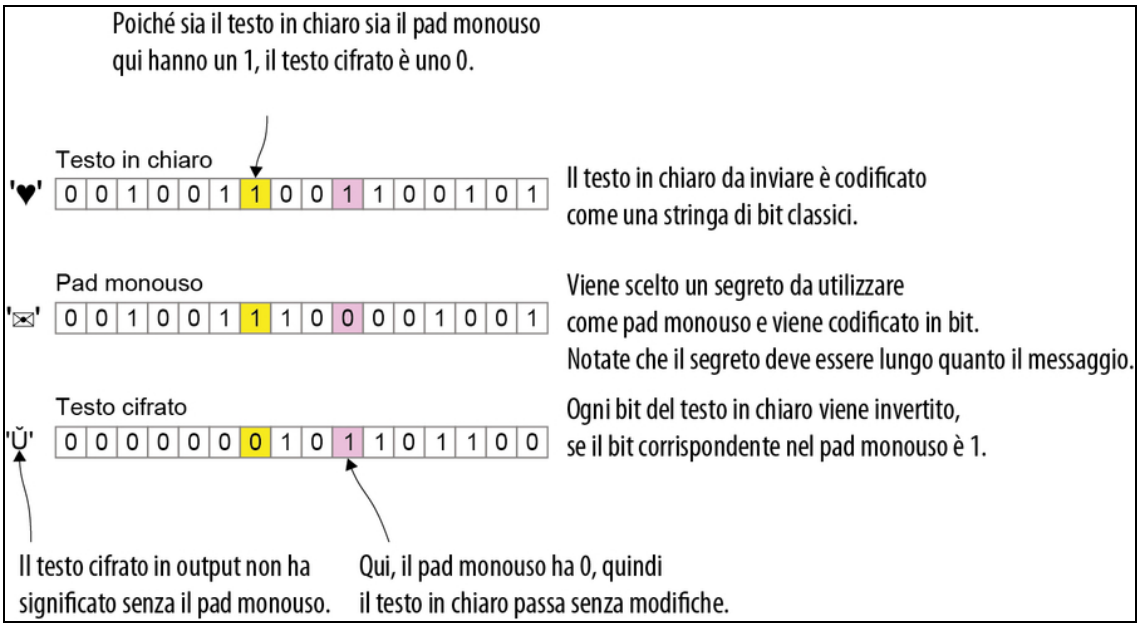


Figura 3.8 Un esempio di crittografia a pad monouso, che utilizza bit casuali per crittografare i messaggi segreti.

Per implementarlo, dobbiamo usare un classico XOR bit-a-bit (l'operatore ^ in Python) per combinare il messaggio e la nostra chiave e creare così il testo cifrato che possiamo inviare in sicurezza a Eve. Per decifrare il nostro messaggio, Eve eseguirà la stessa operazione XOR bit-a-bit con il testo cifrato e la sua chiave (che dovrebbe coincidere con la nostra). Questo le restituirà il messaggio in chiaro, perché quando applichiamo due volte uno XOR a una stringa di bit, otteniamo di nuovo la stringa di bit originale. Ecco il codice in Python.

Listato 3.11 bb84.py: protocollo BB84 per lo scambio di una chiave con Eve.

```
def apply_one_time_pad(message: List[bool], key: List[bool]) ->
List[bool]:
    return [
```

```
        message_bit ^ key_bit (1)
        for (message_bit, key_bit) in zip(message, key)
]
```

(1) L'operatore `^` è uno XOR bit-a-bit in Python. Questo applica un singolo bit della nostra chiave, come pad monouso, al testo del nostro messaggio.

ESERCIZIO 3.4 CRITTOGRAFIA A PAD MONOUSO

Se abbiamo il testo cifrato `10100101` e la chiave `00100110`, quale messaggio è stato inviato, originariamente?

Mettiamo tutto insieme e condividiamo con Eve il messaggio ("💔🐍💻") eseguendo il file `bb84.py` che abbiamo creato.

Listato 3.12 bb84.py: utilizzo di BB84 e della crittografia a pad monouso.

```
if __name__ == "__main__":
    print("Generating a 96-bit key by simulating BB84...")
    key = simulate_bb84(96)
    print(f"Got key:                              {convert_to_hex(key)}.")
    message = [
        1, 1, 0, 1, 1, 0, 0, 0,
        0, 0, 1, 1, 1, 1, 0, 1,
        1, 1, 0, 1, 1, 1, 0, 0,
        1, 0, 0, 1, 0, 1, 1, 0,
        1, 1, 0, 1, 1, 0, 0, 0,
        0, 0, 1, 1, 1, 1, 0, 1,
        1, 1, 0, 1, 1, 1, 0, 0,
        0, 0, 0, 0, 1, 1, 0, 1,
        1, 1, 0, 1, 1, 0, 0, 0,
        0, 0, 1, 1, 1, 1, 0, 1,
        1, 1, 0, 1, 1, 1, 0, 0,
        1, 0, 1, 1, 1, 0, 1, 1
    ]
    print(f"Using key to send secret message:
{convert_to_hex(message)}.")
    encrypted_message = apply_one_time_pad(message, key)
    print(f"Encrypted message:
{convert_to_hex(encrypted_message)}.")
    decrypted_message = apply_one_time_pad(encrypted_message, key)

    print(f"Eve decrypted to get:
{convert_to_hex(decrypted_message)}.")
```

Listato 3.13 Esecuzione della soluzione completa per lo scenario presentato in capitolo.

```
$ python bb84.py
Generating a 96-bit key by simulating BB84...
Took 170 rounds to generate a 96-bit key. (1)
Got key:                          0xb35e061b873f799c61ad8fad. (2)
Using key to send secret message: 0xd83ddc96d83ddc0dd83ddcbb. (3)
Encrypted message:                0x6b63da8d5f02a591b9905316. (4)

Eve decrypted to get:             0xd83ddc96d83ddc0dd83ddcbb. (5)
```

(1) Poiché la nostra base e quella di Eve saranno la stessa circa la metà delle volte, dovrebbero essere necessari circa due round di BB84 per ogni bit della chiave che vogliamo generare.
(2) La chiave esatta che generiamo sarà diversa ogni volta che lanciamo la simulazione BB84, un elemento fondamentale del protocollo, dopotutto.
(3) Il messaggio che otteniamo scrivendo ciascuno dei codici Unicode di "🐭🐍💻".
(4) Quando combiniamo il nostro messaggio segreto con la chiave che abbiamo ottenuto in precedenza, utilizzando la chiave come pad monouso, il nostro messaggio viene crittografato.
(5) Quando Eve usa la stessa chiave, ottiene il nostro messaggio segreto originale.

La distribuzione quantistica delle chiavi è una delle tecnologie derivate più importanti del quantum computing e ha potenzialmente un impatto enorme sulla nostra infrastruttura di sicurezza. Sebbene attualmente sia abbastanza facile configurare QKD per parti relativamente vicine tra loro (circa 200 km o meno), la distribuzione di un sistema globale per QKD impone sfide molto complesse. Di solito, il sistema fisico utilizzato in QKD è un fotone, ed è difficile inviare singole particelle di luce su lunghe distanze senza perderne il significato.

Ora che abbiamo creato un simulatore mono-qubit e abbiamo programmato alcune applicazioni su questo qubit, siamo pronti per iniziare a utilizzare più qubit. Nel prossimo capitolo prenderemo il simulatore che abbiamo costruito e gli aggiungeremo le funzionalità necessarie per simulare più qubit e anche per provare a fare giochi non locali con Eve 💔.

Riepilogo

- La distribuzione quantistica delle chiavi è un protocollo che ci consente di generare casualmente chiavi condivise che possiamo utilizzare per comunicare in modo sicuro e privato.
- Possiamo misurare i qubit su basi differenti; se misuriamo nella stessa base in cui prepariamo i nostri qubit, i risultati saranno deterministici, mentre se misuriamo in basi differenti, i risultati saranno casuali.
- Il teorema di non-clonazione garantisce che un estraneo non possa indovinare la base corretta per la misurazione senza provocare il fallimento del protocollo di distribuzione delle chiavi.
- Dopo aver utilizzato QKD per condividere una chiave, possiamo utilizzare la chiave con un algoritmo classico chiamato pad monouso per inviare i dati in modo sicuro.

Giochi non locali: gestione di più qubit

Nel capitolo precedente abbiamo usato i qubit per comunicare con Eve in modo sicuro, esplorando come possiamo usare i dispositivi quantistici in crittografia. Se lavorare con un qubit è stato divertente, be' lavorare con più qubit sarà… più divertente! In questo capitolo impareremo a modellare stati di più qubit e scopriremo che cosa significa, per loro, essere *entangled*. Giocheremo di nuovo con Eve, ma questa volta avremo anche bisogno di un arbitro.

Giochi non locali

A questo punto, abbiamo visto come i dispositivi mono-qubit possono essere programmati per svolgere compiti utili come la generazione di numeri casuali e la distribuzione quantistica delle chiavi. Le attività computazionali più entusiasmanti, tuttavia, richiedono l'utilizzo di più qubit. In questo capitolo impiegheremo giochi non locali: un modo per verificare insieme ad amici queste nostre descrizioni della meccanica quantistica dell'universo, utilizzando sistemi multi-qubit.

Che cosa sono i giochi non locali?

Tutti abbiamo giocato a qualche tipo di gioco, sportivo, da tavolo, di ruolo o videogiochi. I giochi sono uno dei modi migliori per esplorare nuovi mondi e per mettere alla prova i nostri limiti in termini di forza, resistenza e comprensione delle cose. Abbiamo scoperto che a Eve piacciono i giochi e l'ultimo messaggio crittografato che abbiamo ricevuto da lei era il seguente:

> Ciao giocatore! Sono ansiosa di giocare a un nuovo gioco che si chiama CHSH. È un gioco non locale che prevede un arbitro. Ti invierò le istruzioni nel prossimo messaggio. STOP

Ciò che rende *non locale* il gioco proposto da Eve è il fatto che i giocatori (purtroppo) non si trovano tutti nello stesso posto, ma partecipano alla partita inviando e ricevendo messaggi con un arbitro centrale, senza la possibilità di parlare tra loro durante il gioco. Ciò che è veramente interessante è che giocando possiamo dimostrare che la fisica classica non si limita a descrivere i risultati che otteniamo in questi giochi impiegando particolari strategie. La strategia vincente che vedremo prevede il fatto che i giocatori si scambino un paio di qubit prima dell'inizio del gioco. Ci immergeremo nel significato dell'*entangling* di due qubit mentre procederemo nel capitolo, ma iniziamo descrivendo le regole del nostro gioco non locale.

> **NOTA**
> L'arbitro che giudica una partita non locale può garantire che i giocatori non comunichino separandoli da una distanza sufficientemente ampia da impedire che la luce di un giocatore possa raggiungere l'altro prima della fine della partita.

Un test di fisica quantistica: il gioco CHSH

Il gioco non locale che Eve ha suggerito di giocare si chiama CHSH, ed è illustrato nella Figura 4.1.

> **NOTA**

Il nome CHSH deriva dalle iniziali dei ricercatori che hanno creato il gioco: Clauser, Horne, Shimony e Holt. Potete trovare l'originale, se siete interessati, in: https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.23.880.

Il gioco CHSH prevede due giocatori e un arbitro. Possiamo giocare tutti i round che vogliamo e ogni round ha tre fasi. Una volta iniziato un round, i giocatori non possono comunicare e devono prendere le proprie decisioni (possibilmente pre-programmate).

I passaggi per un round del gioco CHSH sono i seguenti.

1. L'arbitro inizia il round assegnando a voi e a Eve un bit classico. L'arbitro sceglie questi bit *in modo indipendente* e *casualmente uniforme*, quindi noi potremmo ottenere uno 0 o un 1, ciascuno con il 50% di probabilità, e lo stesso per Eve. Ciò significa che ci sono quattro modi possibili in cui l'arbitro può iniziare il gioco (bit nostro, bit di Eve): (0, 0), (0, 1), (1, 0) o (1, 1).
2. Noi ed Eve dobbiamo decidere ciascuno *in modo indipendente* un singolo bit classico da restituire all'arbitro come risposta.
3. L'arbitro calcola quindi la parità (XOR) delle risposte di bit classici nostra e di Eve.

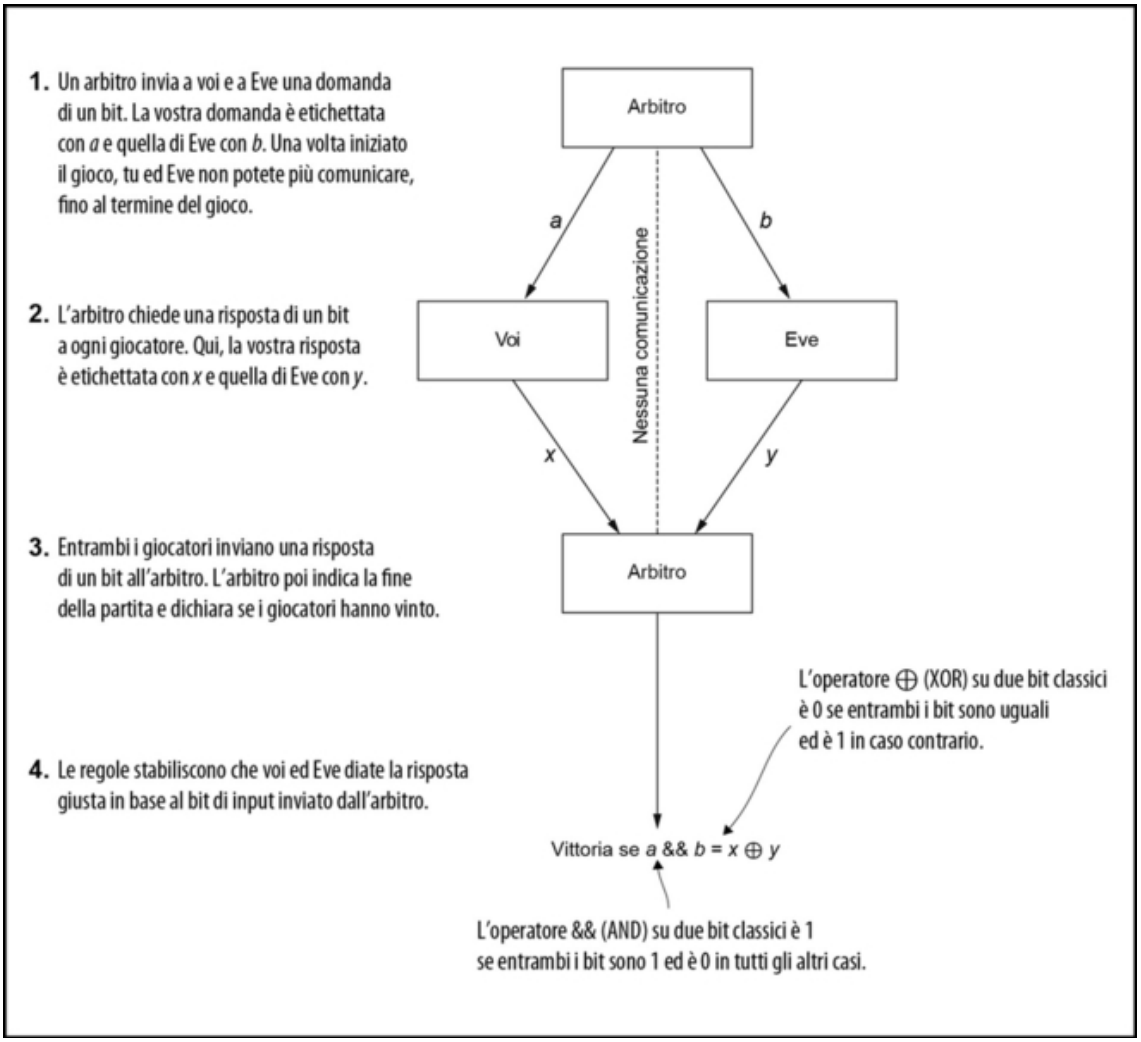


Figura 4.1 CHSH è un gioco non locale che prevede due giocatori e un arbitro. L'arbitro pone a ogni giocatore una domanda sotto forma di un bit; poi ogni giocatore deve rispondere all'arbitro. I giocatori vincono se lo XOR booleano delle loro risposte è uguale all'AND delle domande dell'arbitro.

Come elencato nella Tabella 4.1, in tre casi su quattro, noi ed Eve dobbiamo rispondere con parità *pari* (le nostre risposte devono essere uguali) per vincere, mentre nel quarto caso, le nostre risposte devono essere diverse. Queste sono regole decisamente *insolite*, ma non troppo male rispetto ad alcuni giochi da tavolo che impegnano per più giorni.

Tabella 4.1 Condizioni di vittoria per il gioco CHSH.

Input nostro	Input di Eve	Parità delle risposte per vincere
0	0	Pari
0	1	Pari
1	0	Pari
1	1	Dispari

Possiamo espandere la Tabella 4.1 per ottenere tutti i possibili risultati del gioco; vedi la Tabella 4.2.

Tabella 4.2 Tutti i possibili stati del gioco CHSH con le condizioni di vittoria. I bit di input provengono dall'arbitro ed entrambi i giocatori rispondono all'arbitro.

Input nostro	Input di Eve	Risposta nostra	Risposta di Eve	Parità	Vittoria?
0	0	0	0	Pari	Sì
0	0	0	1	Odd	No
0	0	1	0	Odd	No
0	0	1	1	Pari	Sì
0	1	0	0	Pari	Sì
0	1	0	1	Odd	No
0	1	1	0	Odd	No
0	1	1	1	Pari	Sì
1	0	0	0	Pari	Sì
1	0	0	1	Dispari	No
1	0	1	0	Dispari	No
1	0	1	1	Pari	Sì
1	1	0	0	Pari	No

1	1	0	1	Dispari	Sì
1	1	1	0	Dispari	Sì
1	1	1	1	Pari	No

Vediamo ora il codice Python per simulare questo gioco. Poiché le risposte nostre e di Eve all'arbitro possono dipendere dal messaggio inviatoci dall'arbitro, possiamo rappresentare le azioni di ogni giocatore come una "funzione" richiamata dall'arbitro.

ESERCIZIO 4.1 LO STATO DELL'ARBITRO

Poiché l'arbitro è puramente classico, lo modelleremo utilizzando generatori classici di numeri casuali. Ciò lascia aperta la possibilità, tuttavia, che noi ed Eve potremmo imbrogliare indovinando le domande dell'arbitro. Un possibile miglioramento potrebbe essere quello di utilizzare il generatore quantistico del Capitolo 2. Modificate il codice di esempio del Listato 4.1 in modo che l'arbitro possa porre domande su noi ed Eve misurando un qubit che inizia nello stato $|+\rangle$.

SOLUZIONI DEGLI ESERCIZI

Tutte le soluzioni degli esercizi di questo libro si trovano nel suo archivio di codice: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Aprite la cartella del capitolo in cui vi trovate e poi il notebook Jupyter con il nome che menziona le soluzioni degli esercizi.

Come possiamo vedere nel Listato 4.1, stiamo dichiarando un nuovo tipo `Strategy` per definire la tupla di funzioni che rappresentano le funzioni di un bit nostre e di Eve che rappresentano le nostre strategie individuali. Queste funzioni rappresentano, in pratica, ciò che noi ed Eve facciamo con i bit che ci vengono dati dall'arbitro.

Listato 4.1 chsh.py: implementazione di Python per il gioco CHSH.

```
import random
from functools import partial
from typing import Tuple, Callable
import numpy as np
from interface import QuantumDevice, Qubit
from simulator import Simulator
```

```
Strategy = Tuple[Callable[[int], int], Callable[[int], int]] (1)
def random_bit() -> int:
    return random.randint(0, 1) (2)
def referee(strategy: Callable[[], Strategy]) -> bool:
    you, eve = strategy() (3)
    your_input, eve_input = random_bit(), random_bit() (4)
    parity = 0 if you(your_input) == eve(eve_input) else 1 (5)
    return parity == (your_input and eve_input) (6)
def est_win_probability(strategy: Callable[[], Strategy], (7)
                        n_games: int = 1000) -> float:
    return sum(
        referee(strategy)
        for idx_game in range(n_games)
    ) / n_games (8)
```

(1) L'utilizzo del modulo `typing` di Python ci consente di documentare che un valore di tipo `Strategy` è una tupla di due funzioni, ognuna delle quali accetta un `int` e restituisce un `int`.

(2) Il generatore classico di numeri casuali impiegato dall'arbitro (`referee`).

(3) La funzione `strategy` assegnerà funzioni di un bit che rappresentano ciò che `you` ed `eve` faranno in base al nostro input.

(4) L'arbitro sceglie due bit casuali, uno per ogni giocatore.

(5) Dà a ogni giocatore il suo bit casuale e poi calcola la parità delle loro risposte.

(6) Controllate la Tabella 4.1 per vedere se i giocatori hanno vinto.

(7) Possiamo usare la funzione `sum` di Python per contare il numero di volte in cui l'arbitro restituisce `True` per una determinata strategia, in altre parole, quante volte abbiamo vinto la partita.

(8) Divide per il numero di partite che abbiamo giocato e quindi stima la probabilità che la nostra strategia e quella di Eve vincano la partita.

Notate che nel Listato 4.1, non abbiamo ancora una definizione per la strategia di input dell'arbitro. Ora che abbiamo implementato le regole del gioco in Python, parliamo di *strategia* e arriviamo all'implementazione di una strategia classica per giocare a CHSH.

SUGGERIMENTO

È utile adottare nel nostro codice delle convenzioni di denominazione delle variabili che chiariscano il ruolo da esse svolto. Abbiamo scelto di utilizzare il prefisso `n_` nella variabile `n_games` per indicare che la variabile si riferisce a un numero o a una dimensione, e usiamo il prefisso `idx_` per fare riferimento all'indice per ogni singolo gioco. È sempre una buona cosa se il nostro codice è comprensibile.

Strategia classica

La strategia più semplice, sia per noi sia per Eve, consiste nell'ignorare completamente gli input. Osservando la Tabella 4.3, se entrambi concordiamo prima della partita che non cambieremo mai i nostri risultati (cioè, restituiremo sempre `0`), vinceremo il 75% delle volte (questo presuppone che l'arbitro scelga i bit casuali per ogni giocatore in modo uniforme).

Tabella 4.3 La migliore strategia classica per il gioco CHSH, dove entrambi rispondiamo sempre con 0, e relative condizioni di vittoria.

Input nostro	Input di Eve	Risposta nostra	Risposta di Eve	Parità	Vittoria?
0	0	0	0	Pari	Sì
0	1	0	0	Pari	Sì
1	0	0	0	Pari	Sì
1	1	0	0	Pari	No

Se dovessimo scrivere questa strategia come una funzione Python, avremmo il seguente codice.

Listato 4.2 chsh.py: una strategia facile e costante per il gioco CHSH.

```
def constant_strategy() -> Strategy:
    return (
        lambda your_input: 0,
        lambda eve_input: 0
    )
```

Ora siamo in grado di verificare con quale frequenza ci aspettiamo di vincere una partita di CHSH utilizzando `constant_strategy`:

```
>>> est_win_probability(constant_strategy) (1)
0.771
```

(1) Notate che potreste ottenere un po' più o un po' meno del 75%. Questo perché la probabilità di vittoria è stimata su un numero finito di partite (in statistica, questa è chiamata *distribuzione binomiale*). Per questo esempio, ci aspetteremmo un errore di circa l'1,5%.

OK, questa è una strategia facile, ma possiamo fare qualcosa di più intelligente? Dal momento che noi ed Eve abbiamo solo risorse classiche, purtroppo questo è, *dimostrabilmente*, il meglio che possiamo fare. A meno di barare 😈 (per esempio, comunicando con Eve o cercando di indovinare l'input dell'arbitro), non possiamo vincere questa partita in media più del 75% delle volte.

Tutto questo porta a una domanda ovvia: e se noi ed Eve potessimo usare i qubit? Quale sarebbe la nostra migliore strategia e quanto spesso vinceremmo? Che cosa direbbe, della nostra comprensione dell'universo, se avessimo la prova che non possiamo vincere a CHSH più del 75% circa delle volte, ma poi troviamo un modo per *migliorare* quel tasso di vittorie? Come potete immaginare, possiamo fare meglio di una percentuale di vittorie del 75% a CHSH se i giocatori condividono risorse quantistiche, cioè se hanno qubit. Più avanti nel capitolo, esamineremo queste strategie quantistiche per CHSH, ma… ecco uno "spoiler": dovremo simulare più di un qubit.

Lavorare con più stati di qubit

Finora abbiamo lavorato con un solo qubit per volta. Per un gioco non locale, per esempio, ogni giocatore avrà bisogno del proprio qubit. Ciò solleva una domanda: come cambiano le cose quando il sistema che stiamo considerando ha più di un qubit? La differenza principale è

che non possiamo descrivere ogni singolo qubit e dobbiamo pensare in termini di uno stato che descriva l'intero sistema.

> **NOTA**
>
> Quando si descrive un gruppo, un registro, di qubit, generalmente non si può descrivere semplicemente ogni singolo qubit. I comportamenti quantistici più utili possono essere visti solo quando descriviamo lo stato di un gruppo o registro di qubit.

Il prossimo paragrafo aiuterà a mettere in relazione questa vista a livello di sistema con un concetto di programmazione classica simile: il registro.

I registri

Supponiamo di avere un *registro* di bit classici: cioè una raccolta di più bit classici. Possiamo indicizzare ogni bit di quel registro e controllare il suo valore in modo indipendente, anche se esso continua a far parte di quel registro. Il contenuto del registro può rappresentare un valore più complesso, come i bit che, insieme, rappresentano un carattere Unicode (come abbiamo visto nel Capitolo 3), ma questa interpretazione di alto livello qui non è necessaria.

Quando memorizziamo le informazioni in un registro classico, il numero dei diversi stati di quel registro cresce molto rapidamente man mano che aggiungiamo più bit. Per esempio, se abbiamo tre bit, avremo otto diversi stati in cui può trovarsi il nostro registro (vedi il Listato 4.3 per un esempio). Diciamo, per lo stato `101` di un registro classico, che lo zeresimo bit è un `1`, il primo bit è uno `0` e il secondo bit è un `1`. Quando questi valori sono concatenati insieme, ci danno la stringa `101`.

Listato 4.3 Tutti gli stati di un registro classico di tre bit.

```
>>> from itertools import product
>>> ", ".join(map("".join, product("01", repeat=3)))
```

```
'000, 001, 010, 011, 100, 101, 110, 111'
```

Se abbiamo quattro bit, possiamo memorizzare 16 stati differenti; se abbiamo *n* bit, possiamo memorizzarne 2^n. Diciamo che il numero di stati possibili di un registro classico *cresce esponenzialmente* con il numero dei bit. Le stringhe di bit prodotte dal Listato 4.3 mostrano i dati contenuti nel registro per ogni stato del registro stesso. Fungono anche da utili etichette per uno degli otto possibili messaggi che possiamo codificare con tre bit classici.

Che cosa c'entra tutto questo con i qubit? Abbiamo visto nei Capitoli 2 e 3 che ogni stato di un bit classico descrive anche uno stato di un qubit. Lo stesso vale anche per i registri di qubit. Per esempio, lo stato di tre bit "010" descrive anche lo stato di tre qubit $|010\rangle$.

SUGGERIMENTO

Gli stati dei qubit descritti in questo modo da bit classici sono chiamati *stati base computazionali*; usiamo questo termine anche per gli stati di più qubit descritti da stringhe di bit classici.

Proprio come avviene con i singoli qubit, lo stato di un registro di più qubit può essere composto sommando fra loro diversi stati di qubit. Nello stesso identico modo in cui possiamo scrivere $|+\rangle$ come $(|0\rangle + |1\rangle) / \sqrt{2}$ per ottenere un altro stato valido del qubit, il nostro registro di tre qubit può trovarsi in una grande varietà di stati:

- $(|010\rangle + |111\rangle) / \sqrt{2}$
- $(|001\rangle + |010\rangle + |100\rangle) / \sqrt{3}$

SUGGERIMENTO

Vedremo meglio le cose mentre procederemo con i capitoli, ma proprio come abbiamo bisogno della radice quadrata di 2 per far sì che le probabilità delle misurazioni funzionino per $|+\rangle = (|0\rangle + |1\rangle) / \sqrt{2}$, dobbiamo dividere per $\sqrt{2}$ e per $\sqrt{3}$ nei nostri esempi per assicurarci che tutte le probabilità per tutte le misurazioni siano realistiche: per esempio, che la loro somma sia 1.

Questo esempio di linearità dei registri quantistici è chiamato *principio di sovrapposizione*.

DEFINIZIONE

Il *principio di sovrapposizione* dice che possiamo sommare due diversi stati di un registro quantistico per ottenere un altro stato valido. Lo stato $|+\rangle$ che abbiamo già visto ne è un buon esempio, tranne per il fatto che ogni registro ha un solo qubit.

Per scrivere lo stato di un registro quantistico in un computer, useremo di nuovo i vettori, proprio come abbiamo fatto nel Capitolo 2. La differenza principale è costituita da quanti numeri elenchiamo in ciascun vettore. Vediamo come scrivere lo stato di un registro di due qubit su un computer. Per esempio, il vettore per lo stato di due qubit $(|00\rangle + |11\rangle) / \sqrt{2}$ può anche essere scritto come lo stato $(1 \times |00\rangle + 0 \times |01\rangle + 0 \times |10\rangle + 1 \times |11\rangle) / \sqrt{2}$. Se facciamo un elenco di ciò per cui abbiamo dovuto moltiplicare ogni stato computazionale di base per ottenere lo stato che volevamo, abbiamo esattamente le informazioni che dobbiamo scrivere nel nostro vettore. Nel Listato 4.4, abbiamo scritto $(|00\rangle + |11\rangle) / \sqrt{2}$ come un vettore.

Listato 4.4 Usare Python per scrivere un esempio di uno stato di due qubit.

```
>>> import numpy as np
>>> two_qubit_state = np.array([[ (1)
...
...     1, (2)
...
...     0, (3)
...
...     0,
...     1
... ]]) / np.sqrt(2) (4)
```

(1) Iniziamo allo stesso modo, utilizzando la funzione `np.array` per creare un nuovo vettore.

(2) Ogni elemento di questo vettore descrive uno stato differente della base computazionale. Questo elemento ci dice che dobbiamo moltiplicare $|00\rangle$ per 1.

(3) Allo stesso modo, questo elemento ci dice quanta parte dello stato $|01\rangle$ dobbiamo aggiungere per ottenere lo stato che vogliamo.
(4) Infine, dividiamo per $\sqrt{2}$ per assicurarci che tutte le probabilità di misurazione funzionino, proprio come abbiamo fatto con lo stato $|+\rangle$ nei Capitoli 2 e 3.

I numeri nel vettore del Listato 4.4 sono i coefficienti che moltiplichiamo per ciascuno degli stati computazionali di base, che poi sommiamo fra loro per creare un nuovo stato. Questi coefficienti sono anche chiamati *ampiezze* per ciascuno degli stati di base presenti nella somma.

Pensare in termini di direzioni

Un altro modo di considerare questo esempio è in termini di direzioni su una mappa, così come i vettori vengono trattati nell'Appendice C. Ogni diverso stato computazionale di base ci dice una direzione in cui lo stato di un qubit può essere puntato. Possiamo pensare a uno stato di due qubit come a una direzione su quattro dimensioni invece delle mappe bidimensionali che vediamo nell'Appendice C. Poiché questo libro è, ovviamente, bidimensionale, sfortunatamente non possiamo disegnare un'immagine rappresentativa, ma a volte può essere più utile pensare ai vettori come a direzioni che come a elenchi di numeri.

Perché è difficile simulare i computer quantistici?

Abbiamo visto che all'aumentare del numero di bit, il numero di stati differenti in cui può trovarsi un registro cresce esponenzialmente. Anche se questo non sarà un problema se programmiamo un gioco non locale con due soli qubit, vorremo poter usare più di due qubit mentre procediamo in questo libro.

Quando lo faremo, avremo anche a che fare con un numero esponenziale di stati computazionali di base distinti per il nostro registro quantistico, il che significa che avremo bisogno di impiegare ampiezze esponenziali nei nostri vettori man mano che aumentiamo la

dimensione del nostro registro quantistico. Per scrivere lo stato di un registro di 10 qubit, dovremo usare un vettore che contenga una lista di $2^{10} = 1024$ ampiezze. Per un registro di 20 qubit, avremo bisogno di circa 1 milione di ampiezze nei nostri vettori. Quando arriveremo a 60 qubit, avremo bisogno di circa $1{,}15 \times 10^{18}$ numeri nei nostri vettori. Tale valore corrisponde al numero stimato di granelli di sabbia presenti sull'intero pianeta.

Nella Tabella 4.4 riassumiamo il significato di questa crescita esponenziale mentre proviamo a simulare computer quantistici utilizzando computer classici come telefoni, laptop, cluster e ambienti cloud. Questa tabella mostra che anche se è molto difficile ragionare di computer quantistici impiegando computer classici, possiamo ragionare con una certa facilità su piccoli esempi. Con un computer portatile o desktop, possiamo simulare fino a circa 30 qubit senza troppi problemi. Come vedremo nel resto del libro, questo è più che sufficiente per capire come funzionano i programmi quantistici e come si possono usare i computer quantistici per risolvere problemi interessanti.

Tabella 4.4 Quanta memoria è necessaria per memorizzare uno stato quantistico?

Numero di qubit	Numero di ampiezze	Memoria	Esempio
1	2	128 bit	
2	4	256 bit	
3	8	512 bit	
4	16	1 kb	
8	256	4 KB	Carta di credito contactless
10	1024	16 KB	
20	1.048.576	16 MB	

26	67.108.864	1 GB	RAM del Raspberry Pi
28	268.435.456	4 GB	RAM dell'iPhone XS Max
30	1.073.741.824	16 GB	RAM di un computer portatile o desktop
40	1.099.511.627.776	16 TB	
50	1.125.899.906.842.624	16 PB	
60	1.152.921.504.606.846.976	16 EB	
80	1.208.925.819.614.629.174.706.176	16 YB	Dimensione approssimativa di Internet
410	$2{,}6 \times 10^{123}$	$4{,}2 \times 10^{124}$ byte	Un computer delle dimensioni dell'universo

Approfondimento: i computer quantistici sono esponenzialmente più potenti?
Potreste aver sentito dire che il numero di numeri che dobbiamo gestire per simulare un computer quantistico utilizzando un computer classico è il motivo per cui i computer quantistici sono più potenti o che un computer quantistico può memorizzare così tante informazioni. Questo non è esattamente vero. Il *teorema di Holevo* ci dice che un computer quantistico composto da 410 qubit può memorizzare al massimo 410 bit classici di informazione, anche se ci vorrebbe un computer classico delle dimensioni dell'intero universo per memorizzare lo stato di quel computer quantistico.
In altre parole, *per il solo fatto che è difficile simulare un computer quantistico non significa che faccia qualcosa di utile*. Per tutto il resto del libro, vedremo che non è facile capire come utilizzare un computer quantistico per risolvere problemi utili.

Prodotti tensoriali per la preparazione degli stati

Descrivere i registri quantistici come vettori che descrivono gli stati computazionali di base va benissimo, ma anche se conosciamo lo stato cui vogliamo arrivare, dobbiamo sapere come prepararlo. Per esempio, se un giocatore di un gioco non locale ha un qubit nello stato $|0\rangle$ e l'altro ha un qubit nello stato $|1\rangle$, possiamo combinare con facilità questi due stati mono-qubit per descrivere lo stato del gioco come $|01\rangle$. Che cosa significa "combinare" gli stati di due (o più) qubit? Possiamo farlo aggiungendo un altro concetto alla nostra "cassetta degli attrezzi" matematica: il *prodotto tensoriale*.

Allo stesso modo in cui abbiamo utilizzato la funzione prodotto nel Listato 4.3 per combinare le etichette per un registro di tre bit classici, possiamo usare il concetto di prodotto tensoriale, scritto come $\otimes$, per combinare gli stati quantistici di ogni qubit per creare un stato che descriva più qubit. L'output del prodotto era una lista di tutti i possibili stati di quei tre bit classici. L'output di un prodotto tensoriale è, analogamente, uno stato che elenca tutti gli stati computazionali di base di un registro quantistico. Possiamo usare NumPy per calcolare i prodotti tensoriali, che fornisce un'implementazione del prodotto tensoriale tramite la funzione `np.kron`, come illustrato nel Listato 4.5.

Perché kron?

Il nome `np.kron` può sembrare strano per una funzione che implementa prodotti tensoriali, ma tale nome è l'abbreviazione di un concetto matematico correlato, chiamato *prodotto di Kronecker*. NumPy impiega `kron` come abbreviazione di *Kronecker* secondo la convenzione utilizzata in MATLAB, R, Julia e altre piattaforme di calcolo scientifico.

Listato 4.5 Uno stato di due qubit, utilizzando NumPy e prodotti tensoriali.

```
>>> import numpy as np
>>> ket0 = np.array([[1], [0]]) (1)
>>> ket1 = np.array([[0], [1]])
```

```
>>> np.kron(ket0, ket1) (2)
array([[0],
       [1], (3)
       [0],
       [0]])
```

(1) Definisce i vettori per gli stati mono-qubit $|0\rangle$ e $|1\rangle$, proprio come nei Capitoli 2 e 3.

(2) Possiamo costruire il vettore per $|0\rangle \otimes |1\rangle$ richiamando l'implementazione NumPy del prodotto tensoriale. Per ragioni storiche, il prodotto tensoriale è rappresentato in NumPy dalla funzione `kron`.

(3) Il vettore restituito da `np.kron` ha un `1` per l'elemento corrispondente allo stato computazionale di base $|01\rangle$ e uno 0 ovunque, quindi riconosciamo questo vettore come lo stato $|01\rangle$.

Questo esempio ci mostra che $|0\rangle \otimes |1\rangle = |01\rangle$. Cioè, se abbiamo lo stato di ogni singolo qubit, possiamo combinarli usando il prodotto tensoriale per descrivere lo stato dell'intero registro.

In questo modo possiamo combinare tutti i qubit che vogliamo. Supponiamo di avere quattro qubit, tutti nello stato $|0\rangle$. Il registro di tutti e quattro i qubit potrebbe essere descritto come $|0\rangle \otimes |0\rangle \otimes |0\rangle \otimes |0\rangle$.

Listato 4.6 Creare $|0000\rangle$ come prodotto tensoriale di quattro copie di $|0\rangle$.

```
>>> import numpy as np
>>> ket0 = np.array([[1], [0]])
>>> from functools import reduce
>>> reduce(np.kron, [ket0] * 4) (1)
array([[1], (2)
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
```

```
       [0],
       [0],
       [0]])
```

(1) La funzione `reduce` fornita con la libreria standard Python ci permette di applicare una funzione a due argomenti, come `kron`, fra ogni elemento di una lista. Qui, usiamo `reduce` invece di `np.kron(ket0, np.kron(ket0, np.kron(ket0, ket0)))`.

(2) Otteniamo un vettore di stato a quattro qubit che rappresenta $|0\rangle \otimes |0\rangle \otimes |0\rangle \otimes |0\rangle = |0000\rangle$.

> **NOTA**
>
> Lo stato di due qubit $(|00\rangle + |11\rangle) / \sqrt{2}$ che abbiamo visto in precedenza *non può* essere scritto come il prodotto tensoriale di due stati di un singolo qubit. Gli stati multi-qubit che non possono essere scritti come prodotti tensoriali sono chiamati *entangled*. Parleremo di *entanglement* in tutto il resto di questo capitolo e del libro.

Prodotti tensoriali per operazioni sui qubit nei registri

Ora che sappiamo come utilizzare il prodotto tensoriale per combinare gli stati quantistici, che cosa fa, effettivamente, `np.kron`? In sostanza, il prodotto tensoriale è una tabella di ogni diverso modo di combinare i suoi due argomenti, come illustrato nella Figura 4.2.

Lo stesso prodotto tensoriale illustrato nella Figura 4.2 di due matrici in Python è presentato anche nel seguente listato.

Listato 4.7 Trovare il prodotto tensoriale di A e B con NumPy.

```
>>> import numpy as np
>>> A = np.array([[1, 3], [5, 7]]) (1)
>>> B = np.array([[2, 4], [6, 8]])
>>> np.kron(A, B) (2)
array([[ 2,  4,  6, 12],
       [ 6,  8, 18, 24],
       [10, 20, 14, 28],
       [30, 40, 42, 56]])
>>> np.kron(B, A) (3)
```

```
array([[ 2,  6,  4, 12],
       [10, 14, 20, 28],
       [ 6, 18,  8, 24],
       [30, 42, 40, 56]])
```

(1) Le matrici `A` e `B` sono matrici 2 × 2 arbitrarie, usate come esempi.

(2) Come abbiamo già visto, `np.kron` è l'implementazione NumPy del prodotto tensoriale.

(3) Notate che l'ordine degli argomenti per il prodotto tensoriale è importante. Sebbene sia `np.kron(A, B)` sia `np.kron(B, A)` contengano le stesse informazioni, le voci in ciascuna di esse sono ordinate in modo abbastanza differente.

Usando il prodotto tensoriale tra due matrici, possiamo scoprire come diverse operazioni quantistiche trasformano lo stato di un registro quantistico. Possiamo anche capire come un'operazione quantistica trasforma lo stato di più qubit prendendo invece il prodotto tensoriale di due matrici, cosa che ci fa capire come le nostre mosse e quelle di Eve in un gioco non locale influenzino il nostro stato condiviso.

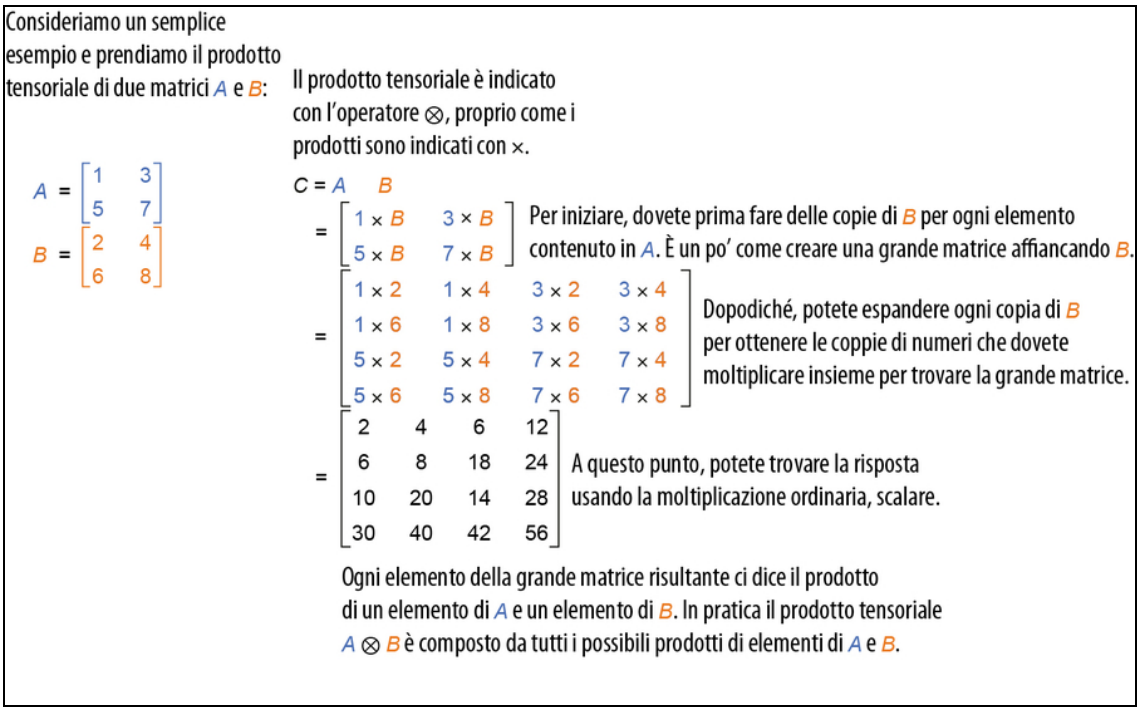


Figura 4.2 Il prodotto tensoriale di due matrici mostrato passo dopo passo. Iniziamo moltiplicando ogni termine di A per una copia completa di B. Le dimensioni della matrice risultante del prodotto tensoriale sono il prodotto delle dimensioni delle matrici di input.

Per esempio, sappiamo che possiamo scrivere $|1\rangle$ come $X|0\rangle$ (ovvero, il risultato dell'applicazione dell'istruzione `x` a un qubit inizializzato). Questo ci offre anche un altro modo per scrivere stati multi-qubit come lo stato $|01\rangle$ che abbiamo già visto. In questo caso, possiamo ottenere $|01\rangle$ applicando un'istruzione `x` *solo* al secondo qubit di un registro di due qubit. Usando il prodotto tensoriale, possiamo trovare una matrice unitaria per rappresentarlo.

Listato 4.8 Calcolo del prodotto tensoriale di due matrici.

```
>>> import numpy as np
>>> I = np.array([[1, 0], [0, 1]]) (1)
>>> X = np.array([[0, 1], [1, 0]]) (2)
>>> IX = np.kron(I, X) (3)
>>> IX
array([[0, 1, 0, 0], (4)
       [1, 0, 0, 0],
       [0, 0, 0, 1],
       [0, 0, 1, 0]])
>>> ket0 = np.array([[1], [0]]) (5)
>>> ket00 = np.kron(ket0, ket0)
>>> ket00
array([[1],
       [0],
       [0],
       [0]])
>>> IX @ ket00
array([[0],
       [1], (6)
       [0],
       [0]])
```

(1) Definisce una matrice che rappresenta un'operazione nulla sul primo qubit, nota come *matrice identità*, $\mathbb{1}$. Poiché $\mathbb{1}$ è difficile da scrivere in Python, usiamo invece `I`.

(2) Definisce la matrice unitaria X che ci permette di simulare l'istruzione `x`.

(3) Combina i due utilizzando il prodotto tensoriale `I` $\otimes$ X.

(4) La matrice $\mathbb{1} \otimes X$ consiste di due copie di X che rappresentano ciò che accade al secondo qubit per ogni possibile stato del primo qubit.

(5) Vediamo che cosa succede quando usiamo $\mathbb{1} \otimes X$ per simulare come l'istruzione `x` trasforma il secondo qubit in un registro di due qubit. Inizieremo con il registro nello stato $|00\rangle = |0\rangle \otimes |0\rangle$.

(6) Riconosciamo lo stato che riceviamo come lo stato $|01\rangle$ di prima. Come previsto, questo è lo stato che otteniamo invertendo il secondo qubit da $|0\rangle$ a $|1\rangle$.

ESERCIZIO 4.2 L'OPERAZIONE DI HADAMARD SU UN REGISTRO DI DUE QUBIT

Come prepareresti uno stato $|+0\rangle$? Innanzitutto, quale vettore impiegheresti per rappresentare lo stato di due qubit $|0\rangle = |+\rangle \otimes |0\rangle$? Avete un registro iniziale di due qubit nello stato $|00\rangle$. Quale operazione dovreste applicare per ottenere lo stato desiderato?

Suggerimento: provate $(H \otimes \mathbb{1})$ se non ci riuscite.

Approfondimento: dimostrazione del teorema di non-clonazione

Il fatto che le operazioni su più qubit siano rappresentabili anche con matrici unitarie ci consente finalmente di dimostrare il teorema di non-clonazione di cui abbiamo già parlato. Il concetto chiave è che la clonazione di uno stato non è lineare e quindi non può essere scritta come una matrice.

Come per molte altre dimostrazioni matematiche, anche la dimostrazione del teorema di non-clonazione funziona *per contraddizione*. Cioè, assumiamo l'opposto del teorema e dimostriamo che, a causa di tale assunto, otteniamo qualcosa di falso.

Partiamo quindi dal presupposto di avere una qualche meravigliosa istruzione `clone` in grado di clonare perfettamente lo stato del suo qubit. Per esempio, se abbiamo un qubit `q1` il cui stato inizia in $|1\rangle$ e un qubit `q2` il cui stato inizia in $|0\rangle$, dopo aver richiamato `q1.clone(q2)`, avremmo il registro $|11\rangle$.

Allo stesso modo, se `q1` parte in $|+\rangle$, allora `q1.clone(q2)` dovrebbe darci un registro nello stato $|++\rangle = |+\rangle \otimes |+\rangle$. Il problema sta nel riconciliare ciò che dovrebbe fare `q1.clone(q2)` in questi due casi. Sappiamo che qualsiasi operazione quantistica diversa dalla misurazione deve essere lineare, quindi diamo un nome alla matrice

che ci permette di simulare `clone` un nome: C sembra un nome abbastanza ragionevole.
Usando C, possiamo scomporre la clonazione di $|+\rangle$ nella clonazione di $|0\rangle$ più $|1\rangle$. Sappiamo che $C|+0\rangle = |++\rangle$, ma sappiamo anche che $C|+0\rangle = C(|00\rangle + |10\rangle) / \sqrt{2} = (C|00\rangle + C|10\rangle) / \sqrt{2}$. Poiché `clone` deve clonare $|0\rangle$ e $|1\rangle$ oltre a $|+\rangle$, sappiamo che $C|00\rangle = |00\rangle$ e $C|10\rangle = |11\rangle$. Questo ci dà che $(C|00\rangle + C|10\rangle) / \sqrt{2} = (|00\rangle + |11\rangle) / \sqrt{2}$, ma abbiamo concluso in precedenza che $C|0\rangle = |++\rangle = (|00\rangle + |01\rangle + |10\rangle + |11\rangle) / \sqrt{2}$.
Giungiamo quindi a una contraddizione, e possiamo concludere che abbiamo sbagliato nel primo passaggio, dove abbiamo supposto che potesse esistere `clone`. Pertanto, abbiamo dimostrato il teorema di non-clonazione.
Una cosa importante da notare da questa dimostrazione è che possiamo sempre copiare le informazioni da un qubit a un altro *se conosciamo le basi giuste*. Il problema si è verificato quando non sapevamo se dovevamo copiare le informazioni su $|0\rangle$ rispetto a $|1\rangle$ o su $|+\rangle$ rispetto a $|-\rangle$, in quanto potremmo copiare o $|0\rangle$ o $|+\rangle$, ma non entrambi. Questo non è un problema classico, in quanto abbiamo solo la base computazionale con cui lavorare.

Finora, abbiamo usato NumPy per scrivere il nostro simulatore di qubit, `SingleQubitSimulator()`. Questo è molto utile, poiché senza NumPy, avremmo dovuto scrivere le nostre funzioni e i nostri metodi di analisi delle matrici. Tuttavia, spesso è utile fare affidamento su package Python dotati di un supporto specifico per i concetti quantistici, basandosi sull'eccellente supporto numerico fornito da NumPy e SciPy (un'estensione numerica di NumPy). Nel prossimo capitolo, parleremo di un package Python progettato per il quantum computing, chiamato QuTiP (Quantum Toolbox in Python), che ci aiuterà a estendere il nostro simulatore per simulare il gioco CHSH.

Riepilogo

- I giochi non locali, come il gioco CHSH, sono esperimenti che possono essere utilizzati per verificare che la meccanica

quantistica sia coerente con il funzionamento del nostro universo.

- Ci vuole un numero esponenziale di bit classici per scrivere lo stato di un registro quantistico, cosa che rende molto difficile simulare più di circa 30 qubit su computer tradizionali.
- Possiamo combinare insieme gli stati mono-qubit usando il prodotto tensoriale, cosa che ci permette di descrivere gli stati multi-qubit.
- Gli stati multi-qubit che non possono essere scritti combinando stati mono-qubit rappresentano qubit *entangled*.

Giochi non locali: implementazione di un simulatore multi-qubit

Nel capitolo precedente, abbiamo imparato a conoscere i giochi non locali e come possiamo usarli per verificare la nostra comprensione della meccanica quantistica. Abbiamo anche imparato a rappresentare stati di più qubit e cos'è l'*entanglement*.

In questo capitolo, ci immergeremo in un nuovo package Python, QuTiP, che ci permetterà di programmare sistemi quantistici più velocemente e che offre alcune funzionalità interessanti per la simulazione della meccanica quantistica. Quindi impareremo a usare QuTiP per programmare un simulatore multi-qubit e vedremo come esso cambia (o non cambia) le tre attività principali per i nostri qubit: preparazione dello stato, operazioni e misurazione. Questo ci consentirà di completare l'implementazione del gioco CHSH del Capitolo 4.

Oggetti quantistici in QuTiP

QuTiP (Quantum Toolbox in Python, `www.qutip.org`) è un package particolarmente utile, in quanto fornisce il supporto necessario per rappresentare gli stati e le misurazioni, rispettivamente come bra e ket, e per costruire le matrici che rappresentano le operazioni quantistiche. Proprio come `np.array` è alla base di NumPy, tutto il nostro uso di

QuTiP sarà incentrato sulla classe `Qobj` (abbreviazione di *quantum object*). Questa classe incapsula i vettori e le matrici, fornendo alcuni metadati aggiuntivi e metodi utili che ci aiuteranno a migliorare il nostro simulatore. La Figura 5.1 mostra un esempio di creazione di un `Qobj` da un vettore, con alcuni metadati.

- `data` contiene l'array che rappresenta il `Qobj`.
- `dims` è la dimensione del nostro registro quantistico. Possiamo considerarlo un modo per sapere come registriamo i qubit con cui abbiamo a che fare.
- `shape` conserva le dimensioni dell'oggetto originale che abbiamo usato per realizzare il `Qobj`. È analogo all'attributo `np.shape`.
- `type` ci dice che cosa rappresenta il `Qobj` (uno stato `= ket`, una misurazione `= bra` o un operatore `= oper`).

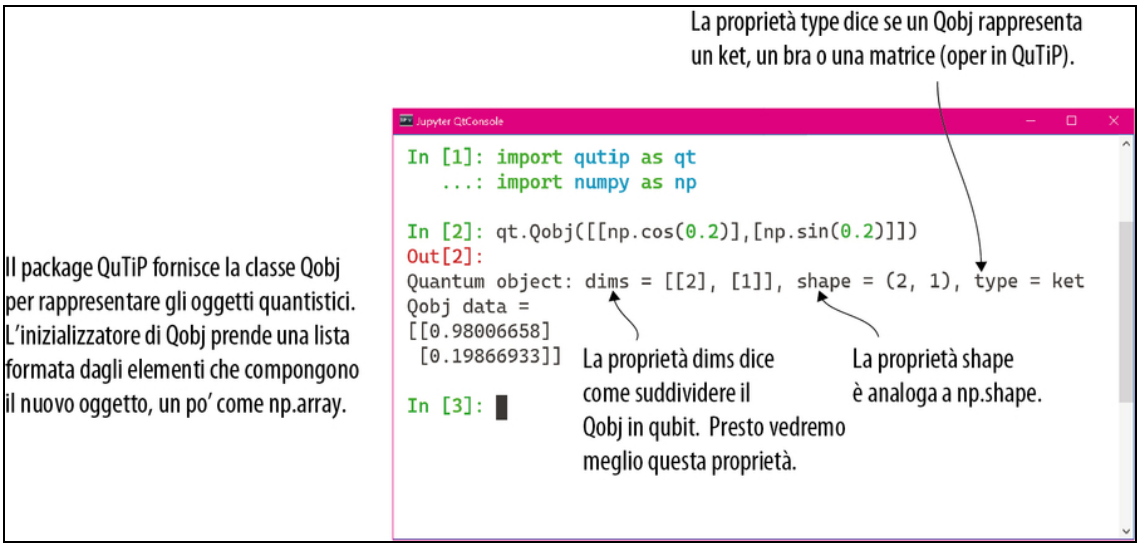


Figura 5.1 Proprietà della classe Qobj del package Python QuTiP. Le proprietà type e dims aiutano a mantenere i metadati relativi ai nostri oggetti quantistici.

Proviamo a importare QuTiP e a chiedergli l'operazione di Hadamard (vedi il Listato 5.1).

NOTA

Assicuratevi di essere nel `conda env` corretto durante l'esecuzione; per ulteriori informazioni, vedere l'Appendice A.

Listato 5.1 Rappresentazione in QuTiP dell'operazione di Hadamard.

```
>>> from qutip.qip.operations import hadamard_transform
>>> H = hadamard_transform()
>>> H
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm =
True
Qobj data =
[[ 0.70710678  0.70710678]

 [ 0.70710678 -0.70710678]]
```

Notate che QuTiP mostra alcune informazioni diagnostiche su ciascuna istanza del `Qobj` insieme ai dati stessi. Qui, per esempio, `type = oper` ci dice che `H` rappresenta un operatore (un termine più formale per le matrici che abbiamo visto finora) insieme ad alcune informazioni sulle dimensioni dell'operatore rappresentato da `H`. Infine, l'output `isherm = True` ci dice che `H` è un esempio di un tipo speciale di matrice chiamato *operatore Hermitiano*.

Possiamo creare nuove istanze di `Qobj` più o meno nello stesso modo in cui abbiamo creato nuovi array NumPy, passando liste Python all'inizializzatore di `Qobj`.

Listato 5.2 Creazione di un Qobj da un vettore che rappresenta uno stato del qubit.

```
>>> import qutip as qt
>>> ket0 = qt.Qobj([[1], [0]]) (1)
>>> ket0
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket (2)
Qobj data =
[[1.]

 [0.]]
```

(1) Una differenza fondamentale tra la creazione di istanze di `Qobj` e di array è che quando creiamo istanze di `Qobj`, abbiamo sempre bisogno di

due livelli di liste. La lista esterna è una lista delle righe che compongono la nuova istanza di `Qobj`.

(2) QuTiP mostra alcuni metadati sulla dimensione e la forma del nuovo oggetto quantistico, insieme ai dati contenuti nel nuovo oggetto. In questo caso, i dati del nuovo `Qobj` hanno due righe, di una colonna ciascuna. Lo identifichiamo come il vettore o ket che usiamo per scrivere lo stato $|0\rangle$.

ESERCIZIO 5.1 CREAZIONE DEL QOBJ PER ALTRI STATI

Come creereste un `Qobj` per rappresentare lo stato $|1\rangle$? E per gli stati $|+\rangle$ e $|-\rangle$? Se necessario, tornate al Capitolo 2 per scoprire quali vettori rappresentano quegli stati.

SOLUZIONI DEGLI ESERCIZI

Tutte le soluzioni degli esercizi di questo libro si trovano nel suo archivio di codice: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Aprite la cartella del capitolo in cui vi trovate e poi il notebook Jupyter con il nome che menziona le soluzioni degli esercizi.

QuTiP è davvero utile, in quanto fornisce molte belle scorciatoie per i tipi di oggetti che ci troviamo a manipolare nel quantum computing. Per esempio, avremmo potuto anche creare `ket0` nell'esempio precedente, utilizzando la funzione QuTiP `basis` (vedi il Listato 5.3). La funzione `basis` prende due argomenti. Il primo dice a QuTiP che vogliamo uno stato del qubit: `2` per un singolo qubit, perché quella è la lunghezza di un vettore necessaria per rappresentarlo. Il secondo argomento dice a QuTiP quale stato base vogliamo.

Listato 5.3 Uso si QuTiP per creare facilmente $|0\rangle$ e $|1\rangle$.

```
>>> import qutip as qt
>>> ket0 = qt.basis(2, 0) (1)
>>> ket0
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket (2)
Qobj data =
[[1.]
 [0.]]
>>> ket1 = qt.basis(2, 1) (3)
```

```
>>> ket1
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[0.]

 [1.]]
```

(1) Passa un `2` come primo argomento, per indicare che vogliamo un singolo qubit e passa uno `0` come secondo argomento, perché vogliamo $|0\rangle$.

(2) Notate che qui otteniamo esattamente lo stesso output dell'esempio precedente.

(3) Possiamo anche costruire un oggetto quantistico per $|1\rangle$, passando `1` invece di `0`.

... base?

Come abbiamo già visto, gli stati $|0\rangle$ e $|1\rangle$ costituiscono la *base computazionale* per un singolo qubit. La funzione `basis` di QuTiP prende il nome da questa definizione, poiché crea oggetti quantistici per rappresentare stati computazionali di base.

Ci sono più cose in cielo e in terra dei nostri qubit…

Può sembrare un po' strano che abbiamo dovuto dire a QuTiP che volevamo un qubit. Dopotutto, che cos'altro potremmo volere? Come vedremo, un bel po' di cose.

Ci sono molti altri modi per rappresentare le informazioni classiche, non solo i bit: ci sono i *trit*, che hanno tre possibili valori. Tuttavia, tendiamo a vedere le informazioni classiche rappresentate quasi sempre come bit quando scriviamo programmi, poiché è molto utile scegliere una convenzione e rispettarla. Cose diverse dai bit hanno comunque i loro usi, in domini specializzati come i sistemi di telecomunicazione.

Allo stesso modo, i sistemi quantistici possono avere un numero qualsiasi di stati differenti: possiamo avere qutrit, qu4it, qu5it, qu17it e così via, noti collettivamente come qu*d*it. Sebbene in alcuni casi possa essere utile rappresentare le informazioni quantistiche utilizzando qudit diversi dai qubit, ed essi possano avere alcune proprietà matematiche molto interessanti, i qubit ci danno tutto ciò di cui abbiamo bisogno per immergerci nella programmazione quantistica.

ESERCIZIO 5.2 UTILIZZO DI QT.BASIS PER PIÙ QUBIT

Come potreste usare la funzione `qt.basis` per creare un registro di due qubit nello stato $|10\rangle$? Come potreste creare lo stato $|001\rangle$? Ricordate che il secondo argomento di `qt.basis` è un indice degli stati computazionali di base, che abbiamo visto in precedenza.

QuTiP fornisce anche una serie di altre funzioni per creare oggetti quantistici per rappresentare le matrici unitarie. Per esempio, possiamo creare un oggetto quantistico per la matrice *X* utilizzando la funzione `sigmax`.

Listato 5.4 Utilizzo di QuTiP per creare un oggetto per la matrice X.

```
>>> import qutip as qt
>>> qt.sigmax()
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm =
True
Qobj data =
[[0. 1.]

 [1. 0.]]
```

Come abbiamo visto nel Capitolo 2, la matrice per `sigmax` rappresenta una rotazione di 180° (Figura 5.2).

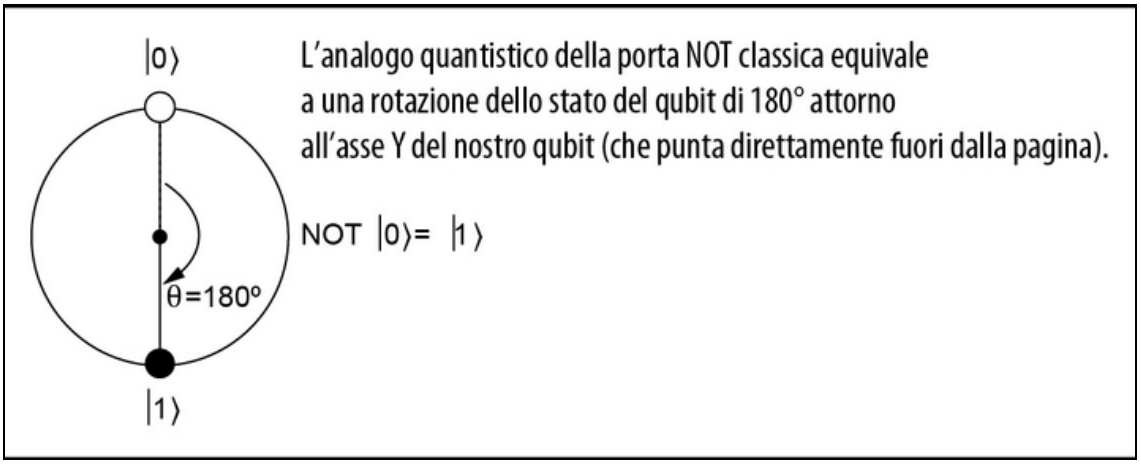


Figura 5.2 Una rappresentazione dell'equivalente quantistico di un NOT operante su un qubit nello stato $|0\rangle$, che lo lascia nello stato $|1\rangle$.

QuTiP fornisce anche una funzione `ry` per rappresentare la rotazione di un angolo qualsiasi invece di 180° come l'operazione `x`. Abbiamo già visto l'operazione rappresentata da `ry` nel Capitolo 2, quando abbiamo considerato la rotazione di $|0\rangle$ di un angolo arbitrario θ. La Figura 5.3 ripropone l'operazione che oggi conosciamo come `ry`.

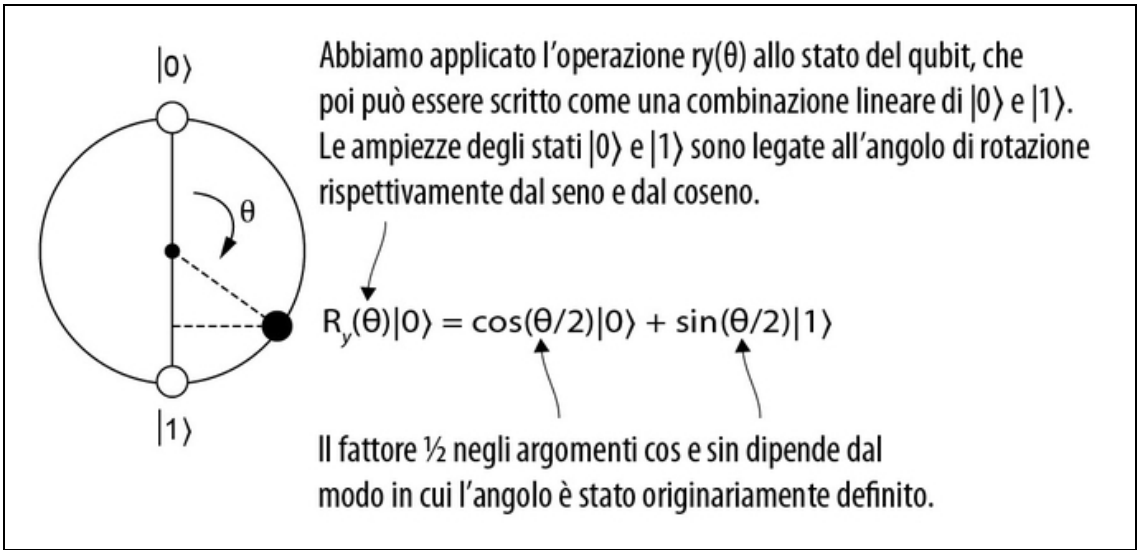


Figura 5.3 Una rappresentazione della funzione QuTiP ry, che corrisponde a una rotazione variabile di θ attorno all'asse Y del nostro qubit (che punti direttamente fuori dalla pagina).

Ora che abbiamo a disposizione qualche operazione mono-qubit, come possiamo simulare facilmente le operazioni multi-qubit in QuTiP? Possiamo utilizzare la funzione `tensor` di QuTiP per ottenere rapidamente il prodotto tensoriale necessario per creare i registri e le operazioni multi-qubit, come illustrato nel Listato 5.5.

NOTA

Poiché le matrici di identità sono spesso scritte usando la lettera *I*, molti package di calcolo scientifico usano il nome `eye`, un gioco di parole, quindi, per far riferimento alla matrice di identità.

Listato 5.5 Prodotti tensoriali in QuTiP.

```
>>> import qutip as qt
>>> from qutip.qip.operations import hadamard_transform
>>> psi = qt.basis(2, 0) (1)
>>> phi = qt.basis(2, 1) (2)
>>> qt.tensor(psi, phi)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.] (3)
 [ 1.]
 [ 0.]
 [ 0.]]
>>> H = hadamard_transform() (4)
>>> I = qt.qeye(2) (5)
>>> qt.tensor(H, I) (6)
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper,
isherm = True
Qobj data =
[[ 0.70710678  0.          0.70710678  0.        ]
 [ 0.          0.70710678  0.          0.70710678]
 [ 0.70710678  0.         -0.70710678  0.        ]

 [ 0.          0.70710678  0.         -0.70710678]]
```

(1) Imposta `psi` per rappresentare $|\psi\rangle = |0\rangle$

(2) Imposta `phi` per rappresentare $|\varphi\rangle = |1\rangle$

(3) Dopo aver richiamato `tensor`, QuTiP ci dice le ampiezze per ogni etichetta classica in $|\psi\rangle \otimes |\varphi\rangle = |0\rangle \otimes |1\rangle = |01\rangle$, usando lo stesso ordine del Listato 4.3.

(4) Imposta `H` per rappresentare l'operazione di Hadamard discussa in precedenza.

(5) Possiamo usare la funzione `qeye` fornita da QuTiP per ottenere una copia di un'istanza di `Qobj` che rappresenta la matrice di identità che abbiamo visto per la prima volta nel Listato 4.8.

(6) Le matrici unitarie che rappresentano le operazioni quantistiche si combinano utilizzando prodotti tensoriali allo stesso modo degli stati e delle misurazioni.

Possiamo usare un classico trucco matematico per dimostrare come funziona l'applicazione di prodotti tensoriali di stati e operazioni. Diciamo che vogliamo dimostrare questa affermazione:

Se applichiamo un operatore unitario a uno stato e quindi prendiamo il prodotto tensoriale, otteniamo lo stesso risultato che otterremmo applicando il prodotto tensoriale e poi l'operatore unitario.

In matematica, diremmo che per ogni operatore unitario U e V e per ogni stato $|\psi\rangle$ e $|\varphi\rangle$, $(U|\psi\rangle) \otimes (V|\varphi\rangle) = (U \otimes V)\,(|\psi\rangle \otimes |\varphi\rangle)$. Il trucco matematico che possiamo usare è prendere il lato sinistro e sottrargli il lato destro. Dovremmo ottenere 0. Dimostriamolo nel seguente listato.

Listato 5.6 Verifica del prodotto tensoriale in QuTiP.

```
>>> (
... qt.tensor(H, I) * qt.tensor(psi, phi) - (1)
... qt.tensor(H * psi, I * phi) (2)
... )
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.] (3)
 [ 0.]
 [ 0.]

 [ 0.]]
```

(1) Il lato destro dell'istruzione che stiamo cercando di dimostrare, dove usiamo `H` e `I` come U e V: $(U \otimes V)\,(|\psi\rangle \otimes |\varphi\rangle)$.

(2) Il lato sinistro dell'istruzione che stiamo cercando di dimostrare, dove usiamo `H` e `I` come U e V: $(U|\psi\rangle) \otimes (V|\varphi\rangle)$.

(3) Esatto! I due lati dell'equazione sono uguali se la loro differenza è 0.

NOTA

Per un elenco di tutti gli stati e le operazioni disponibili in QuTiP, vedi http://qutip.org/docs/latest/guide/guide-basics.html#states-and-operators.

Aggiornamento del simulatore

L'obiettivo ora è quello di utilizzare QuTiP per aggiornare il nostro simulatore mono-qubit, trasformandolo in un simulatore multi-qubit

con alcune delle funzionalità di QuTiP. Lo faremo aggiungendo alcune nuove funzionalità al nostro simulatore mono-qubit sviluppato nei Capitoli 2 e 3.

Il cambiamento più significativo che dovremo apportare al nostro simulatore rispetto ai capitoli precedenti è che non possiamo più assegnare uno stato a ciascun qubit. Piuttosto, dobbiamo assegnare uno stato a un intero *registro* di qubit del nostro dispositivo, in quanto alcuni dei qubit potrebbero essere *entangled* gli uni con gli altri. Passiamo ora ad apportare le modifiche necessarie per separare il concetto di stato a livello del dispositivo.

> **NOTA**
>
> Per guardare il codice che abbiamo scritto in precedenza, così come gli esempi per questo capitolo, consultate l'archivio GitHub del libro: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp.

Per ricapitolare, abbiamo due file per il nostro simulatore: l'interfaccia (`interface.py`) e il simulatore stesso (`simulator.py`). L'interfaccia del dispositivo (`QuantumDevice`) definisce un modo per interagire con un dispositivo quantistico, reale o simulato, che è rappresentato in Python come un oggetto che ci consente di allocare e deallocare qubit.

Non avremo bisogno di inserire nulla di nuovo nella classe `QuantumDevice` dell'interfaccia per modellare il nostro gioco CHSH, poiché dovremo ancora allocare e deallocare qubit. Dove possiamo aggiungere funzionalità è nella classe `Qubit` fornita insieme al nostro `SingleQubitSimulator` in `simulator.py`.

Ora dobbiamo considerare che cosa, eventualmente, dobbiamo cambiare nella nostra interfaccia per un `Qubit` che allochiamo dal `QuantumDevice`. Nel Capitolo 2, abbiamo visto che l'operazione di Hadamard era utile per ruotare i qubit tra basi differenti per creare un generatore quantistico di numeri casuali. Partiamo da qui, aggiungendo

a `Qubit` un nuovo metodo per consentire ai programmi quantistici di inviare un nuovo tipo di istruzione di rotazione di cui avremo bisogno per utilizzare la strategia quantistica per CHSH.

Listato 5.7 interface.py: aggiunta di una nuova operazione ry.

```
class Qubit(metaclass=ABCMeta):
    @abstractmethod
    def h(self): pass
    @abstractmethod
    def x(self): pass
    @abstractmethod
    def ry(self, angle: float): pass (1)
    @abstractmethod
    def measure(self) -> bool: pass
    @abstractmethod

    def reset(self): pass
```

(1) Il metodo astratto `ry`, che accetta un argomento `angle` per specificare di quanto ruotare il qubit attorno all'asse Y.

Questo intervento dovrebbe coprire tutte le modifiche che dobbiamo apportare alla nostra interfaccia con `Qubit` e `QuantumDevice` per giocare a CHSH con Eve. Dobbiamo considerare le modifiche che dobbiamo apportare a `simulator.py` per consentirgli di allocare, operare e misurare anche stati multi-qubit.

Le principali modifiche alla nostra classe `Simulator` che implementa un `QuantumDevice` riguardano il fatto che abbiamo bisogno di nuovi attributi per tenere traccia di quanti qubit ha e dello stato generale del registro. Il prossimo listato mostra queste modifiche e un aggiornamento dei metodi di allocazione e deallocazione.

Listato 5.8 simulator.py: il simulatore multi-qubit.

```
class Simulator(QuantumDevice): (1)
    capacity: int (2)
    available_qubits: List[SimulatedQubit] (3)
    register_state: qt.Qobj (4)
    def __init__(self, capacity=3):
        self.capacity = capacity
        self.available_qubits = [ (5)
            SimulatedQubit(self, idx)
```

```
            for idx in range(capacity)
        ]
        self.register_state = qt.tensor( (6)
            *[
                qt.basis(2, 0)
                for _ in range(capacity)
            ]
        )
    def allocate_qubit(self) -> SimulatedQubit: (7)
        if self.available_qubits:
            return self.available_qubits.pop()
    def deallocate_qubit(self, qubit: SimulatedQubit):

        self.available_qubits.append(qubit)
```

(1) Abbiamo cambiato il nome da `SingleQubitSimulator` a `Simulator`, per indicare il fatto che è più generalizzato. Ciò significa che possiamo usarlo per simulare più qubit.

(2) La classe `Simulator`, più generale, necessita di alcuni attributi, il primo è `capacity`, che rappresenta il numero di qubit che può simulare.

(3) `available_qubits` è una lista contenente i qubit utilizzati dal simulatore.

(4) `register_state` utilizza il nuovo `Qobj` QuTiP per rappresentare lo stato dell'intero simulatore.

(5) Una *list comprehension* ci permette di fare una lista dei qubit disponibili richiamando `SimulatedQubit` con gli indici compresi in `capacity`.

(6) `register_state` viene inizializzato prendendo il prodotto tensoriale di un numero di copie dello stato $|0\rangle$ uguale alla capacità del simulatore. La notazione `*[…]` trasforma la lista generata in una sequenza di argomenti per `qt.tensor`.

(7) I metodi `allocate_qubit` e `deallocate_qubit` sono gli stessi del Capitolo 3.

Non aprire la scatola, mortale!

Proprio come abbiamo usato NumPy per rappresentare lo stato di un simulatore, la proprietà `register_state` del nostro simulatore appena aggiornato usa QuTiP per

prevedere come ogni istruzione ha trasformato lo stato del nostro registro. Quando scriviamo programmi quantistici, però, lo facciamo con l'interfaccia del Listato 5.7, che non ha alcun modo per permetterci di accedere a `register_state`.

Possiamo pensare al simulatore come a una sorta di "scatola nera" che racchiude il concetto di stato. Se i nostri programmi quantistici potessero guardare dentro quella scatola, sarebbero in grado di imbrogliare copiando le informazioni in modi proibiti dal teorema di non-clonazione. Ciò significa che, affinché un programma quantistico sia corretto, non possiamo guardare nel simulatore per conoscere il suo stato.

In questo capitolo, imbrogliamo un po'; ma nel prossimo capitolo aggiusteremo le cose per assicurarci che i nostri programmi possano essere eseguiti anche su vero hardware quantistico.

Aggiungeremo anche un nuovo metodo *privato* al nostro simulatore, che ci consentirà di applicare le operazioni a specifici qubit nel nostro dispositivo. Questo ci consentirà di scrivere i metodi per i nostri qubit, che rimandano le operazioni al simulatore, perché le applichi allo stato di un intero registro di qubit.

SUGGERIMENTO

Python non è rigoroso nel mantenere privati i metodi e gli attributi, ma anteporremo a questo metodo un carattere di sottolineatura, per indicare che è concepito per l'uso solo all'interno della classe.

Listato 5.9 simulator.py: un metodo aggiuntivo per Simulator.

```
def _apply(self, unitary: qt.Qobj, ids: List[int]): (1)
    if len(ids) == 1: (2)
        matrix = qt.circuit.gate_expand_1toN(
            unitary, self.capacity, ids[0]
        )
    else:
        raise ValueError("Only single-qubit unitary matrices are
supported.")

    self.register_state = matrix * self.register_state (3)
```

(1) Il metodo privato `_apply` prende un input unitario di tipo `Qobj` che rappresenta un'operazione unitaria da applicare e una lista di `int` per indicare gli indici della lista `available_qubits` in cui vogliamo applicare

l'operazione. Per ora, quella lista conterrà sempre e solo un elemento, poiché stiamo implementando solo operazioni mono-qubit nel nostro simulatore. Estenderemo questo aspetto nel prossimo capitolo.
(2) Se vogliamo applicare un'operazione mono-qubit a un qubit in un registro, possiamo usare QuTiP per generare la matrice di cui abbiamo bisogno. QuTiP può farlo applicando la matrice della nostra operazione mono-qubit al qubit corretto e applicando $\mathbb{1}$ in ogni altro punto. Questa operazione viene svolta automaticamente dalla funzione `gate_expand_1toN`.

(3) Ora che abbiamo la matrice corretta per la quale moltiplicare il nostro intero `register_state`, possiamo aggiornare il valore di tale registro.

Veniamo all'implementazione di `SimulatedQubit`, la classe che rappresenta il modo in cui simuliamo un singolo qubit, sapendo che fa parte di un dispositivo multi-qubit. La differenza principale tra le versioni mono- e multi-qubit di `SimulatedQubit` è che abbiamo bisogno che ogni qubit ricordi il suo dispositivo "genitore" e la posizione o `id` in quel dispositivo, in modo da poter associare lo stato al registro e non al singolo qubit. Questo è importante, come vedremo nel prossimo paragrafo, quando vogliamo misurare i qubit in un dispositivo multi-qubit.

Listato 5.10 simulator.py: operazioni mono-qubit su un dispositivo multi-qubit.

```
class SimulatedQubit(Qubit):
    qubit_id: int
    parent: "Simulator"
    def __init__(self, parent_simulator: "Simulator", id: int): (1)
        self.qubit_id = id
        self.parent = parent_simulator
    def h(self) -> None:
        self.parent._apply(H, [self.qubit_id]) (2)
    def ry(self, angle: float) -> None:
        self.parent._apply(qt.ry(angle), [self.qubit_id]) (3)
    def x(self) -> None:

        self.parent._apply(qt.sigmax(), [self.qubit_id])
```

(1) Per inizializzare un qubit, abbiamo bisogno del nome del simulatore genitore (così possiamo associarlo facilmente) e dell'indice del qubit nel registro del simulatore. `__init__` quindi imposta quegli attributi e reimposta il qubit allo stato |0⟩.
(2) Per implementare l'operazione `h`, chiediamo al genitore di `SimulatedQubit` (che è un'istanza di `Simulator`) di utilizzare il metodo `_apply` per generare la matrice corretta che rappresenterebbe l'operazione sul registro completo, e poi aggiornare `register_state`.
(3) Possiamo anche passare ad `_apply` l'operazione `qt.ry` parametrizzata di QuTiP per ruotare il nostro qubit attorno all'asse *Y* di un angolo `angle`.

Ottimo! Abbiamo quasi finito di aggiornare il nostro simulatore per utilizzare QuTiP e supportare più qubit. Nel prossimo paragrafo affronteremo la simulazione della misurazione su stati multi-qubit.

Misurazione: come possiamo misurare più qubit?

SUGGERIMENTO

Questo paragrafo è uno dei più impegnativi del libro. Non preoccupatevi se non ne afferrerete completamente il senso alla prima lettura.

In un certo senso, misurare più qubit funziona nello stesso modo in cui siamo abituati a misurare i sistemi mono-qubit. Possiamo ancora utilizzare la regola di Born per predire la probabilità di un qualsiasi risultato della misurazione. Per esempio, torniamo allo stato (|00⟩ + |11⟩) / √2 che ormai abbiamo visto più volte. Se dovessimo misurare una coppia di qubit in quello stato, otterremmo "00" o "11" come risultati classici con uguale probabilità, poiché entrambi hanno la stessa ampiezza: 1 / √2.

Allo stesso modo, chiederemo ancora che se misuriamo lo stesso registro due volte di seguito, otterremo la stessa risposta. Se otteniamo

il risultato “00”, per esempio, sappiamo che i qubit vengono lasciati nello stato $|00\rangle = |0\rangle \otimes |0\rangle$.

Tuttavia, questo diventa un po’ più complicato se misuriamo una *parte* di un registro quantistico senza misurare il tutto. Vediamo un paio di esempi per vedere come potrebbe funzionare la cosa. Prendendo come esempio ancora una volta $(|00\rangle + |11\rangle) / \sqrt{2}$, se misuriamo *solo* il primo qubit e otteniamo uno “0”, sappiamo che dobbiamo ottenere di nuovo la stessa risposta la prossima volta che lo misuriamo. L’unico modo in cui ciò può accadere è se lo stato si trasforma in $|00\rangle$ dopo aver osservato “0” sul primo qubit.

D’altra parte, che cosa succede se misuriamo il primo qubit da una coppia di qubit nello stato $|+\rangle$? In primo luogo, è utile rinfrescare la memoria sull’aspetto di $|+\rangle$ quando è scritto come un vettore.

Listato 5.11 Rappresentazione dello stato $|++\rangle$ con QuTiP.

```
>>> import qutip as qt
>>> from qutip.qip.operations import hadamard_transform
>>> ket_plus = hadamard_transform() * qt.basis(2, 0) (1)
>>> ket_plus
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket (2)
Qobj data =
[[0.70710678]
 [0.70710678]]
>>> register_state = qt.tensor(ket_plus, ket_plus) (3)
>>> register_state
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data = (4)
[[0.5]
 [0.5]
 [0.5]

 [0.5]]
```

(1) Inizia scrivendo $|+\rangle$ come $H|0\rangle$. In QuTiP, usiamo la funzione `hadamard_transform` per ottenere un’istanza di `Qobj` per rappresentare H, e usiamo `basis(2, 0)` per ottenere un `Qobj` che rappresenta $|0\rangle$.

(2) Possiamo mostrare `ket_plus` per ottenere una lista degli elementi contenuti in quel vettore; come prima, chiamiamo ciascuno di questi

elementi con il nome *ampiezza*.

(3) Per rappresentare lo stato $|+\rangle$, usiamo che $|+\rangle = |+\rangle \otimes |+\rangle$.

(4) Questo output ci dice che $|{+}{+}\rangle$ ha la stessa ampiezza per ciascuno dei quattro stati computazionali di base $|00\rangle$, $|01\rangle$, $|10\rangle$ e $|11\rangle$, proprio come `ket_plus` ha la stessa ampiezza per ciascuno degli stati computazionali di base $|0\rangle$ e $|1\rangle$.

Supponiamo di misurare il primo qubit e di ottenere un risultato "1". Per assicurarci di ottenere lo stesso risultato la prossima volta che lo misuriamo, lo stato dopo la misurazione non può avere un'ampiezza $|00\rangle$ o $|01\rangle$. Se manteniamo le ampiezze solo su $|10\rangle$ e $|11\rangle$ (la terza e la quarta riga del vettore che abbiamo calcolato in precedenza), otteniamo che lo stato dei nostri due qubit diventa $(|10\rangle + |11\rangle) / \sqrt{2}$.

Da dove viene $\sqrt{2}$?

Abbiamo incluso un $\sqrt{2}$ per garantire che tutte le nostre probabilità di misurazione siano ancora a somma 1 quando misuriamo il secondo qubit. Affinché la regola di Born abbia senso, abbiamo sempre bisogno che la somma dei quadrati di ciascuna ampiezza abbia somma 1.

C'è un altro modo per scrivere questo stato, però, che possiamo verificare impiegando QuTiP.

Listato 5.12 Rappresentazione dello stato $|1+\rangle$ con QuTiP.

```
>>> import qutip as qt
>>> from qutip.qip.operations import hadamard_transform
>>> ket_0 = qt.basis(2, 0)
>>> ket_1 = qt.basis(2, 1)
>>> ket_plus = hadamard_transform() * ket_0 (1)
>>> qt.tensor(ket_1, ket_plus)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[0.        ]
 [0.        ]
 [0.70710678]

 [0.70710678]]
```

(1) Ricordate che possiamo scrivere $|+\rangle$ come $H|0\rangle$.

Questo ci dice che se manteniamo solo le parti dello stato $|+\rangle$ che sono coerenti con l'ottenimento di un risultato "1" dalla misurazione del primo qubit, allora otteniamo $|1\rangle = |1\rangle \oplus |+\rangle$. Cioè, in questo caso non succede assolutamente nulla al secondo qubit.

ESERCIZIO 5.3 MISURARE L'ALTRO QUBIT

Nell'esempio in cui i nostri due qubit partono nello stato $|++\rangle$, supponiamo di aver misurato, invece, il secondo qubit. Verificate che, indipendentemente dal risultato che otteniamo, non accada nulla allo stato del primo qubit.

Per capire più in generale che cosa significa misurare una parte di un registro, possiamo usare un altro concetto dell'algebra lineare: i *proiettori*.

DEFINIZIONE

Un *proiettore* è il prodotto di un vettore di stato (la parte "ket" o $|+\rangle$ di un bra-ket) e di una misurazione (la parte "bra" o $\langle+|$ di un bra-ket) e rappresenta il nostro requisito che *se* si verifica un determinato risultato della misurazione, *allora* dobbiamo eseguire una trasformazione in uno stato coerente con tale misurazione.

La Figura 5.4 rappresenta un rapido esempio di un proiettore mono-qubit. La definizione dei proiettori su più qubit funziona esattamente allo stesso modo.

In QuTiP, scriviamo il bra corrispondente a un ket usando il metodo `.dag()` (abbreviazione di *dagger*, un richiamo alla notazione matematica che vediamo nella Figura 5.4). Fortunatamente, anche se la matematica impiegata non è semplice, non è poi così difficile scriverla in Python, come possiamo vedere di seguito.

Listato 5.13 simulator.py: misurare i singoli qubit di un registro.

```
def measure(self) -> bool:
    projectors = [ (1)
        qt.circuit.gate_expand_1toN( (2)
            qt.basis(2, outcome) * qt.basis(2, outcome).dag(),
            self.parent.capacity,
            self.qubit_id
        )
```

```
        for outcome in (0, 1)
    ]
    post_measurement_states = [
        projector * self.parent.register_state (3)
        for projector in projectors
    ]
    probabilities = [ (4)
        post_measurement_state.norm() ** 2
        for post_measurement_state in post_measurement_states
    ]
    sample = np.random.choice([0, 1], p=probabilities) (5)
    self.parent.register_state = post_measurement_states[sample].unit()
(6)
    return bool(sample)
def reset(self) -> None: (7)

    if self.measure(): self.x()
```

(1) Impiega QuTiP per creare una lista di proiettori, uno per ogni possibile risultato della misurazione.

(2) Come nel Listato 5.9, utilizziamo la funzione `gate_expand_1toN` per espandere ogni proiettore mono-qubit in un proiettore che agisce sull'intero registro.

(3) Utilizza ciascun proiettore per scegliere le parti di uno stato che sono coerenti con il risultato delle misurazioni.

(4) La lunghezza di ciò che viene selezionato da ciascun proiettore (il metodo QuTiP `.norm()`) ci dice la probabilità di ogni risultato della misurazione.

(5) Una volta che abbiamo le probabilità di ogni risultato, possiamo scegliere un risultato usando NumPy.

(6) Utilizza il metodo QuTiP `.unit()` per garantire che le probabilità delle misurazioni abbiano ancora somma 1.

(7) Se il risultato di una misurazione è $|1\rangle$, allora l'applicazione di un'istruzione `x` lo riporta a $|0\rangle$.

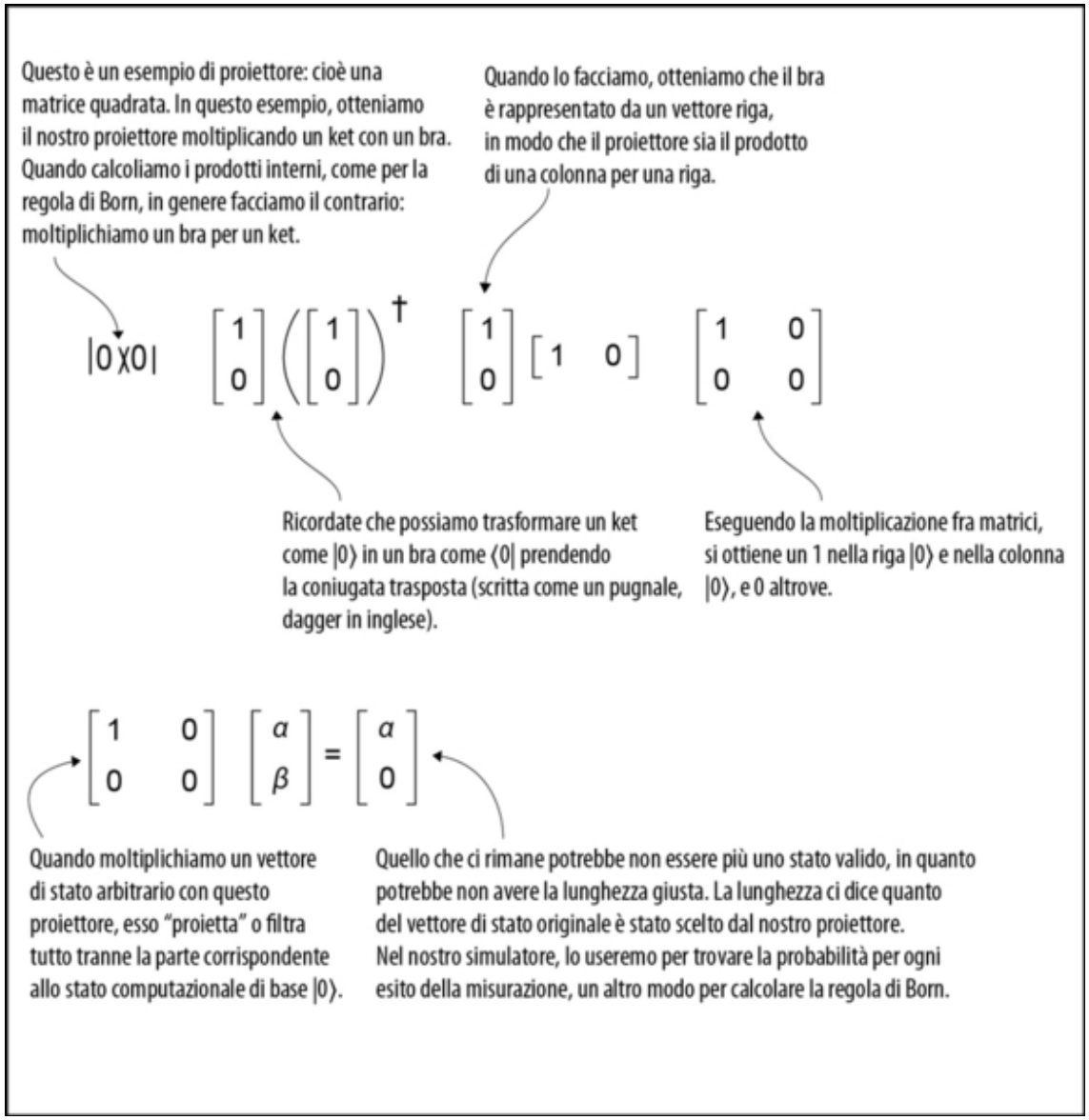


Figura 5.4 Un esempio di un proiettore che agisce su uno stato mono-qubit.

CHSH: strategia quantistica

Ora che abbiamo esteso il nostro simulatore per fargli gestire più qubit, vediamo come possiamo simulare una strategia quantistica per i nostri giocatori, che si tradurrà in una probabilità di vittoria più elevata

di quanto sarebbe possibile con qualsiasi strategia classica. La Figura 5.5 mostra un promemoria del gioco CHSH.

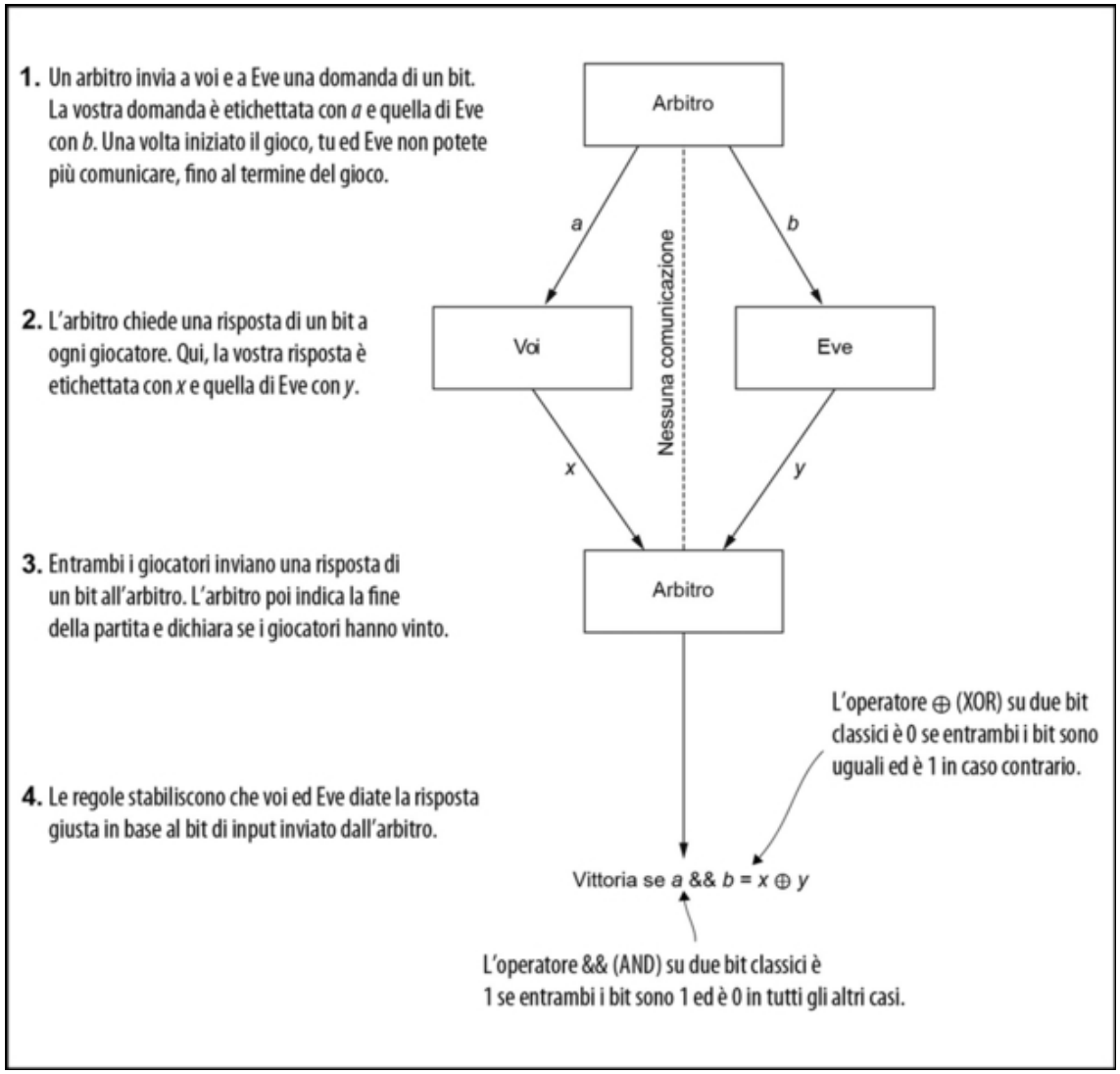


Figura 5.5 CHSH è un gioco non locale che prevede due giocatori e un arbitro. L'arbitro pone a ogni giocatore una domanda sotto forma di un bit; poi ogni giocatore deve rispondere all'arbitro. I giocatori vincono se lo XOR booleano delle loro risposte è uguale all'AND delle domande dell'arbitro.

Voi ed Eve ora avete a disposizione delle risorse quantistiche, quindi iniziamo con l'opzione più semplice: ognuno di voi ha un qubit allocato sullo stesso dispositivo. Useremo il nostro simulatore per implementare questa strategia, quindi questo non è tanto un test di

meccanica quantistica, quanto una verifica che il nostro simulatore concordi con la meccanica quantistica.

> **NOTA**
>
> Non possiamo simulare giocatori veramente non locali, poiché le parti del simulatore devono comunicare per emulare la meccanica quantistica. Simulare fedelmente giochi e protocolli di rete quantistici in questo modo fa sorgere molte e interessanti domande sulla topologia di rete classica, che vanno ben oltre gli scopi di questo libro. Se siete interessati ai simulatori destinati a un utilizzo nel networking quantistico più che al quantum computing, consultate il progetto *SimulaQron* (`www.simulaqron.org`) per ulteriori informazioni.

Vediamo con quale frequenza noi ed Eve possiamo vincere se ciascuno di noi inizia con un singolo qubit, e se questi qubit partono nello stato $(|00\rangle + |11\rangle) / \sqrt{2}$ che abbiamo già visto più volte nel capitolo. Non preoccupatevi di come preparare questo stato; impareremo a farlo nel Capitolo 6. Per ora, vediamo che cosa possiamo fare con i qubit in quello stato una volta che li abbiamo.

Impiegando questi qubit, possiamo ideare una nuova strategia quantistica per il gioco CHSH che abbiamo visto all'inizio del capitolo. Il trucco è che noi ed Eve possiamo applicare operazioni a ciascuno dei nostri qubit, una volta che otteniamo i nostri rispettivi messaggi dall'arbitro.

Come vedremo, `ry` è un'operazione molto utile per questa strategia. Permette a noi e a Eve di sacrificare leggermente la frequenza della vincita quando l'arbitro ci chiede di fornire le stesse risposte (i casi 00, 01 e 10), facendo un po' meglio quando invece dobbiamo produrre risposte differenti (il caso 11) come illustrato nella Figura 5.6.

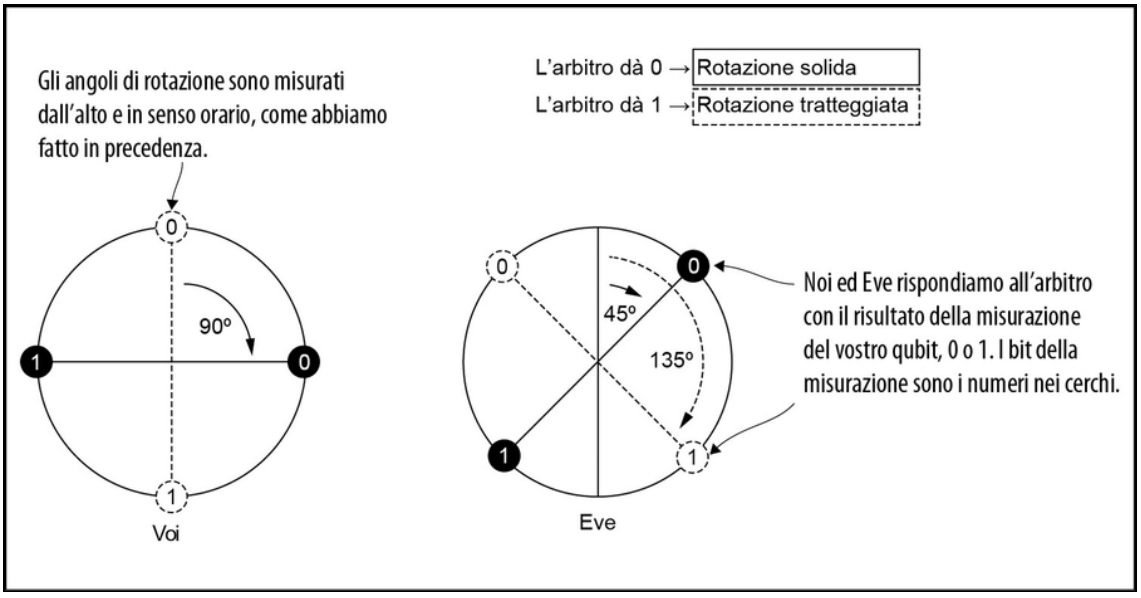


Figura 5.6 Qubit rotanti per vincere a CHSH. Se otteniamo uno 0 dall'arbitro, dovremmo ruotare il nostro qubit di 45°; se otteniamo un 1, dovremmo ruotare il nostro qubit di 135°.

Possiamo vedere da questa strategia che sia noi sia Eve abbiamo una regola piuttosto semplice e diretta su che cosa fare con i nostri rispettivi qubit prima di misurarli. Se otteniamo uno 0 dall'arbitro, dovremmo ruotare il nostro qubit di 45°; se otteniamo un 1, dovremmo ruotare il nostro qubit di 135°. Se preferite un approccio tabellare a questa strategia, la Tabella 5.1 mostra un riepilogo.

Tabella 5.1 Rotazioni che noi ed Eve applicheremo ai nostri qubit in funzione del bit di input che riceviamo dall'arbitro. Notate che sono tutte rotazioni ry, solo con angoli differenti (convertiti in radianti per ry).

Input dell'arbitro	Rotazione nostra	Rotazione di Eve
0	ry(90 * np.pi / 180)	ry(45 * np.pi / 180)
1	ry(0)	ry(135 * np.pi / 180)

Non preoccupatevi se questi angoli vi sembrano casuali. Possiamo verificare che funzionino utilizzando il nostro nuovo simulatore. Il

prossimo listato utilizza le nuove funzionalità che abbiamo aggiunto al simulatore per implementare una strategia quantistica.

Listato 5.14 chsh.py: una strategia CHSH quantistica, che utilizza due qubit.

```
import qutip as qt
def quantum_strategy(initial_state: qt.Qobj) -> Strategy:
    shared_system = Simulator(capacity=2) (1)
    shared_system.register_state = initial_state
    your_qubit = shared_system.allocate_qubit() (2)
    eve_qubit = shared_system.allocate_qubit()
    shared_system.register_state = qt.bell_state() (3)
    your_angles = [90 * np.pi / 180, 0] (4)
    eve_angles = [45 * np.pi / 180, 135 * np.pi / 180]
    def you(your_input: int) -> int:
        your_qubit.ry(your_angles[your_input]) (5)
        return your_qubit.measure() (6)
    def eve(eve_input: int) -> int: (7)
        eve_qubit.ry(eve_angles[eve_input])
        return eve_qubit.measure()
    return you, eve (8)
```

(1) Per avviare la strategia quantistica, dobbiamo creare un'istanza di `QuantumDevice` in cui simuleremo i nostri qubit.

(2) Possono essere assegnate delle etichette a ogni qubit mentre li allochiamo in `shared_system`.

(3) Imbrogliamo un po' per impostare lo stato dei nostri qubit sullo stato *entangled* $(|00\rangle + |11\rangle) / \sqrt{2}$. Vedremo nel Capitolo 6 come preparare questo stato da zero e perché la funzione per preparare questo stato si chiama `bell_state`.

(4) Angoli per le rotazioni che noi ed Eve dobbiamo applicare in base al nostro input dall'arbitro.

(5) La strategia per giocare a CHSH inizia con noi che ruotiamo il nostro qubit in base al bit classico di input che abbiamo ricevuto dall'arbitro.

(6) Il valore di bit classico restituito dalla nostra strategia è il valore di bit che otteniamo quando misuriamo il nostro qubit.

(7) La strategia di Eve è simile alla nostra; usa solo angoli diversi per la sua rotazione iniziale.

(8) Proprio come la nostra strategia classica, `quantum_strategy` restituisce una tupla di funzioni che rappresentano le azioni individuali nostra e di Eve.

Ora che abbiamo implementato una versione Python di `quantum_strategy`, vediamo quanto spesso possiamo vincere usando la nostra funzione `est_win_probability` del gioco CHSH.

Listato 5.15 Eseguire CHSH con la nostra nuova quantum_strategy.

```
>>> est_win_probability(quantum_strategy) (1)

0.832
```

(1) Potreste ottenere più o meno l'85%, perché la probabilità di vincita è stimata tramite una distribuzione binomiale. Per questo esempio, ci aspetteremmo un errore di circa l'1,5%.

La probabilità di vittoria stimata all'83,2% nel Listato 5.15 è superiore a quella che possiamo ottenere con qualsiasi strategia classica. Ciò significa che noi ed Eve possiamo iniziare a vincere il gioco CHSH più frequentemente di qualsiasi altro giocatore classico, e questo è fantastico! Ciò che mostra questa strategia, però, è un esempio di come stati come $(|00\rangle + |11\rangle) / \sqrt{2}$ sono risorse importanti, disponibili solo grazie alla meccanica quantistica.

> **NOTA**
>
> Stati come $(|00\rangle + |11\rangle) / \sqrt{2}$ sono chiamati *entangled* perché non possono essere scritti come il prodotto tensoriale di stati mono-qubit. Vedremo molti altri esempi di *entanglement* man mano che procederemo, ma l'*entanglement* è uno degli aspetti più sorprendenti e divertenti che possiamo usare nella scrittura di programmi quantistici.

Come abbiamo visto in questo esempio, l'*entanglement* ci consente di creare correlazioni nei nostri dati che possiamo sfruttare per ottenere informazioni utili dai nostri sistemi quantistici.

La velocità della luce è ancora un limite

Se avete nozioni di relatività (se non l'avete, non preoccupatevi), potreste aver sentito che è impossibile inviare informazioni più velocemente della velocità della luce. Da quello che abbiamo imparato finora sull'*entanglement* può sembrare che la meccanica quantistica violi questo assunto, ma l'*entanglement* non può *mai* essere usato da solo per comunicare un messaggio a nostra scelta. Abbiamo sempre bisogno di inviare qualcos'altro oltre a impiegare l'*entanglement*. Ciò significa che la velocità della luce continua a limitare la velocità con cui le informazioni possono viaggiare per l'universo…

Lungi dall'essere strano o bizzarro, l'*entanglement* è il risultato diretto di ciò che abbiamo già imparato sul quantum computing: è una conseguenza diretta del fatto che la meccanica quantistica è *lineare*. Se possiamo preparare un registro di due qubit nello stato $|00\rangle$ e $|11\rangle$, allora possiamo anche preparare uno stato in una combinazione lineare dei due, come $(|00\rangle + |11\rangle) / \sqrt{2}$.

Poiché l'*entanglement* è un risultato diretto della linearità della meccanica quantistica, il gioco CHSH ci offre anche un ottimo modo per verificare che la meccanica quantistica sia veramente corretta (quanto meno al meglio di quanto i nostri dati possono mostrare). Torniamo alla probabilità di vittoria del Listato 5.15. Se facciamo un esperimento e rileviamo una probabilità di vittoria dell'83,2%, questo ci dice che il nostro esperimento non avrebbe potuto essere puramente classico, poiché sappiamo che una strategia classica può vincere al massimo il 75% delle volte. Questo esperimento è stato condotto molte volte e fa parte del modo in cui sappiamo che il nostro universo non è solo classico: abbiamo bisogno anche della meccanica quantistica per descriverlo.

NOTA

Nel 2015, un esperimento prevedeva due giocatori di CHSH separati da più di un chilometro.

Il simulatore che abbiamo scritto in questo capitolo ci fornisce tutto ciò di cui abbiamo bisogno per vedere come funzionano questi tipi di

esperimenti. Ora possiamo proseguire, e usare la meccanica quantistica e i qubit per fare cose stupefacenti, armati della consapevolezza che la meccanica quantistica è davvero il modo in cui funziona il nostro universo.

Autotest: un'applicazione per i giochi non locali
Questo esercizio suggerisce un'altra applicazione per i giochi non locali: se possiamo giocare e vincere una partita non locale con Eve, allora dobbiamo aver costruito qualcosa che possiamo usare per inviare dati quantistici. Questa conoscenza porta a sviluppare idee note come *autotest quantistico*, in cui sottoponiamo a giochi non locali determinate parti di un dispositivo, mentre altre parti del dispositivo servono ad assicurarci che il dispositivo funzioni correttamente.

Riepilogo

- Possiamo usare il package QuTiP per aiutarci a lavorare con i prodotti tensoriali e altri calcoli di cui abbiamo bisogno per scrivere un simulatore multi-qubit in Python.
- La classe `Qobj` di QuTiP conserva molte proprietà utili degli stati e degli operatori che vogliamo simulare.
- Noi ed Eve possiamo usare una strategia quantistica per giocare a CHSH, in cui condividiamo una coppia di qubit *entangled* prima di iniziare a giocare.
- Scrivendo il gioco CHSH come un programma quantistico, possiamo dimostrare che i giocatori che usano coppie di qubit *entangled* possono vincere più spesso dei giocatori che usano solo computer classici; questo è coerente con la nostra conoscenza della meccanica quantistica.

Teletrasporto ed entanglement: trasferimento di dati quantistici

Nel capitolo precedente, con l'aiuto del package QuTiP abbiamo aggiunto al nostro simulatore di dispositivi quantistici il supporto multi-qubit. Ciò ci ha permesso di giocare a CHSH e di dimostrare che la nostra comprensione della meccanica quantistica è coerente con ciò che osserviamo nel mondo reale.

In questo capitolo vedremo come trasferire i dati tra persone o registri in un dispositivo quantistico. Vedremo come cose come il teorema di non-clonazione influenza il modo in cui gestiamo i nostri dati su un dispositivo quantistico. Verificheremo anche un protocollo quantistico unico che il nostro dispositivo quantistico può eseguire, chiamato *teletrasporto*, che sposta i dati (e non le loro copie).

Trasferimento di dati quantistici

Proprio come nel calcolo classico, a volte in un computer quantistico abbiamo alcuni dati che vorremmo tanto poter trasferire da qualche altra parte. Classicamente, questo è un problema facile da risolvere copiando i dati; ma come abbiamo visto nei Capitoli 3 e 4, il teorema di non-clonazione significa che, in generale, non possiamo copiare i dati memorizzati in qubit.

Trasferire i dati in modo classico

In alcuni aspetti del calcolo classico, ci imbattiamo nello stesso problema dell'impossibilità di copiare le informazioni, ma per motivi differenti. La copia dei dati in un'applicazione multithread può introdurre situazioni di competizione sui dati, mentre alcune considerazioni sulle prestazioni possono indurci a ridurre la quantità di dati che copiamo.
La soluzione adottata da molti linguaggi classici (per esempio, C++11 e Rust) si concentra sul *trasferimento* dei dati. Pensare in termini di trasferimento dei dati è utile nel quantum computing, anche se implementeremo le operazioni in un modo molto diverso.

Quindi che cosa possiamo fare se vogliamo trasferire i dati in un dispositivo quantistico? Fortunatamente, esistono diversi modi per trasferire i dati quantistici invece di copiarli. In questo capitolo, vedremo alcuni di questi approcci e aggiungeremo le ultime due funzionalità al nostro simulatore, per implementarli. Cominciamo quindi a condividere le nostre informazioni quantistiche.

Supponiamo che Eve abbia alcuni qubit che codificano i dati che vorrebbe condividere con noi:

> Ehi giocatore! Ho alcune informazioni quantistiche che voglio condividere con te. Posso inviartele?

Qui Eve parla dell'istruzione `swap`, che è un po' diversa dalle istruzioni che abbiamo visto finora, in quanto opera su *due* qubit contemporaneamente. Al contrario, ogni operazione che abbiamo visto finora funziona solo su un qubit per volta.

Vediamo che cosa fa `swap`, il cui nome è piuttosto "parlante": scambia letteralmente lo stato di due qubit nello stesso registro. Per esempio, supponiamo di avere due qubit nello stato $|01\rangle$. Se usiamo l'istruzione `swap` su entrambi i qubit, il nostro registro sarà ora nello stato $|10\rangle$. Vediamo un esempio di utilizzo della matrice di `swap` fornita da QuTiP.

Listato 6.1 Usare lo swap di QuTiP su $|0\rangle$ per avere lo stato $|0+\rangle$.

```
>>> import qutip as qt
>>> from qutip.qip.operations import hadamard_transform
>>> ket_0 = qt.basis(2, 0)
>>> ket_plus = hadamard_transform() * ket_0
>>> initial_state = qt.tensor(ket_plus, ket_0)
>>> initial_state (1)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[0.70710678] (2)
 [0. ]
 [0.70710678]
 [0. ]]
>>> swap_matrix = qt.swap() (3)
>>> swap_matrix * initial_state (4)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[0.70710678] (5)
 [0.70710678]
 [0. ]
 [0. ]]
>>> qt.tensor(ket_0, ket_plus) (6)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[0.70710678]
 [0.70710678]
 [0. ]
 [0. ]]
```

(1) Usa `qt.basis`, `hadamard_transform` e `qt.tensor` per definire una variabile per la nostra vecchia conoscenza del Capitolo 5: il vettore di stato $|+0\rangle$.

(2) Come abbiamo visto nel Capitolo 4, questo stato ha ampiezze uguali sugli stati computazionali di base $|00\rangle$ e $|10\rangle$.

(3) Ottiene una copia della matrice unitaria per l'istruzione `swap` richiamando `qt.swap`.

(4) Allo stesso modo in cui abbiamo simulato le operazioni mono-qubit, possiamo moltiplicare il nostro stato per la matrice unitaria per far sì che `swap` trovi lo stato del nostro registro di due qubit dopo aver applicato l'istruzione `swap`.

(5) Quando lo facciamo, finiamo in una sovrapposizione tra $|00\rangle$ e $|01\rangle$ invece che fra $|00\rangle$ e $|10\rangle$.

(6) Verifica rapidamente che il risultato dell'utilizzo dell'istruzione `swap` su un registro di due qubit che partono nello stato e $|+0\rangle$ sia $|0+\rangle$.

Osservando il Listato 6.1, possiamo vedere che l'istruzione `swap` ha fatto più o meno quello che suggerisce il nome. In particolare, `swap` ha preso due qubit partiti nello stato $|+0\rangle$ e li ha portati nello stato $|0+\rangle$. Più in generale, possiamo scoprire che cosa fa l'istruzione `swap` osservando la matrice unitaria che abbiamo usato per simularla.

Listato 6.2 Matrice unitaria per l'istruzione swap.

```
>>> import qutip as qt
>>> qt.swap()
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper,
isherm = True (1)
Qobj data =
[[1. 0. 0. 0.] (2)
 [0. 0. 1. 0.] (3)
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]] (4)
```

(1) La matrice unitaria che utilizziamo per simulare l'istruzione `swap` è una matrice 4×4, perché agisce sugli stati di due qubit.

(2) Ogni colonna di questa matrice unitaria ci dice che cosa succede a uno degli stati computazionali di base; qui, l'istruzione `swap` non fa nulla ai qubit che partono nello stato $|00\rangle$.

(3) Al contrario, gli stati $|01\rangle$ e $|10\rangle$ vengono scambiati dall'istruzione `swap`.

(4) Anche $|11\rangle$ non viene modificato dall'istruzione `swap`.

Ogni colonna di questa matrice unitaria ci dice che cosa succede a uno degli stati computazionali di base. Per esempio, la prima colonna ci dice che lo stato $|00\rangle$ è mappato sul vettore `[[1], [0], [0], [0]]`, che riconosciamo come $|00\rangle$.

NOTA

La matrice unitaria per l'istruzione `swap` *non può* essere scritta come prodotto tensoriale di due matrici unitarie mono-qubit. Cioè, non possiamo capire che

cosa fa `swap` considerando un qubit alla volta: dobbiamo capire che cosa fa allo stato della coppia di qubit su cui agisce l'istruzione `swap`.

La Figura 6.1 mostra che possiamo vedere che cosa fa `swap` in generale, indipendentemente dallo stato in cui iniziano i nostri due qubit.

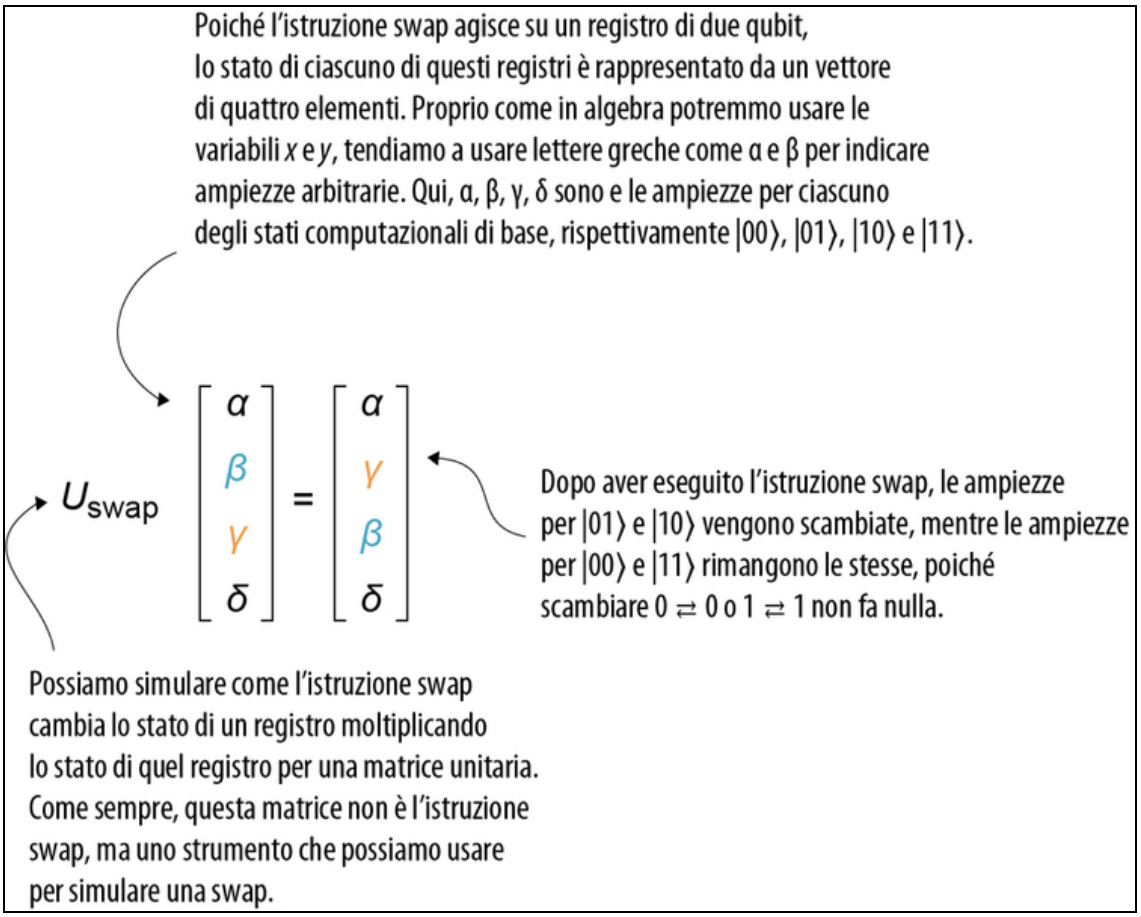


Figura 6.1 L'operazione di swap su due qubit scambia gli stati di due qubit in un registro. Possiamo vederlo nell'esempio generico qui mostrato, dove vengono scambiati i termini che descrivono gli stati |01⟩ e |10⟩. Gli altri due non vengono scambiati, perché non siamo in grado di trovare una differenza quando i due qubit sono nello stesso stato.

Ricordate che nel Capitolo 2 abbiamo visto che una matrice unitaria è molto simile a una tabella di verità. Cioè, le matrici unitarie come quelle che otteniamo da `qt.swap` sono utili in quanto ci aiutano a

simulare ciò che fa l'istruzione `swap`. Proprio come un sommatore classico non è la sua tabella della verità, tuttavia, è utile ricordare che queste matrici unitarie non sono programmi quantistici, ma strumenti che utilizziamo per simulare il funzionamento dei programmi quantistici.

ESERCIZIO 6.1 SCAMBIARE IL SECONDO E IL TERZO QUBIT IN UN REGISTRO

Supponiamo di avere un registro con tre qubit nello stato $|01+\rangle$. Impiegando QuTiP, scrivete questo stato, quindi scambiate il secondo e il terzo qubit in modo che il registro sia nello stato $|0+1\rangle$.

Suggerimento: poiché non accadrà nulla al primo qubit, assicuratevi di calcolare il prodotto tensoriale di una matrice identità e di usare `qt.swap` per creare l'operazione corretta per il vostro registro.

SOLUZIONI DEGLI ESERCIZI

Tutte le soluzioni degli esercizi di questo libro si trovano nel suo archivio di codice: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Aprite la cartella del capitolo in cui vi trovate e poi il notebook Jupyter con il nome che menziona le soluzioni degli esercizi.

A questo punto, Eve è *decisamente ansiosa* di inviare i suoi qubit. Aggiungiamo quello che ci serve al nostro simulatore per non farla aspettare troppo.

Il simulatore con swap

Il simulatore su cui abbiamo lavorato nel Capitolo 4 necessita solo di un paio di modifiche per poter utilizzare operazioni a due qubit come `swap`. Le modifiche che dobbiamo apportare sono le seguenti:

- modificare `_apply` per operazioni a due qubit;
- aggiungere `swap` e altre istruzioni a due qubit;
- aggiungere le altre istruzioni di rotazione mono-qubit.

Come abbiamo visto nel Capitolo 4, se una matrice agisce su un registro mono-qubit, possiamo usare QuTiP per applicarla a un registro con un numero arbitrario di qubit utilizzando la funzione `gate_expand_1toN`, che prende il prodotto tensoriale degli operatori di identità su ciascun qubit, tranne i qubit sui quali stiamo lavorando.

Allo stesso modo, possiamo richiamare la funzione `gate_expand_2toN` di QuTiP per trasformare matrici unitarie a due qubit in matrici che possiamo usare per simulare il modo in cui operazioni a due qubit, come `swap`, trasformano lo stato di un intero registro. Aggiungiamo ora il tutto al nostro simulatore (Listato 6.3).

SUGGERIMENTO

Abbiamo apportato un paio di piccole modifiche al codice in questo capitolo, con lo scopo di rendere l'output un po' più gradevole. Queste modifiche, insieme a tutti gli esempi per questo e degli altri capitoli, si trovano nell'archivio GitHub di questo libro: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp.

Listato 6.3 simulator.py: applicazione di matrici unitarie a due qubit.

```
def _apply(self, unitary: qt.Qobj, ids: List[int]):
if len(ids) == 1:
matrix = qt.circuit.gate_expand_1toN(unitary,
self.capacity, ids[0])
elif len(ids) == 2: (1)
matrix = qt.circuit.gate_expand_2toN(unitary, self.capacity, *ids) (2)
else:
raise ValueError("Only one- or two-qubit unitary matrices supported.")
self.register_state = matrix * self.register_state
```

(1) Per simulare operazioni a due qubit, abbiamo bisogno di due indici per i qubit del registro: uno per ogni qubit su cui agisce la nostra istruzione.

(2) La chiamata a `gate_expand_2toN` è molto simile alla nostra chiamata a `gate_expand_1toN`, tranne per il fatto che passiamo una matrice 4×4 anziché una matrice 2×2.

Abbiamo visto che QuTiP fornisce la funzione `swap` per darci una copia della matrice unitaria che simula l'istruzione `swap`. Questo può essere utilizzato per aggiungere abbastanza rapidamente l'istruzione `swap` al nostro simulatore, utilizzando le modifiche apportate a `Simulator._apply`.

Listato 6.4 simulator.py: aggiunta di un'istruzione swap.

```
def swap(self, target: Qubit) -> None:
    self.parent._apply(
        qt.swap(), (1)
            [self.qubit_id, target.qubit_id] (2)
    )
```

(1) Per ottenere la matrice unitaria 4×4, dobbiamo impiegare `apply`; usiamo solo la funzione `qt.swap` che abbiamo già visto in questo capitolo.
(2) Dobbiamo fare in modo di passare gli indici per entrambi i qubit che vogliamo scambiare, in modo che `gate_expand_2toN` applichi correttamente la matrice unitaria per la nostra nuova istruzione `swap` allo stato di un intero registro.

Già che lavoriamo sul simulatore, aggiungiamo un'altra istruzione per mostrare il suo stato più facilmente, senza dover accedere ai suoi dettagli interni.

Listato 6.5 simulator.py: aggiunta di un'istruzione dump.

```
def dump(self) -> None:

    print(self.register_state)
```

In questo modo, possiamo chiedere al simulatore di aiutarci a eseguire il debug dei programmi quantistici, ma in un modo che può essere rimosso con sicurezza per i dispositivi che non lo supportano (per esempio, l'hardware quantistico effettivo).

SUGGERIMENTO

Ricordate che un qubit non è uno stato. Uno stato è solo un modo comodo per rappresentare come si comporterà il sistema quantistico.

Dopo queste modifiche, siamo pronti per utilizzare l'istruzione `swap`. Usiamola per ripetere l'esperimento in cui abbiamo scambiato due qubit per trasformarli dallo stato |0+⟩ allo stato |+0⟩.

SUGGERIMENTO

Come sempre, per i file di esempio completi, consultate l'archivio GitHub per questo libro: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp.

Listato 6.6 Test delle istruzioni di swap sullo stato |0+⟩.

```
>>> from simulator import Simulator
>>> sim = Simulator(capacity=2)
>>> with sim.using_register(n_qubits=2) as (you, eve): (1)
...     eve.h()
...     sim.dump()
...     you.swap(eve)
...     sim.dump()
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket (2)
Qobj data =
[[0.70710678]
 [0.70710678]
 [0.        ]
 [0.        ]]
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket (3)
Qobj data =
[[0.70710678]
 [0.        ]
 [0.70710678]

 [0.        ]]
```

(1) Dal momento che in questo capitolo lavoreremo molto con i registri multi-qubit, abbiamo aggiunto un nuovo metodo pratico che ci permette di allocare più qubit contemporaneamente.

(2) Il primo output proviene dalla nostra prima chiamata a `sim.dump` e conferma che `eve.h()` ha preparato i qubit nello stato |0+⟩.

(3) Dopo aver richiamato `you.swap(eve)`, il nostro qubit finisce nello stato |+⟩, e il qubit di Eve finisce nel modo in cui è iniziato il nostro: nello stato |0⟩.

Ottimo: ora abbiamo un modo per condividere i dati quantistici con Eve. Bene, almeno *fintantoché condividiamo un singolo dispositivo quantistico* in modo da poter applicare l'istruzione `swap` a entrambi i nostri qubit contemporaneamente.

Che cosa succede se vogliamo condividere le informazioni quantistiche tra più dispositivi? Fortunatamente, il quantum computing ci offre un modo per inviare qubit comunicando solo dati classici, purché entrambi iniziamo l'attività con un certo *entanglement* tra i nostri qubit. Come a molte altre cose del quantum computing, anche a questa tecnica è stato dato un nome stravagante: *teletrasporto quantistico*. Tuttavia, non lasciatevi ingannare dal nome: in buona sostanza, il teletrasporto utilizza ciò che abbiamo imparato nel Capitolo 4 per consentirci di condividere i dati quantistici in modo utile. La Figura 6.2 mostra l'elenco dei passaggi di un programma di teletrasporto.

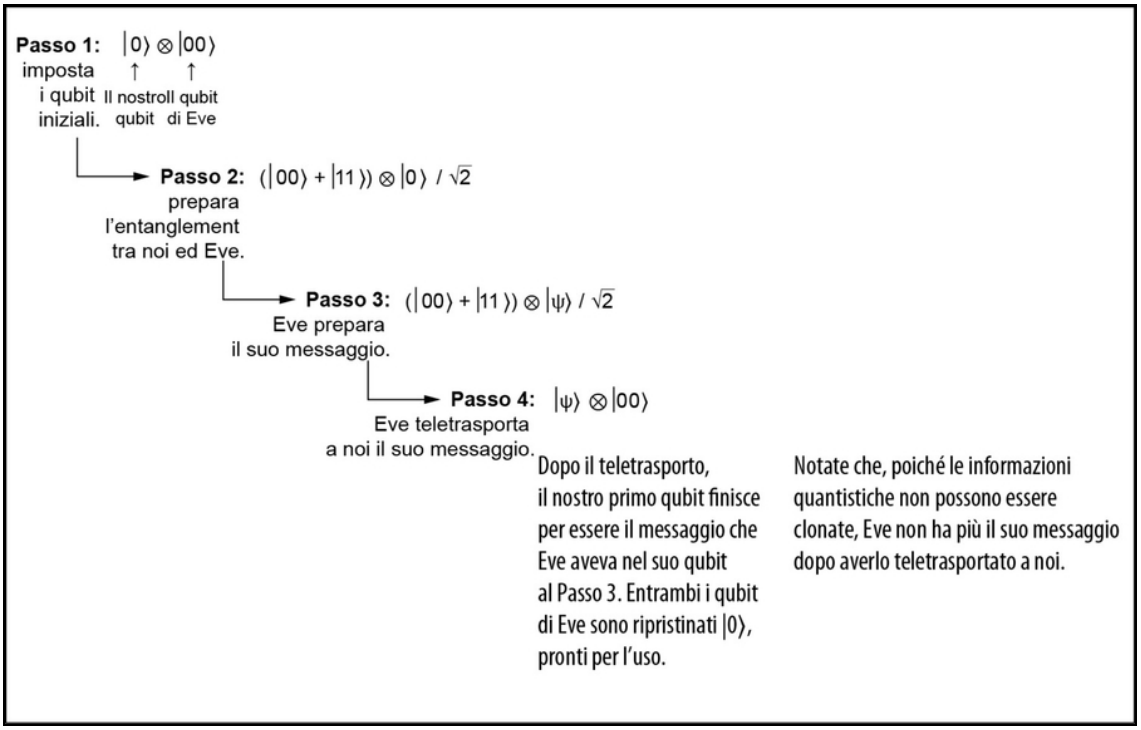


Figura 6.2 I passi del programma di teletrasporto. Eseguiamo la preparazione e l'entanglement di un registro di qubit, quindi Eve può preparare e teletrasportare a noi il suo stato. Notate che Eve non avrà più il suo stato, dopo il teletrasporto.

La cosa veramente interessante del teletrasporto è che, mentre noi ed Eve dobbiamo ancora fare alcune operazioni a due qubit sui nostri rispettivi qubit, Eve può decidere quali dati vuole inviarci anche *dopo* aver eseguito tali operazioni. Ciò significa che potremmo preparare i qubit entangled prima di dover scambiare dati quantistici e usarli solo se necessario.

Utilizzando il simulatore che abbiamo sviluppato negli ultimi capitoli, potremmo scrivere il teletrasporto con un programma quantistico come quello mostrato nel listato seguente.

Listato 6.7 Un programma di teletrasporto in Python.

```
def teleport(msg : Qubit, here : Qubit, there : Qubit) -> None:
    here.h()
    here.cnot(there)
    msg.cnot(here)
    msg.h()
    if msg.measure(): there.z()
    if here.measure(): there.x()
    msg.reset()
    here.reset()
```

Tuttavia, ci sono alcune nuove istruzioni in questo programma. Nel resto di questo capitolo vedremo tutto il necessario per iniziare a eseguire il teletrasporto quantistico utilizzando il nostro simulatore.

Quali altri porte a due qubit esistono?

Come potete immaginare, `swap` non è l'unica operazione a due qubit. Infatti, come illustrato nel Listato 6.7, per ottenere un teletrasporto funzionante, dobbiamo aggiungere al simulatore un'altra istruzione a due qubit chiamata `cnot`. L'istruzione `cnot` fa qualcosa di simile a swap, tranne per il fatto che scambia gli stati computazionali di base $|10\rangle$ e

|11 invece degli stati |01⟩ e |10. Un altro modo per dirlo è che `cnot` inverte il secondo qubit *controllato* nello stato del primo qubit, che è |1⟩. Da qui il nome `cnot`, un'abbreviazione di "NOT controllato".

> **SUGGERIMENTO**
>
> Spesso chiamiamo il primo qubit passato a un'istruzione `cnot` "qubit control" e il secondo qubit "qubit target". Come vedremo nel Capitolo 7, questi nomi nascondono alcuni significati subdoli.

Vediamo come funziona l'istruzione `cnot` applicandola a un esempio: lo stato |0⟩.

```
>>> import qutip as qt
>>> from qutip.qip.operations import hadamard_transform
>>> ket_0 = qt.basis(2, 0)
>>> ket_plus = hadamard_transform() * ket_0
>>> initial_state = qt.tensor(ket_plus, ket_0) (1)
>>> qt.cnot() * initial_state (2)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket (3)
Qobj data =
[[0.70710678]
 [0.        ]
 [0.        ]
 [0.70710678]]
>>> qt.cnot()
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper,
isherm = True (4)
Qobj data =
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 1. 0.]]
```

(1) Inizializza due qubit in modo che inizino nello stato |+0⟩ = (|00⟩ + |10⟩) / √2

(2) QuTiP fornisce la matrice unitaria per l'istruzione `cnot`, tramite la funzione `qt.cnot`.

(3) L'istruzione `cnot` lascia i nostri qubit nello stato (|00⟩ + |11⟩) / √2.

(4) La matrice di `cnot` mappa lo stato computazionale di base |10⟩ su |11⟩ e viceversa, proprio come ci aspettavamo dalla descrizione.

> **NOTA**

L'istruzione `cnot` non è la stessa cosa delle istruzioni `if` dei linguaggi di programmazione classici, in quanto un'istruzione `cnot` preserva la sovrapposizione. Se volessimo usare un'istruzione `if`, dovremmo misurare il qubit control, provocando il collasso di qualsiasi sovrapposizione su tale qubit. In realtà useremo sia istruzioni `cnot` sia istruzioni `if` condizionate sui risultati delle misurazioni quando scriveremo il nostro programma di teletrasporto alla fine di questo capitolo, perché entrambe possono essere utili! Nei Capitoli 8 e 9 vedremo più in dettaglio come le operazioni controllate differiscono dalle istruzioni `if`.

La Figura 6.3 mostra come agisce in generale l'istruzione `cnot` su stati a due qubit. Per ora, però, riconosciamo lo stato di output che abbiamo ottenuto nel precedente frammento di codice applicando `cnot` a due qubits nello stato $|0\rangle$ come l'`entangled` di cui avevamo bisogno per giocare a CHSH nel Capitolo 4, $(|00\rangle + |11\rangle) / \sqrt{2}$.

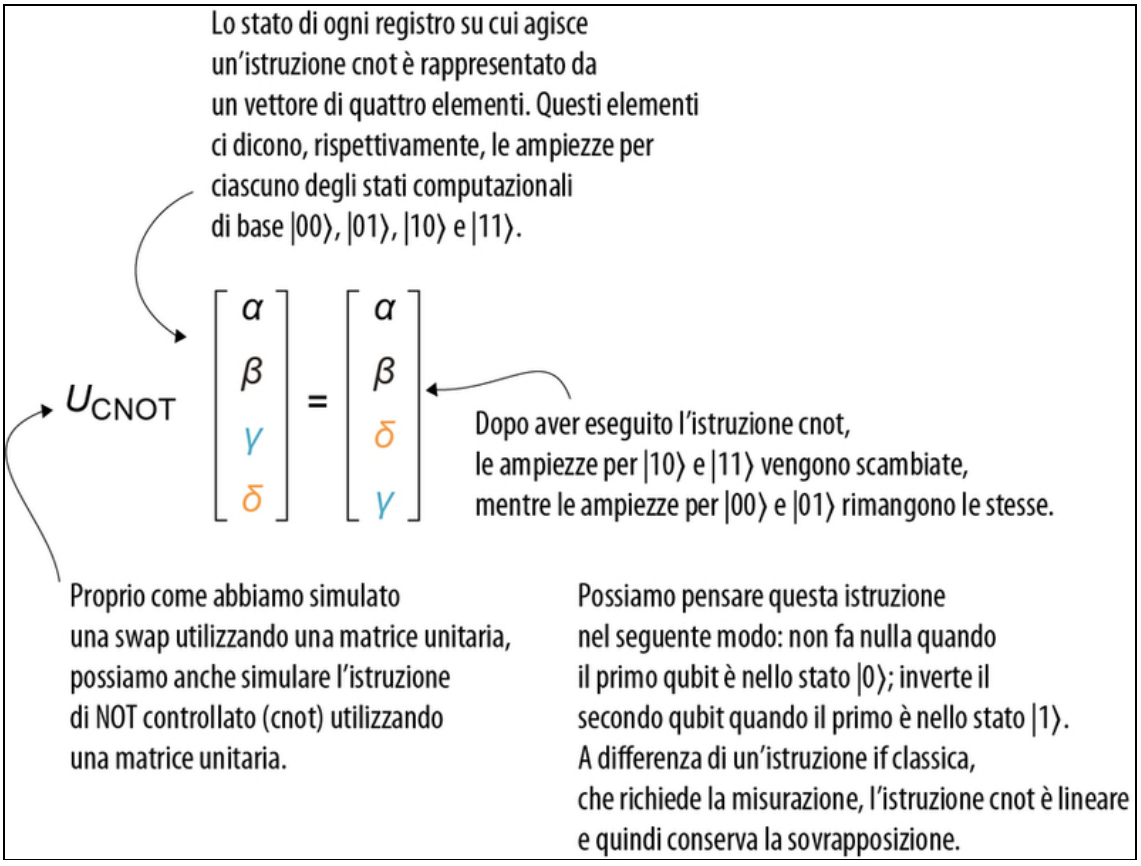


Figura 6.3 L'operazione a due qubit cnot applica un'operazione NOT condizionata allo stato di un qubit control.

Ciò significa che abbiamo tutto ciò di cui abbiamo bisogno per scrivere un programma quantistico che esegua l'*entanglement* di due qubit che partono nello stato |00⟩. Tutto quello che dobbiamo fare è aggiungere l'istruzione `cnot` al nostro simulatore, nello stesso modo in cui abbiamo aggiunto `swap` in precedenza.

Listato 6.8 simulator.py: aggiunta dell'istruzione cnot.

```
def cnot(self, target: Qubit) -> None:
    self.parent._apply(
        qt.cnot(),
        [self.qubit_id, target.qubit_id]
    )
```

```
Ora possiamo scrivere un programma per preparare due qubit in una coppia
entangled:
>>> from simulator import Simulator
>>> sim = Simulator(capacity=2)
>>> with sim.using_register(2) as (you, eve):
...     eve.h()
...     eve.cnot(you)
...     sim.dump()
...
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[0.70710678]
 [0.        ]
 [0.        ]
 [0.70710678]]
```

A questo punto, è utile fermarsi un momento a riflettere su quello che abbiamo appena fatto. Nel Capitolo 4, quando abbiamo simulato il gioco CHSH con Eve, abbiamo dovuto "barare" supponendo che noi ed Eve potremmo avere accesso a due qubit che partono "magicamente" in uno stato *entangled* (|00⟩ + |11⟩) / √2. Ora, però, vediamo esattamente come noi ed Eve siamo in grado di preparare tale *entanglement* eseguendo un altro programma quantistico prima di giocare a CHSH. L'istruzione `h` prepara la sovrapposizione di cui abbiamo bisogno, mentre la nuova istruzione `cnot` ci permette di

preparare l'*entanglement* con Eve. Questo *entanglement* "condivide" tale sovrapposizione tra i nostri due qubit.

Proprio come preparare un "accordo" tra noi ed Eve è stato il modo in cui ci siamo preparati per il gioco CHSH nel Capitolo 4, l'*entanglement* è il primo passo di cui abbiamo bisogno per permettere a Eve di teletrasportarci i suoi dati quantistici. Questo rende `cnot` un'istruzione molto importante per il futuro.

Tornando a Eve, però, il passo successivo per consentirle di teletrasportarci i suoi dati prevede l'uso di una delle quattro operazioni mono-qubit per *decodificare* i dati quantistici che ci invia (vedi la Figura 6.2). Vediamo come fare.

Tutte le rotazioni mono-qubit

L'ultima cosa di cui abbiamo bisogno per programmare il teletrasporto quantistico è applicare una correzione basata su alcuni dati classici che Eve ci invia. Per fare ciò, abbiamo bisogno di un paio di nuove istruzioni mono-qubit. Per questo, è utile rivedere le immagini che abbiamo usato per rappresentare le istruzioni quantistiche come rotazioni, perché potremmo aver "barato" un po'. Finora abbiamo rappresentato i nostri qubit come posizioni qualsiasi su un cerchio, ma in realtà ci manca una dimensione per il nostro modello di qubit. Lo stato di un *singolo* qubit è rappresentato da un qualsiasi punto sulla superficie di una sfera, generalmente chiamata *sfera di Bloch*.

Solo qubit singoli!

Questo (e il precedente) modo di visualizzare lo stato di un qubit funziona *solo* se quel qubit non è *entangled* con nessun altro qubit. Un altro modo per dirlo è che non possiamo rappresentare facilmente uno stato multi-qubit. Anche provare a visualizzare lo stato di un registro di due qubit con *entanglement* ci obbligherebbe a disegnare immagini in sette dimensioni. Può darsi che "7D" sia uno slogan ottimo

per pubblicizzare un giro alle cascate del Niagara, ma è molto difficile disegnare immagini utili in questo modo.

Il cerchio che ormai conosciamo bene era in realtà solo una "fetta" di una sfera, e tutte le rotazioni che abbiamo svolto individuavano stati sempre su quel cerchio. La Figura 6.4 mostra il confronto tra il precedente modello di qubit e la sfera di Bloch.

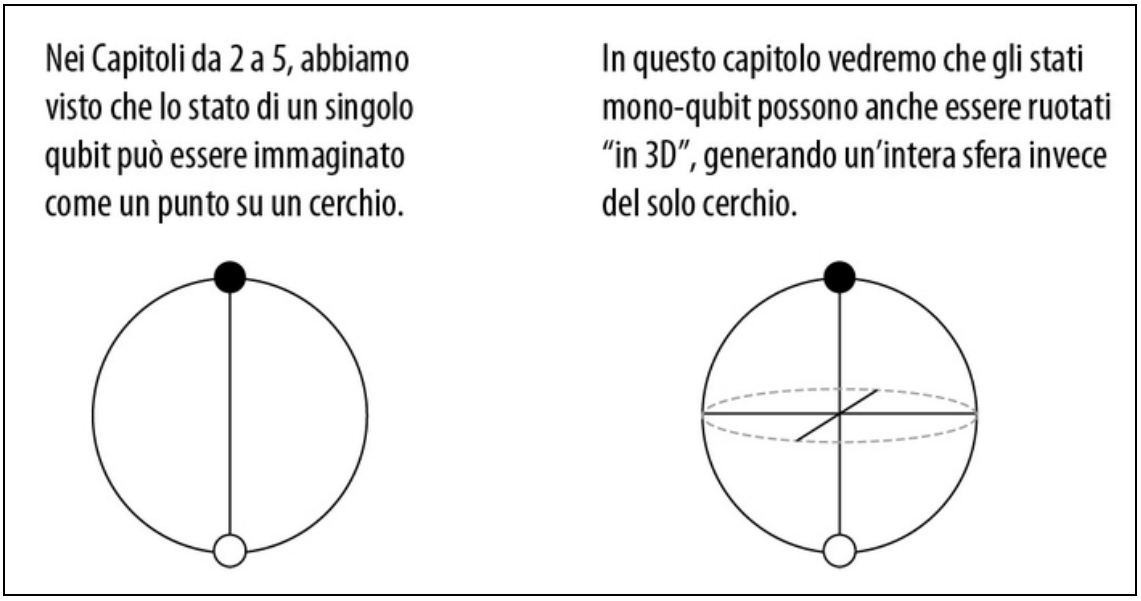


Figura 6.4 Confronto fra il nostro modello precedente di un qubit (un punto su un cerchio) e la sfera di Bloch. La sfera di Bloch è quindi un modello più generale per lo stato di un singolo qubit. Abbiamo bisogno di un'altra dimensione per catturare lo stato di un qubit rappresentato da un vettore di numeri complessi, ma funziona solo per qubit singoli.

SUGGERIMENTO

Potreste aver dedotto, dal fatto che abbiamo mostrato l'asse Z e l'asse X, che l'asse Y era nascosto da qualche parte. In effetti, quando passiamo da un cerchio a una sfera, l'asse che esce dalla pagina viene spesso chiamato "asse Y".

Quando abbiamo introdotto per la prima volta la rappresentazione vettoriale di uno stato del qubit nel Capitolo 2, ricorderete che le ampiezze di ciascun vettore erano numeri complessi. Nel resto di

questo capitolo vedremo che, in generale, quando usiamo istruzioni di rotazione per trasformare lo stato di un singolo qubit, otteniamo numeri complessi. I numeri complessi sono uno strumento incredibilmente utile per tenere traccia delle rotazioni, e quindi svolgono un ruolo importante nel quantum computing. Principalmente ci aiutano a capire gli angoli e le fasi tra i diversi stati quantistici. Non preoccupatevi se siete un po' arrugginiti con numeri complessi: avrete molte possibilità di esercitarvi in tutto il resto del libro.

Correlare le rotazioni alle coordinate: le operazioni di Pauli

Nella Figura 6.5, ripassiamo rapidamente un paio di operazioni mono-qubit che abbiamo visto finora: `x` e `ry`.

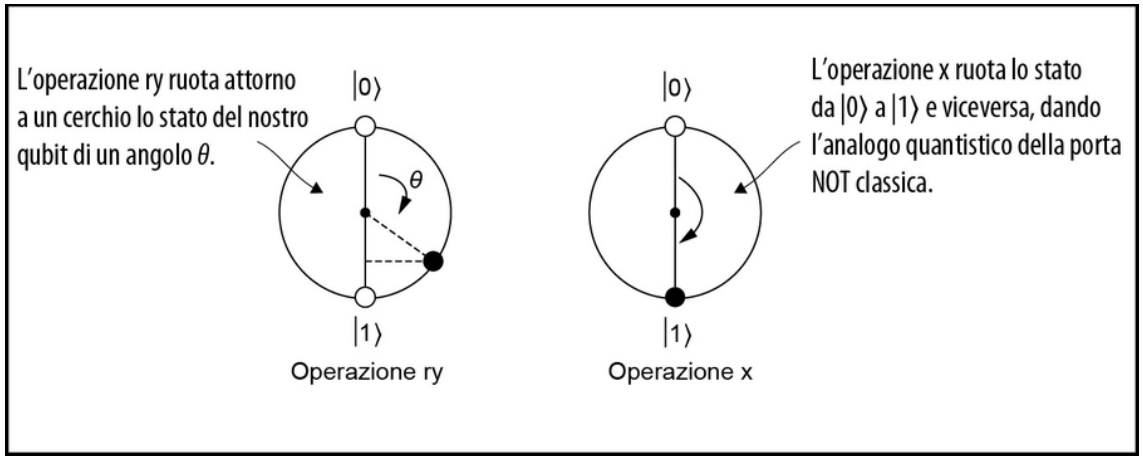


Figura 6.5 Le operazioni x e ry su un qubit. Abbiamo visto entrambe le operazioni nei capitoli precedenti. L'istruzione `ry` ruota lo stato del nostro qubit di un angolo theta; x scambia il qubit fra gli stati |0⟩ e |1⟩.

Ora che sappiamo che lo stato del nostro qubit può essere ruotato sulla superficie di una sfera, quali altre rotazioni possono aiutarci a ruotare lo stato fuori dal piano? Possiamo aggiungere una rotazione attorno alla linea tra gli stati |+⟩ e |-⟩. Questa linea è convenzionalmente

chiamata asse *X* per distinguerlo dall'asse *Z* che collega gli stati $|0\rangle$ e $|1\rangle$. La funzione `rx` di QuTiP ci fornisce un oggetto `Qobj` che incapsula la matrice di rotazione per una rotazione attorno all'asse *X*.

Listato 6.9 Utilizzo della funzione qt.sigmaz di QuTiP.

```
>>> import qutip as qt
>>> import numpy as np
>>> qt.rx(np.pi).tidyup() (1)
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm =
False
Qobj data =
[[ 0.+0.j 0.-1.j] (2)
 [ 0.-1.j 0.+0.j]]
```

(1) Le istanze di `Qobj` hanno un metodo `tidyup` per aiutare a rendere le matrici più leggibili, poiché l'aritmetica in virgola mobile sui computer classici può produrre piccoli errori.

(2) Fino a un coefficiente di -*i* (scritto in Python come `-1j`), la rotazione di 180° attorno all'asse *X* produce l'istruzione `x` (NOT) che abbiamo visto per la prima volta nel Capitolo 2.

SUGGERIMENTO

In Python, il numero complesso *i* è rappresentato da `1.0j`: 1 per *j*, che è il nome impiegato talvolta per il numero immaginario.

Questo frammento illustra un concetto di molto importante: l'operazione `x` è esattamente ciò che otteniamo ruotando attorno all'asse *X* di un angolo di 180° (π).

DEFINIZIONE

Come abbiamo scritto nelle note del Listato 6.1, possiamo controllare che `qt.rx(np.pi)` si discosti effettivamente di un fattore -*i* da `qt.sigmax()`. Quel fattore è un esempio di una *fase globale*. Come vedremo tra poco, le fasi globali *non possono* influenzare i risultati della misurazione. Pertanto, `qt.rx(np.pi)` e `qt.sigmax()` sono matrici unitarie distinte, che rappresentano la stessa operazione. Faremo più pratica con le fasi globali e locali nei Capitoli 7 e 8.

Per analogia, chiamiamo l'operazione di rotazione di 180° attorno all'asse *Z* operazione `z`. Nel Capitolo3, QuTiP ha fornito la funzione `qt.sigmax` che simula l'istruzione `x`. Allo stesso modo, `qt.sigmaz` fornisce le matrici unitarie necessarie per simulare le istruzioni `z`. Il prossimo listato mostra un esempio di utilizzo di `qt.sigmaz`. Notate che abbiamo incluso il coefficiente (cioè la fase globale) di $-i$ moltiplicando immediatamente per i; questo funziona in quanto $-i \times i = -(-1) = 1$.

Listato 6.10 Utilizzo delle funzioni QuTiP qt.rz e qt.sigmaz.

```
>>> import qutip as qt
>>> import numpy as np
>>> 1j * qt.rz(np.pi).tidyup() (1)
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm =
True
Qobj data =
[[ 1.  0.]
 [ 0. -1.]]
>>> qt.sigmaz() (2)
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm =
True
Qobj data =
[[ 1.  0.]
 [ 0. -1.]]
```

(1) Annullare la fase globale in questo modo facilita la lettura dell'output.

(2) Come promesso, fino al coefficiente $-i$, l'istruzione `z` applica una rotazione di 180° attorno all'asse *Z*.

Allo stesso modo in cui l'operazione `x` inverte tra $|0\rangle$ e $|1\rangle$ lasciando immutati $|+\rangle$ e $|-\rangle$, l'operazione `z` inverte tra $|+\rangle$ e $|-\rangle$ lasciando immutati $|0\rangle$ e $|1\rangle$.

La Tabella 6.1 mostra una tabella di verità come quelle che abbiamo creato per l'operazione di Hadamard nel Capitolo 2. Osservando la tabella di verità, possiamo confermare che, se eseguiamo un'operazione due volte su qualsiasi stato di input, torneremo al punto di partenza. In altre parole, `z` corrisponde all'operazione di identità $\mathbb{1}$ nello stesso modo in cui $X^2 = \mathbb{1}$.

Tabella 6.1 Rappresentazione dell'istruzione z come tabella.

Stato di input	Stato di output
$\|0\rangle$	$\|0\rangle$
$\|1\rangle$	$-\|1\rangle$
$\|+\rangle$	$\|-\rangle$
$\|-\rangle$	$\|+\rangle$

NOTA

Abbiamo elencato quattro righe nella Tabella 6.1, ma abbiamo bisogno di due sole righe per specificare completamente come *Z* agisce per qualsiasi input. Le altre due righe servono a sottolineare che possiamo scegliere tra la definizione di *Z* con la sua azione su $|0\rangle$ e $|1\rangle$ o con la sua azione su $|+\rangle$ e $|-\rangle$.

ESERCIZIO 6.2 FARE PRATICA CON RZ E Z

Supponiamo di preparare un qubit nello stato $|-\rangle$ e di applicargli una rotazione `z`. Se misurate lungo l'asse *X*, che cosa ottenete? Che cosa misurate se applicate due rotazioni `z`? Se implementate le stesse due rotazioni con `rz`, quali angoli dovreste usare?

Possiamo definire anche un'altra rotazione allo stesso modo: la rotazione attorno a un asse che "esce dalla pagina". Questo asse collega gli stati $(|0\rangle + i|1\rangle) / \sqrt{2} = \mathrm{Rx}(\pi / 2)\ |0\rangle$ e $(|0\rangle - i|1\rangle) / \sqrt{2} = \mathrm{Rx}(\pi / 2)\ |1\rangle$ ed è convenzionalmente chiamato asse *Y*. Una rotazione di 180° attorno all'asse *Y* inverte le etichette dei bit ($|0\rangle \Leftrightarrow |1\rangle$) e della fase ($|+\rangle \Leftrightarrow |-\rangle$) ma lascia immutati i due stati lungo l'asse *Y*.

ESERCIZIO 6.3 TABELLA DI VERITÀ PER SIGMAY

Utilizzate la funzione `qt.sigmay()` per creare una tabella simile alla Tabella 6.1, ma per l'istruzione `y`.

DEFINIZIONE

Insieme, le tre matrici *X*, *Y* e *Z* che rappresentano le operazioni `x`, `y` e `z` sono chiamate *matrici di Pauli*, in onore del fisico Wolfgang Pauli. Talvolta viene

inclusa anche la matrice identità, $\mathbb{1}$, che rappresenta l'operazione "non fare nulla" o identità.

Giocare a sasso-carta-forbici con le matrici Pauli

Le matrici Pauli hanno tutta una serie di proprietà utili che useremo nel resto del libro. Molte di queste proprietà aiutano a elaborare diverse equazioni che coinvolgono gli operatori di Pauli. Per esempio, se moltiplichiamo *X* per *Y*, otteniamo *iZ*, ma se moltiplichiamo *Y* per *X*, otteniamo *-iZ*.

QuTiP può aiutare a esplorare che cosa succede quando moltiplichiamo le matrici di Pauli fra loro: per esempio, moltiplicando *X* e *Y* insieme in entrambi i possibili ordini:

```
>>> import qutip as qt
>>> qt.sigmax() * qt.sigmay()
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm =
False
Qobj data =
[[0.+1.j 0.+0.j]
 [0.+0.j 0.-1.j]]
>>> qt.sigmay() * qt.sigmax()
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm =
False
Qobj data =
[[0.-1.j 0.+0.j]
 [0.+0.j 0.+1.j]]
```

Allo stesso modo, $YZ = iX$ e $ZX = iY$, ma $ZY = -iX$ e $XZ = -iY$. Un modo per ricordarlo è pensare a *X*, *Y* e *Z* come a un gioco di sasso-carta-forbici: *X* "batte" *Y*, *Y* "batte" *Z* e *Z* "batte" *X*.

Possiamo considerare queste matrici come se stabilissero una sorta di sistema di coordinate per gli stati dei qubit, chiamato *sfera di Bloch*. Come vediamo nella Figura 6.6, gli assi *X* e *Z* formano il cerchio che abbiamo visto finora nel libro, mentre l'asse *Y* esce dalla pagina.

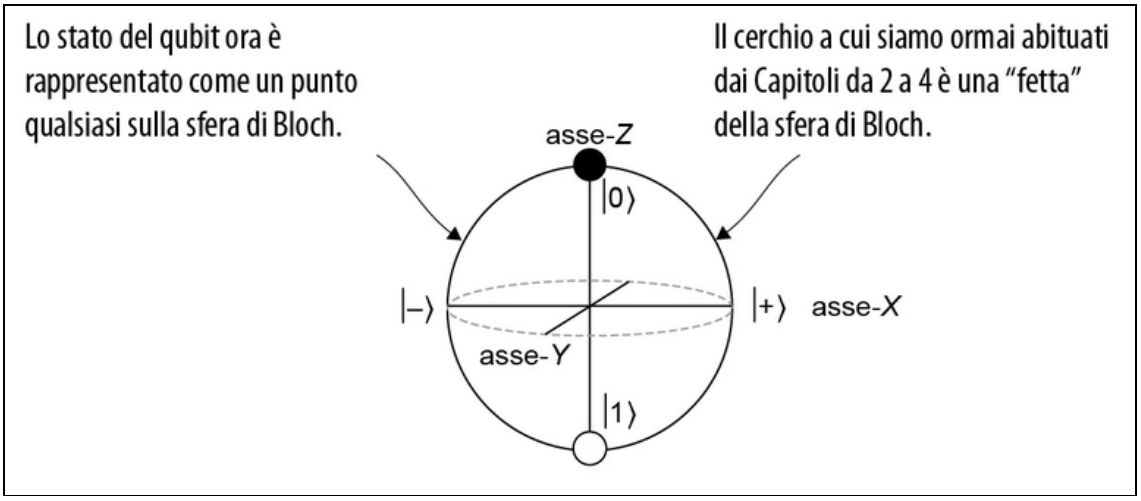


Figura 6.6 La sfera di Bloch, in tutto il suo splendore sferico. Qui, ciascuno degli assi è etichettato con l'operatore di Pauli corrispondente, che rappresenta le rotazioni attorno a quell'asse.

Descrivere gli stati con le misurazioni di Pauli

Qualsiasi stato mono-qubit può essere interamente specificato fino a una fase globale dalle probabilità di misurazione per *X*, *Y* e *Z*. Cioè, se vi diciamo la probabilità di ottenere un risultato "1" per ciascuna delle tre misurazioni di Pauli che potremmo eseguire, potete usare tale informazione per scrivere un vettore di stato identico al nostro, fino alla fase globale. Ciò rende l'analogia con i punti in tre dimensioni utile per immaginare gli stati mono-qubit.

Una questione di i

Gli stati alle estremità dell'asse *Y* sono solitamente etichettati $|i\rangle$ e $|\text{-}i\rangle$ ma raramente sono usati da soli. Resteremo fedeli agli stati etichettati che abbiamo usato finora: $|0\rangle$, $|1\rangle$, $|+\rangle$ e $|\text{-}\rangle$.

Con questa immagine in mente, è più facile capire perché alcune rotazioni non influiscono sui risultati delle misurazioni. Per esempio, come illustrato nella Figura 6.7, l'immagine della sfera di Bloch ci aiuta a capire che cosa succede se ruotiamo $|0\rangle$ attorno all'asse *Z*.

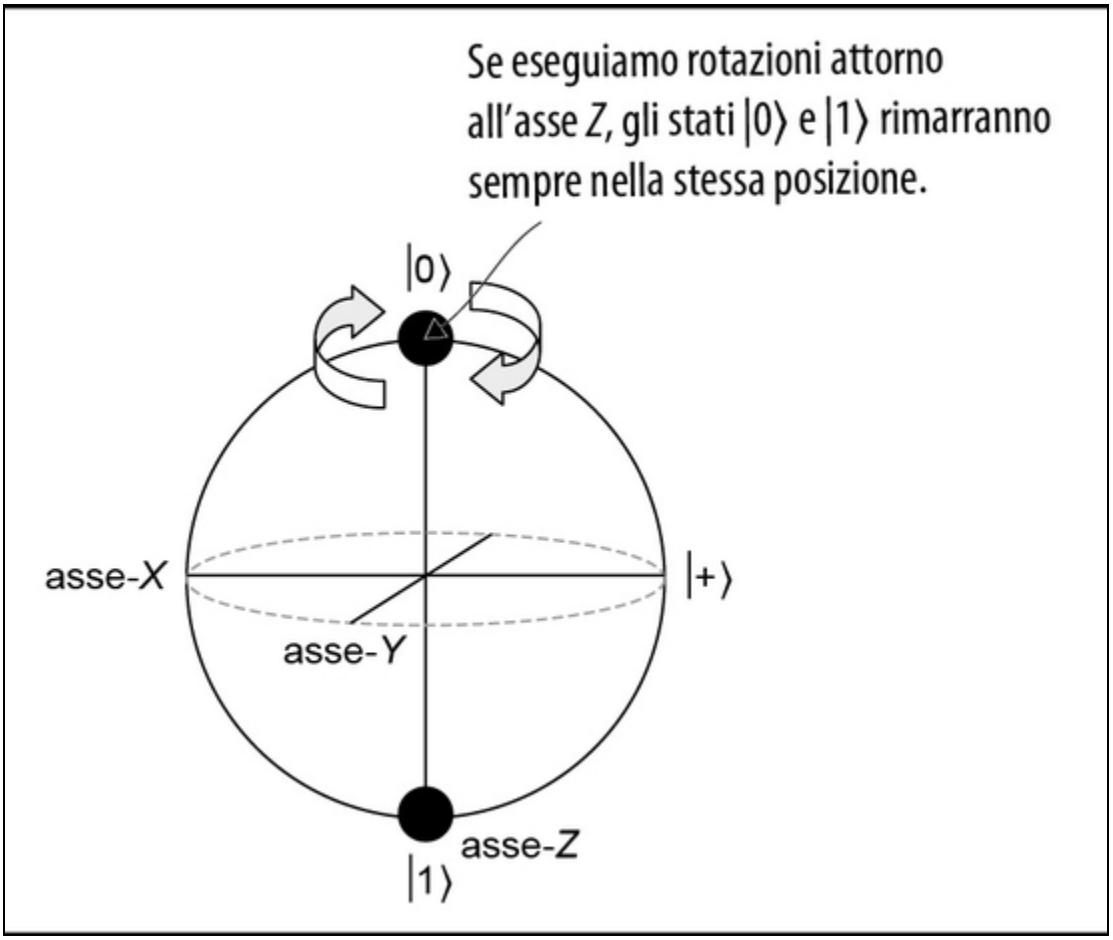


Figura 6.7 La sfera di Bloch illustra come la rotazione rz lasci invariato lo stato |0⟩.

Allo stesso modo in cui il Polo Nord rimane nello stesso punto indipendentemente da quanto ruotiamo il globo, se ruotiamo uno stato attorno a un asse parallelo a quello stato, non avremo alcun effetto osservabile sul nostro qubit. Possiamo anche vedere questo effetto in termini matematici.

Listato 6.11 Lo stato |0⟩ non è influenzato dalle rotazioni rz.

```
>>> ket0 = qt.basis(2, 0) (1)
>>> ket_psi = qt.rz(np.pi / 3) * ket0 (2)
>>> ket_psi (3)
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[ 0.8660254-0.5j]
 [ 0.0000000+0.j ]]
>>> bra0 = ket0.dag() (4)
```

```
>>> bra0
Quantum object: dims = [[1], [2]], shape = (1, 2), type = bra
Qobj data = (5)
[[ 1. 0.]]
>>> np.abs((bra0 * ket_psi)[0, 0]) ** 2 (6)

1.0
```

(1) Definisce una variabile per rappresentare lo stato $|0\rangle$.
(2) Introduce un nuovo stato $|\psi\rangle$ che è una rotazione di 60° (π / 3 in radianti) di $|0\rangle$ attorno all'asse *Z*.
(3) Lo stato risultante è $|\psi\rangle = [\cos(60° / 2) - i \sin(60° / 2)] |0\rangle = [\sqrt{3} / 2 - i / 2] |0\rangle$.
(4) Ricordate che in QuTiP, scriviamo l'operatore "pugnale" richiamando il metodo `.dag` delle istanze di `Qobj`, in questo caso passandogli il vettore riga per $\langle 0|$.
(5) Prendendo il prodotto interno $\langle 0|\psi\rangle$, possiamo calcolare la regola di Born $\Pr(0|\psi) = |\langle 0|\psi\rangle|^2$. Notate che dobbiamo indicizzare per [0, 0], in quanto QuTiP rappresenta il prodotto interno di $|0\rangle$ per $|\psi\rangle$ come una matrice 1×1.
(6) Come prima, la probabilità di osservare uno "0" durante la misurazione lungo l'asse *Z* non è cambiata.

> **SUGGERIMENTO**
>
> Quando si scrivono gli stati, $|\psi\rangle$ è spesso usato come nome arbitrario, così come la lettera *x* è spesso utilizzata in algebra per rappresentare una variabile arbitraria.

In generale, possiamo sempre moltiplicare uno stato per un numero complesso il cui valore assoluto è 1 senza modificare le probabilità di qualsiasi misurazione. Qualsiasi numero complesso $z = a + bi$ può essere scritto come $z = re^{i}q$ per i numeri reali *r* e q, dove *r* è il valore assoluto di *z* e q è un angolo. Quando $r = 1$, abbiamo un numero della forma $e^{i}q$, che chiamiamo *fase*. Diciamo quindi che moltiplicando lo stato per una fase si applica a tale stato una fase globale.

> **NOTA**

Nessuna fase globale può mai essere rilevata da nessuna misurazione.

ESERCIZIO 6.4 VERIFICARE CHE L'APPLICAZIONE DI RZ NON CAMBIA |0⟩

Abbiamo solo verificato che una probabilità della misurazione è sempre la stessa, ma forse le probabilità sono cambiate per le misurazioni *X* o *Y*. Per verificare completamente che la fase globale non cambi nulla, preparate lo stesso stato e la stessa rotazione del Listato 6.11 e verificate che le probabilità della misurazione dello stato lungo l'asse *X* o *Y* non vengano modificate applicando un'istruzione `rz`.

Gli stati $|\psi\rangle$ ed $e^{i}q|\psi\rangle$ sono in ogni modo concepibile due modi diversi di descrivere esattamente lo stesso stato. Non ci sono misurazioni che possiamo fare, *anche in linea di principio*, per conoscere le fasi globali. D'altra parte, abbiamo visto che possiamo distinguere stati come $|+\rangle = (|0\rangle + |1\rangle) / \sqrt{2}$ e $|-\rangle = (|0\rangle - |1\rangle) / \sqrt{2}$ che differiscono solo nella *fase locale* dello stato computazionale di base $|1\rangle$.

Facendo un passo indietro, riassumiamo ciò che abbiamo imparato finora sulle istruzioni `x`, `y` e `z` e sulle matrici di Pauli che usiamo per simulare tali istruzioni. Abbiamo visto che l'istruzione `x` scambia gli stati $|0\rangle$ e $|1\rangle$, mentre l'istruzione `z` scambia gli stati $|+\rangle$ e $|-\rangle$. In altre parole, l'istruzione `x` inverte i bit, mentre l'istruzione `z` inverte le fasi.

Osservando la sfera di Bloch, possiamo vedere che la rotazione attorno all'asse *Y* dovrebbe fare entrambe le cose. Possiamo anche vederlo tramite $Y = -iXZ$, poiché è semplice da verificare con QuTiP. Nella Tabella 6.2 riassumiamo il significato di ogni istruzione di Pauli.

Tabella 6.2 Matrici di Pauli come bit e inversioni di fase.

Istruzione	Matrice di Pauli	Inverte i bit ($\|0\rangle \Leftrightarrow \|1\rangle$)?	Inverte le fasi ($\|+\rangle \Leftrightarrow \|-\rangle$)?
(nessuna istruzione)		No	No

`x`	X	Sì	No
`y`	Y	Sì	Sì
`z`	Z	No	Sì

ESERCIZIO 6.5 HMM, CHE COSA DICE BELL?

Lo stato $(|00\rangle + |11\rangle) / \sqrt{2}$ stato che abbiamo visto un paio di volte non è l'unico esempio di stato di *entanglement*. In effetti, se scegliamo a caso uno stato di due qubit, quasi certamente subirà *entanglement*. Proprio come la base computazionale $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ è un insieme particolarmente utile di stati non *entangled*, esiste un insieme di quattro particolari stati *entangled* conosciuti come la *base di Bell* dal fisico John Stewart Bell:

Nome	Espansione nella base computazionale
$\|\beta 00\rangle$	$(\|00\rangle + \|11\rangle) / \sqrt{2}$
$\|\beta 01\rangle$	$(\|00\rangle - \|11\rangle) / \sqrt{2}$
$\|\beta 10\rangle$	$(\|01\rangle + \|10\rangle) / \sqrt{2}$
$\|\beta 11\rangle$	$(\|01\rangle - \|10\rangle) / \sqrt{2}$

Usando ciò che avete imparato sulle istruzioni `cnot` e le istruzioni di Pauli (`x`, `y` e `z`), scrivete dei programmi per preparare ciascuno dei quattro stati di Bell della tabella.

Suggerimento: la Tabella 6.2 dovrebbe essere molto utile in questo esercizio.

Concludiamo la nostra discussione sulle operazioni mono-qubit aggiungendo alla nostra interfaccia e simulatore `Qubit` le istruzioni per le operazioni `x`, `y` e `z`. Per farlo, possiamo implementare le istruzioni di rotazione `rx`, `ry` e `rz` utilizzando le corrispondenti funzioni QuTiP `qt.rx`, `qt.ry` e `qt.rz` per ottenere le copie delle matrici unitarie di cui abbiamo bisogno per simulare ciascuna istruzione. Ecco come possiamo farlo nel nostro simulatore.

Listato 6.12 simulator.py: aggiunta di tutte le rotazioni di Pauli.

```
def rx(self, theta: float) -> None:
    self.parent._apply(qt.rx(theta), [self.qubit_id])
def ry(self, theta: float) -> None:
    self.parent._apply(qt.ry(theta), [self.qubit_id])
def rz(self, theta: float) -> None:
    self.parent._apply(qt.rz(theta), [self.qubit_id])
def x(self) -> None:
    self.parent._apply(qt.sigmax(), [self.qubit_id])
def y(self) -> None:
    self.parent._apply(qt.sigmay(), [self.qubit_id])
def z(self) -> None:

    self.parent._apply(qt.sigmaz(), [self.qubit_id])
```

QuTiP utilizza la notazione σx invece di *X* per le matrici di Pauli. Usando questa notazione, la funzione `sigmax()` restituisce un nuovo `Qobj` che rappresenta la matrice di Pauli *X*. In questo modo, possiamo implementare le istruzioni `x`, `y` e `z` corrispondenti a ciascuna matrice di Pauli.

"Nessuno di noi, purtroppo, è in grado di descrivere Matrix agli altri. Dovrai scoprire con i tuoi occhi che cos'è"

Abbiamo parlato molto delle matrici nella Parte I di questo libro. Potremmo essere tentati di dire che la programmazione quantistica riguarda esclusivamente le matrici e che i qubit sono in realtà solo vettori. In realtà, però, le matrici sono il modo in cui *simuliamo* ciò che fa un dispositivo quantistico. Come vedremo meglio nella Parte II, i programmi quantistici non manipolano affatto matrici e vettori: manipolano dati classici, come quali istruzioni inviare a un dispositivo quantistico e che cosa fare con i dati che otteniamo da tali dispositivi. Per esempio, se eseguiamo un'istruzione su un dispositivo, non esiste un modo semplice per vedere quale matrice dovremmo usare per simulare tale istruzione, piuttosto dovremo ricostruire quella matrice tramite molte misurazioni ripetute, utilizzando una tecnica chiamata *tomografia di processo*.

Quando scriviamo una matrice, nel codice o su un foglio di carta, simuliamo implicitamente un sistema quantistico. Se pensate che questo "sia davvero troppo", non preoccupatevi; tutto avrà molto più senso proseguendo la lettura del libro.

Teletrasporto

OK, ora abbiamo tutto ciò di cui abbiamo bisogno per scoprire l'aspetto del teletrasporto sotto forma di programma quantistico. In breve, la Figura 6.8 è ciò che vogliamo ottenere da questo programma.

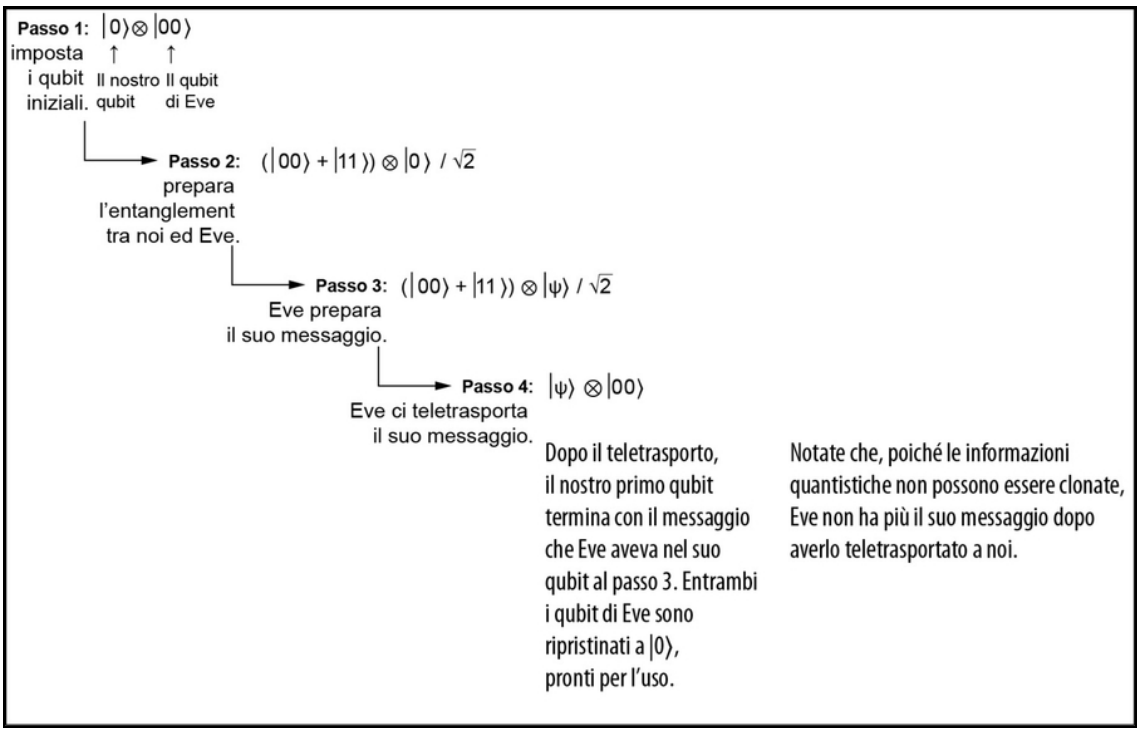


Figura 6.8 I passaggi del programma di teletrasporto.

Supporremo che possiate preparare alcuni qubit *entangled* mentre sono nello stesso dispositivo e che noi ed Eve abbiamo un mezzo di comunicazione classico che possiamo usare per segnalare la giusta correzione da usare. Ora possiamo utilizzare le funzionalità che abbiamo aggiunto al nostro simulatore in questo capitolo per implementare il programma di teletrasporto.

Listato 6.13 teleport.py: teletrasporto quantistico in poche righe.

```
from interface import QuantumDevice, Qubit
from simulator import Simulator
def teleport(msg: Qubit, here: Qubit, there: Qubit) -> None: (1)
    here.h() (2)
    here.cnot(there) (3)
    # ... (4)
    msg.cnot(here) (5)
```

```
    msg.h()
    if msg.measure(): there.z() (6)
    if here.measure(): there.x()
    msg.reset() (7)

    here.reset()
```

(1) La funzione `teleport` accetta come input due qubit: il qubit che vogliamo spostare (`msg`) e dove vogliamo che venga spostato (`there`). Abbiamo anche bisogno di un qubit temporaneo, che chiamiamo `here`. Presumiamo per convenzione che sia `here` sia `there` inizino nello stato $|0\rangle$.

(2) Dobbiamo iniziare con un *entanglement* tra `here` e `there`. Possiamo usare la nostra vecchia conoscenza, l'istruzione `h`, insieme al nostro nuovo amico, l'istruzione `cnot`.

(3) L'unica istruzione in questo programma che deve agire sia su `here` e `there`. Dopo averla eseguita, possiamo inviare a Eve il nostro qubit ed entrambi possiamo eseguire il resto del programma solo con la comunicazione classica.

(4) A questo punto del programma, `here` e `there` sono nello stato $(|00\rangle + |11\rangle) / \sqrt{2}$ che abbiamo visto per la prima volta nel Capitolo 4.

(5) Esegue il programma che abbiamo usato per preparare lo stato $(|00\rangle + |11\rangle) / \sqrt{2}$, ma sui qubit `msg` e `here` che risiedono interamente sul nostro dispositivo. Possiamo pensare di eseguire una preparazione all'indietro come una sorta di misurazione, in modo tale che questi passaggi ci servano per misurare il messaggio quantistico che stiamo cercando di inviare a Eve in un *entangled*.

(6) Quando eseguiamo effettivamente la misurazione, otteniamo i dati classici da inviare a Eve. Una volta che lei ha quei dati, può usare le istruzioni `x` e `z` per decodificare il messaggio quantistico.

(7) Ora che abbiamo finito con i nostri qubit, è bene rimetterli a $|0\rangle$, in modo che siano pronti per essere riutilizzati. Ciò non influisce sullo

stato di `there`, poiché abbiamo ripristinato solo i nostri qubit, non quelli che abbiamo dato a Eve.

Che cosa abbiamo fatto?
Supponiamo di non aver bisogno di inviare a Eve un risultato della misurazione classico, come parte del teletrasporto. In tal caso, potremmo usare il teletrasporto per inviare dati sia classici sia quantistici più velocemente della velocità della luce. Proprio come non potevamo comunicare con Eve quando abbiamo giocato al gioco CHSH nel Capitolo 5, la velocità della luce fa sì che dobbiamo comunicare con Eve in modo classico per usare l'*entanglement* per inviare i dati quantistici. In entrambi i casi, l'*entanglement* può aiutarci a comunicare, ma da solo non ci consente di comunicare: abbiamo sempre bisogno anche di un altro tipo di comunicazione.

Per vedere come funziona, possiamo preparare qualcosa sul nostro qubit e inviarlo a Eve; poi lei può annullare la nostra preparazione sul suo qubit. Finora, i messaggi che noi ed Eve abbiamo inviato sono stati bit classici, ma qui il messaggio è *quantistico*. Possiamo misurare il messaggio quantistico per ottenere un bit classico, ma possiamo anche usare il messaggio quantistico che otteniamo da Eve come qualsiasi altro dato quantistico. Per esempio, possiamo applicargli tutte le rotazioni e le istruzioni che vogliamo.

A che cosa serve?
L'invio di dati quantistici potrebbe non sembrare molto più utile dell'invio di dati classici; dopotutto, l'invio di dati classici ci ha permesso di ottenere cose molte interessanti, finora. Al contrario, le applicazioni per l'invio di dati quantistici tendono a essere un po' più di nicchia, al momento.
Detto questo, il trasferimento dei dati quantistici è un esempio davvero utile per aiutarci a capire come funzionano i computer quantistici. Le idee sviluppate in questo capitolo spesso non sono direttamente utili, ma ci aiuteranno a costruire grandi cose in futuro.

Supponiamo di preparare un *messaggio quantistico* utilizzando l'operazione `msg.ry(0.123)`. Il seguente listato mostra come possiamo teletrasportare questo messaggio a Eve.

Listato 6.14 teleport.py: utilizzo del teletrasporto per trasferire dati quantistici.

```
if __name__ == "__main__":
    sim = Simulator(capacity=3)
    with sim.using_register(3) as (msg, here, there): (1)
        msg.ry(0.123) (2)
        teleport(msg, here, there) (3)
        there.ry(-0.123) (4)

        sim.dump()
```

(1) Come prima, alloca un registro di qubit e assegna un nome a ogni qubit.

(2) Prepara un messaggio da inviare a Eve. Qui, abbiamo mostrato l'utilizzo di un determinato angolo come messaggio, ma può trattarsi di qualsiasi cosa.

(3) Richiama il programma di teletrasporto che abbiamo scritto in precedenza per trasferire il messaggio che abbiamo preparato sul qubit di Eve

(4) Controlla l'output dell'istruzione `dump` per vedere che il registro che abbiamo allocato sia di nuovo nello stato $|000\rangle$.

Se Eve poi annulla la nostra rotazione ruotando dall'angolo opposto, possiamo controllare che il registro che abbiamo allocato sia tornato nello stato $|000\rangle$. Questo dimostra che il nostro teletrasporto ha funzionato. Eseguendo questo programma, otterrete un output simile al seguente:

```
Quantum object: dims = [[2, 2, 2], [1, 1, 1]], shape = (8, 1), type =
ket
Qobj data =
[[1.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]
 [0.]]
```

NOTA

L'output potrebbe differire di una fase globale, a seconda dei risultati di misurazione ottenuti.

Per verificare che il teletrasporto abbia funzionato, se Eve annulla l'istruzione che abbiamo applicato al suo qubit (`there.ry(0.123)`), dovrebbe recuperare lo stato $|0\rangle$ con cui abbiamo iniziato. Con il teletrasporto, noi ed Eve possiamo scambiarci informazioni quantistiche usando l'*entanglement* e la comunicazione classica, come potete dimostrare anche voi stessi nell'Esercizio 6.6.

ESERCIZIO 6.6 E SE NON FUNZIONASSE?

Provate a cambiare l'operazione vostra o di Eve per convincervi che alla fine otterrete uno stato $|000\rangle$ solo se annullate la stessa operazione che Eve ha applicato al suo qubit.

Ora possiamo vantarci con tutti i nostri amici (facendo qualsiasi riferimento fantascientifico) che possiamo teletrasportarci. Probabilmente avete capito perché questo "teletrasporto" *non* può portarci su un altro pianeta e perché quando teletrasportiamo un messaggio, non superiamo mai la velocità della luce.

Riepilogo

- Il teorema di non-clonazione ci impedisce di copiare dati arbitrari memorizzati dai registri quantistici, ma possiamo comunque trasferire i dati utilizzando l'operazione di `swap` e algoritmi come il teletrasporto.
- Quando un qubit non è *entangled* con nessun altro qubit, possiamo rappresentare il suo stato come un punto su una sfera, nota come *sfera di Bloch*.
- In generale, possiamo ruotare lo stato di un singolo qubit attorno all'asse *X*, *Y* o *Z*. Le operazioni che applicano rotazioni di 180°

attorno a ciascuno di questi assi sono chiamate *operazioni di Pauli* e descrivono l'inversione dei bit, delle fasi o di entrambi.
- Nel teletrasporto quantistico, l'*entanglement* viene utilizzato insieme alle rotazioni di Pauli per trasferire i dati quantistici da un qubit a un altro qubit, senza copiarli.

Parte I: conclusioni

Siamo arrivati alla fine della prima parte del libro, ma purtroppo i nostri qubit sono in un altro castello 😆. Superare questa parte non è stata un'impresa da poco, poiché mentre apprendevamo tutta una serie di nuovi concetti quantistici stavamo anche costruendo un simulatore per un dispositivo quantistico. Probabilmente sarete un po' dubbiosi su certi argomenti e pieni di domande, e va bene così.

SUGGERIMENTO

L'Appendice B contiene alcuni riferimenti rapidi (glossario e definizioni) che possono essere utili nella lettura del libro e nelle vostre successive attività di sviluppo quantistico.

Useremo e metteremo in pratica queste abilità per sviluppare programmi quantistici più complicati per applicazioni interessanti come la chimica e la crittografia. Prima di passare a questi problemi concreti, però, datevi una pacca sulla spalla: avete già fatto molto! Riassumiamo alcuni dei risultati che avete realizzato.

- Avete fatto un bel ripasso di algebra lineare e numeri complessi.
- Avete imparato che cos'è un qubit e che cosa potete farci.
- Avete costruito un simulatore multi-qubit in Python.
- Avete scritto dei programmi quantistici per attività come la distribuzione quantistica delle chiavi (QKD), l'esecuzione di giochi non locali e persino il teletrasporto quantistico.

- Avete imparato la notazione braket per gli stati dei sistemi quantistici.

Il vostro simulatore Python continuerà a essere uno strumento utile per cercare di capire che cosa sta succedendo quando arriveremo ad applicazioni più corpose. Nella Parte II, passeremo principalmente all'uso di Q# come strumento privilegiato per la scrittura di programmi quantistici. Nel prossimo capitolo parleremo del motivo per cui scriveremo programmi quantistici più avanzati in Q# piuttosto che in Python, ma i motivi principali sono la velocità e l'estendibilità. Inoltre, potrete usare Q# da Python o con un kernel Q# per Jupyter, e in tal modo potrete lavorare con il vostro ambiente di sviluppo preferito.

Ci rivediamo nella Parte II!

Programmazione di algoritmi quantistici in Q#

Se sei uno sviluppatore, è utile avere a disposizione lo strumento giusto per il tuo lavoro e il quantum computing non fa eccezione. Gli stack di software del quantum computing tendono ad essere piuttosto peculiari. Nella Parte II, espandiamo il nostro stack per includere Q#, un linguaggio di programmazione *specificamente progettato* per lavorare con dispositivi quantistici. Esattamente come altri linguaggi specifici del dominio possono aiutarci a sfruttare un dispositivo hardware classico specializzato, come una GPU o un FPGA, Q# ci aiuta a ottenere il massimo dal quantum computing, fornendoci gli strumenti giusti per implementare e applicare algoritmi quantistici. Come altri linguaggi specializzati, anche Q# funziona al meglio quando interagisce con linguaggi e piattaforme più generici, come Python e .NET.

Data questa premessa, iniziamo ad affinare le nostre capacità quantistiche e ad apprendere le basi della scrittura di algoritmi quantistici. Nel Capitolo 7, rielaboriamo i generatori quantistici di numeri casuali, con lo scopo di apprendere le basi del linguaggio Q#, basandoci sulle funzionalità che abbiamo sviluppato nel Capitolo 2. Successivamente, nel Capitolo 8, espandiamo la nostra cassetta degli attrezzi di tecniche di programmazione quantistica parlando di *phase kickback*, *oracoli* e altri nuovi trucchi, e li usiamo per provare nuovi

giochi. Infine, nel Capitolo 9 apprendiamo la tecnica della stima della fase, per sviluppare le competenze di cui avremo bisogno quando inizieremo a lavorare su problemi più pratici, nella Parte III.

Cambiare le probabilità: un'introduzione a Q#

Finora, abbiamo utilizzato Python per implementare il nostro stack software per simulare programmi quantistici. La Figura 2.1 (che riproponiamo nella Figura 7.1) rappresenta un buon modello del modo in cui i programmi che stiamo scrivendo interagiscono con il simulatore quantistico e i dispositivi che usiamo e costruiamo come sviluppatori quantistici.

Proseguendo, scriveremo programmi quantistici più complessi che potranno sfruttare funzionalità più specializzate del linguaggio, che sono difficili da implementare impiegando il nostro stack software in Python. Soprattutto mentre esploriamo gli algoritmi quantistici, è utile avere a disposizione un linguaggio dedicato alla programmazione quantistica. In questo capitolo inizieremo con Q#, il linguaggio Microsoft dedicato specificamente al dominio della programmazione quantistica, incluso nel Quantum Development Kit.

Presentazione del Quantum Development Kit

Il Quantum Development Kit fornisce un nuovo linguaggio, Q#, dedicato alla scrittura di programmi quantistici e alla loro simulazione su risorse classiche. I programmi quantistici scritti in Q# vengono

eseguiti considerando i dispositivi quantistici come se fossero "acceleratori", un po' come se dovessimo eseguire del codice su una scheda grafica.

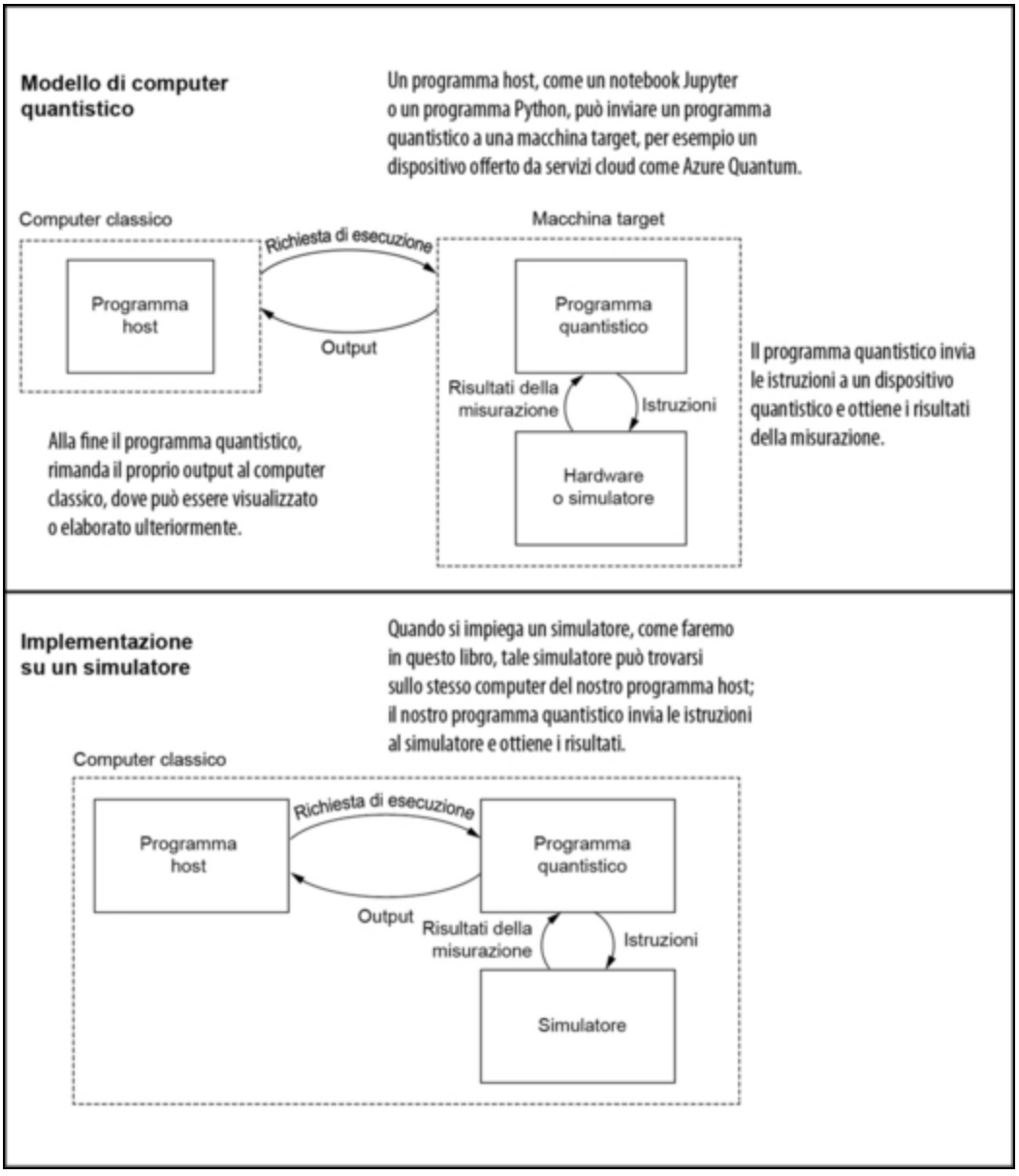


Figura 7.1 Un modello del modo in cui possiamo usare un computer quantistico. La metà superiore della figura rappresenta il modello generale di un computer quantistico. In questo libro impiegheremo simulatori locali e, appunto, la metà inferiore rappresenta ciò che costruiremo e utilizzeremo.

SUGGERIMENTO

Se avete mai utilizzato un framework di programmazione per schede grafiche come CUDA o OpenCL, questo è un modello molto simile.

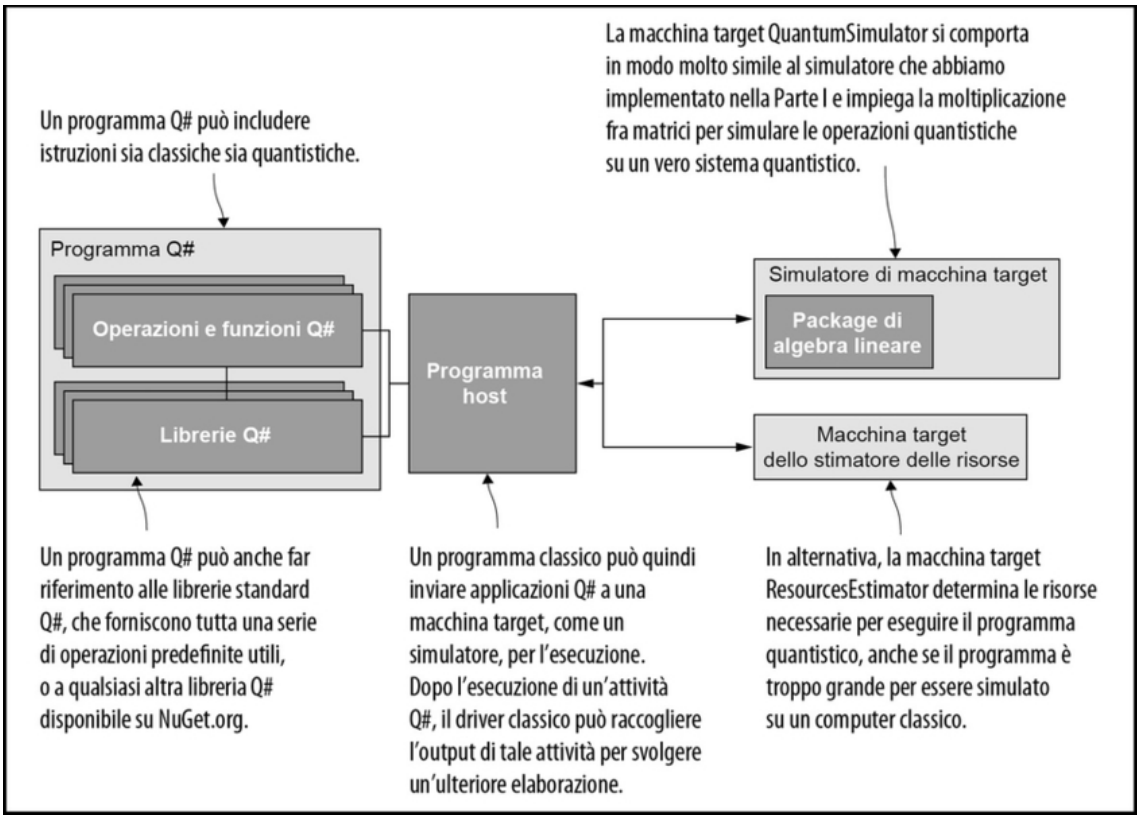


Figura 7.2 Stack software del Quantum Development Kit Microsoft su un computer classico. Possiamo scrivere un programma Q# composto da funzioni e operazioni, facendo riferimento a qualsiasi libreria Q# che vogliamo includere. Un programma host può quindi coordinare le comunicazioni tra il nostro programma Q# e la macchina target (per esempio, un simulatore, in esecuzione localmente sul nostro computer).

Cerchiamo di descrivere meglio lo stack software Q# illustrato nella Figura 7.2.

Un programma Q# è costituito da operazioni e funzioni che istruiscono l'hardware quantistico e classico a svolgere determinate attività. Esistono anche molte librerie Q# che offrono operazioni e funzioni utili e pronte all'uso da usare nei programmi.

Una volta che il programma Q# è stato scritto, abbiamo bisogno di un modo per passare le istruzioni all'hardware. Un programma classico, talvolta chiamato *driver* o *programma host*, è responsabile dell'allocazione di una macchina target e dell'esecuzione di un'operazione Q# su tale macchina.

Il Quantum Development Kit include un plug-in per Jupyter Notebook, chiamato IQ#, che semplifica l'avvio di Q# fornendo automaticamente i programmi host. Nel Capitolo 9 vedremo come scrivere programmi host utilizzando Python, ma per ora ci concentreremo su Q#. L'Appendice A presenta le istruzioni di configurazione dell'ambiente Q# per lavorare con i Jupyter Notebook.

Utilizzando il plug-in IQ# per Jupyter Notebook, possiamo usare una delle due diverse macchine target per eseguire il codice Q#. La prima è la macchina target `QuantumSimulator`, che è molto simile al simulatore Python che abbiamo sviluppato. Sarà però molto più veloce del nostro codice Python per la simulazione dei qubit.

La seconda è la macchina target `ResourcesEstimator`, che ci consentirà di stimare quanti qubit e istruzioni quantistiche saranno necessari per eseguirlo senza simularlo completamente. Ciò è particolarmente utile per farsi un'idea delle risorse necessarie per eseguire un programma Q# per la nostra applicazione, quando inizieremo a vedere programmi Q# più impegnativi, più avanti nel libro.

Per farsi un'idea del modo in cui funziona il tutto, cerchiamo di iniziare scrivendo un'applicazione "hello, world" puramente classica. Innanzitutto, avviamo Jupyter Notebook lanciando il seguente comando in un terminale:

```
jupyter notebook
```

Si aprirà automaticamente una nuova scheda nel nostro browser con la home page della nostra sessione di Jupyter Notebook. Dal menu *New*↓, selezionate Q# per creare un nuovo notebook Q#. Digitate

quanto segue nella prima cella vuota del notebook e premete Ctrl-Invio per eseguirlo:

```
function HelloWorld() : Unit { (1)
    Message("Hello, classical world!"); (2)
}
```

(1) Definisce una nuova funzione che non accetta argomenti e restituisce la tupla vuota, il cui tipo è scritto come `Unit`.

(2) Indica alla macchina target di raccogliere un messaggio diagnostico. La macchina target `QuantumSimulator` mostra la diagnostica sullo schermo: in pratica possiamo usare `Message` come `print` in Python.

Dovreste ottenere qualcosa di simile a quanto rappresentato nella Figura 7.3.

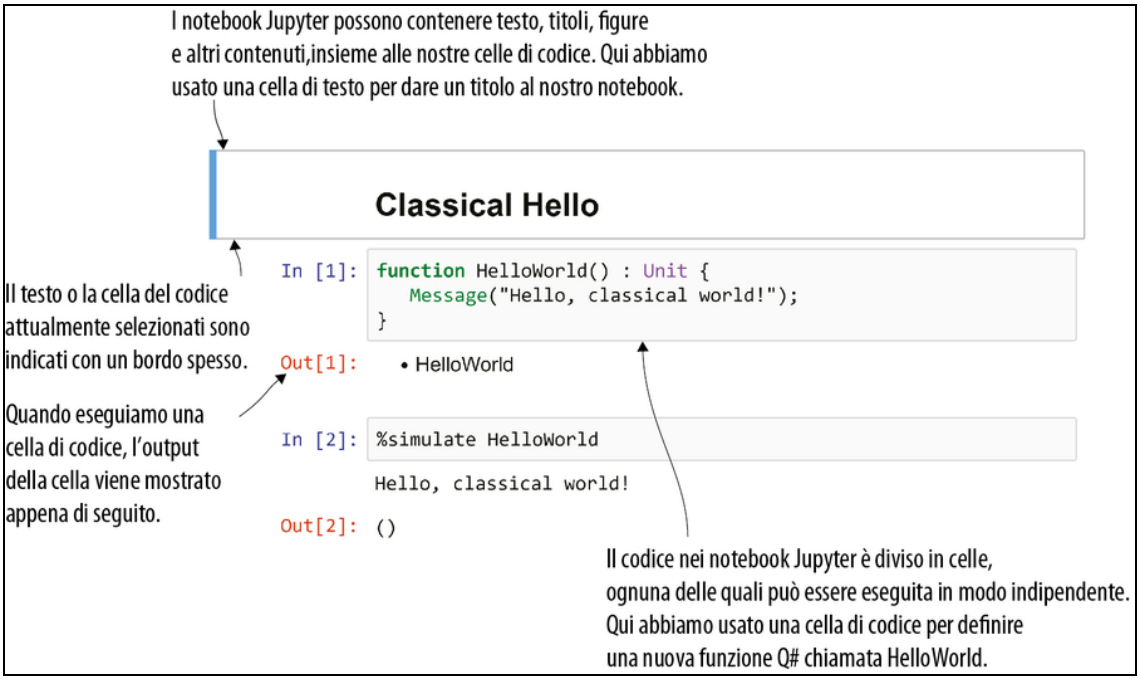


Figura 7.3 Introduzione a IQ# e a Jupyter Notebook. Qui definiamo una nuova funzione Q# chiamata HelloWorld come prima cella di un notebook Jupyter; nella seconda cella compaiono i risultati della simulazione di quella funzione.

SUGGERIMENTO

A differenza di Python, Q# usa il punto e virgola anziché il codice di fine riga per concludere le istruzioni. Se ottenete molti errori del compilatore,

assicuratevi di non aver dimenticato qualche punto e virgola.

Dovreste ottenere come risposta il fatto che la funzione `HelloWorld` è stata compilata con successo. Per eseguire la nuova funzione, possiamo usare il comando `%simulate` in una nuova cella.

Listato 7.1 Utilizzo del comando %simulate in Jupyter.

```
In [2]: %simulate HelloWorld

Hello, classical world!
```

Un po' di magia classica

Il comando `%simulate` è un esempio di comando "magico", in quanto, in realtà, non fa parte di Q#, ma è un'istruzione per l'ambiente Jupyter Notebook. Se avete familiarità con il plug-in IPython per Jupyter, potreste aver usato anche altri comandi magici simili, per dire a Jupyter come gestire le funzionalità di disegno di Python.

In questo libro, i comandi magici che usiamo iniziano tutti con `%` per distinguerli facilmente dal codice Q#.

In questo esempio, `%simulate` alloca una macchina target e le invia una funzione o un'operazione Q#. Nel Capitolo 9, vedremo come eseguire qualcosa di simile utilizzando programmi host Python invece di utilizzare Jupyter Notebook.

Il programma Q# viene inviato al simulatore, ma in questo caso il simulatore esegue solo la logica classica, in quanto il programma non contiene ancora istruzioni quantistiche.

ESERCIZIO 7.1 CAMBIARE I SALUTI

Modificate la definizione di `HelloWorld` per includere il vostro nome invece di "classical world", quindi eseguite di nuovo `%simulate` usando la vostra nuova definizione.

SOLUZIONI DEGLI ESERCIZI

Tutte le soluzioni degli esercizi di questo libro si trovano nel suo archivio di codice: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Aprite la cartella del capitolo e aprite il notebook Jupyter con il nome che menziona le soluzioni degli esercizi.

Funzioni e operazioni in Q#

Ora che abbiamo il Quantum Development Kit installato e funzionante con Jupyter Notebook, cerchiamo di utilizzare Q# per scrivere alcuni programmi quantistici. Tornando al Capitolo 2, abbiamo visto che una cosa utile da fare con un qubit consiste nel generare numeri casuali un bit classico alla volta. Riprendere in mano quell'applicazione può essere un ottimo punto di partenza con Q#, soprattutto perché i numeri casuali sono sempre utili, per ogni tipo di gioco.

Giocare con i generatori quantistici di numeri casuali in Q#

Molto tempo fa, a Camelot, Morgana condivideva la nostra passione per i giochi. Essendo un'esperta matematica, dotata di abilità ben oltre i suoi tempi, Morgana era anche nota per usare i qubit, ogni tanto, nei suoi giochi. Un giorno, mentre Sir Lancillotto giaceva addormentato sotto un albero, Morgana lo intrappolò e lo sfidò a un piccolo gioco: ognuno di loro doveva cercare di indovinare il risultato della misurazione di uno dei qubit di Morgana.

Due facce dello stesso qubit?

Nel Capitolo 2 abbiamo visto come si generano numeri casuali un bit alla volta preparando e misurando i qubit. Questo significa che un qubit può essere utilizzato per implementare una moneta. In questo capitolo useremo lo stesso tipo di idea e costruiremo una moneta come una sorta di interfaccia che consente al suo utente di "lanciarla" e ottenere un bit casuale. Cioè, possiamo implementare l'interfaccia della moneta preparando e misurando qubit.

Se il risultato della misurazione lungo l'asse Z è 0, Lancillotto vince la partita e torna da Ginevra. Ma se il risultato è 1, Morgana vince e Lancillotto deve restare con lei e giocare di nuovo. Notate la

somiglianza con il nostro precedente programma QRNG. Proprio come nel Capitolo 2, misureremo un qubit per generare numeri casuali, questa volta per un gioco. Certo, Morgana e Lancillotto potrebbero lanciare una moneta più tradizionale, ma allora… dove sarebbe il divertimento?

Ecco i passaggi del gioco di Morgana.

1. Preparare un qubit nello stato $|0\rangle$.
2. Applicare l'operazione di Hadamard (ricordate che l'operatore unitario H porta $|0\rangle$ in $|+\rangle$ e viceversa).
3. Misurare il qubit sull'asse Z. Se il risultato della misurazione è 0, Lancillotto può tornare a casa. Altrimenti, deve restare a giocare di nuovo.

Seduti in un bar a guardare la gente che passa, possiamo usare il nostro portatile per prevedere che cosa accadrà nel gioco di Morgana con Lancillotto scrivendo un programma quantistico in Q#. A differenza della funzione `HelloWorld` che abbiamo scritto in precedenza, il nostro nuovo programma avrà bisogno di lavorare con i qubit, quindi cerchiamo di dedicare un momento a vedere come implementarlo con il Quantum Development Kit.

Il modo principale in cui interagiamo con i qubit in Q# è tramite chiamate a *operazioni* che rappresentano istruzioni quantistiche. Per esempio, l'operazione `H` in Q# rappresenta l'istruzione di Hadamard che abbiamo visto nel Capitolo 2. Per capire come funzionano queste operazioni, è utile conoscere la differenza tra le operazioni Q# e le funzioni che abbiamo visto nell'esempio `HelloWorld`.

- Le *funzioni* Q# rappresentano la logica classica, prevedibile, come le funzioni matematiche (`Sin`, `Log`). Dato un certo input, le funzioni restituiscono sempre lo stesso output.

- Le *operazioni* Q# rappresentano il codice che può avere *effetti collaterali*, come campionamento di numeri casuali o l'emissione di istruzioni quantistiche che modificano lo stato di uno o più qubit.

Questa distinzione aiuta il compilatore a capire come elaborare automaticamente il nostro codice in programmi quantistici più grandi; ne riparleremo più avanti.

Un altro punto di vista su funzioni e operazioni

Un altro modo per pensare alla differenza tra funzioni e operazioni è che le funzioni calcolano qualcosa, ma *non possono far accadere nulla*. Non importa quante volte richiamiamo la funzione per la radice quadrata `Sqrt`, ma non cambierà nulla nel nostro programma Q#. Al contrario, se eseguiamo l'operazione `X`, viene inviata un'istruzione `X` al nostro dispositivo quantistico, cosa che provoca un cambiamento nello stato del dispositivo. A seconda dello stato iniziale del qubit cui è stata applicata l'istruzione `X`, possiamo quindi dire che l'istruzione `X` è stata applicata misurando il qubit. Poiché le funzioni non fanno nulla di tutto ciò, dato un input possiamo sempre prevedere esattamente il loro output.

Una conseguenza importante è che le funzioni non possono richiamare operazioni, ma le operazioni possono richiamare funzioni. Questo perché possiamo fare in modo che un'operazione, che non è necessariamente prevedibile, richiami una funzione prevedibile e abbiamo ancora qualcosa che non è prevedibile. Al contrario, una funzione prevedibile non può richiamare un'operazione potenzialmente imprevedibile e rimanere comunque prevedibile.

Parleremo della differenza tra le funzioni e le operazioni Q# man mano che le useremo nel libro.

Poiché vogliamo che le istruzioni quantistiche influenzino i nostri dispositivi quantistici (e anche il destino di Lancillotto), tutte le operazioni quantistiche in Q# sono definite come… operazioni. Per esempio, supponiamo che Morgana e Lancillotto preparino il loro qubit nello stato $|+\rangle$ usando l'istruzione di Hadamard. Possiamo prevedere il

risultato del loro gioco scrivendo l'esempio del generatore quantistico di numeri casuali (QRNG) del Capitolo 2 come un'operazione Q#.

Questa operazione potrebbe avere effetti collaterali…
Quando vogliamo inviare istruzioni alla nostra macchina target per fare qualcosa con i nostri qubit, dobbiamo farlo tramite un'operazione, poiché l'invio di un'istruzione è una sorta di *effetto collaterale*. Cioè, quando eseguiamo un'operazione, non stiamo solo calcolando qualcosa: stiamo *facendo* qualcosa. Eseguendo un'operazione due volte non sarà la stessa cosa di eseguirla una sola volta, anche se entrambe le volte otteniamo lo stesso output. Gli effetti collaterali non sono deterministici o prevedibili, quindi non possiamo usare funzioni per inviare istruzioni per manipolare i nostri qubit.

Nel Listato 7.2, facciamo proprio questo, iniziando con la scrittura dell'operazione `GetNextRandomBit` per simulare ogni round del gioco di Morgana. Notate che poiché `GetNextRandomBit` deve funzionare con i qubit, deve essere un'operazione e non una funzione. Possiamo chiedere alla macchina target uno o più nuovi qubit con l'istruzione `use`.

Allocazione di qubit in Q#
L'istruzione `use` è uno dei due modi per chiedere dei qubit alla macchina target. Non c'è limite al numero di istruzioni `use` che possiamo avere nei programmi Q#, a parte il numero di qubit che ogni macchina target può allocare.
Alla fine di un blocco (cioè un `operation`, `for` o `if`) contenente un'istruzione `use`, i qubit tornano alla macchina target. Spesso, i qubit vengono allocati in questo modo all'inizio di un'operazione, quindi un modo per immaginare le istruzioni `use` è assicurarsi che ogni qubit allocato sia "di proprietà" di una determinata operazione. Questo rende impossibile la "perdita" di qubit all'interno di un programma Q#, il che è molto utile, dal momento che è probabile che i qubit siano risorse molto costose su vero hardware quantistico.
Se abbiamo bisogno di avere un maggiore controllo sul momento in cui i qubit vengono deallocati, le istruzioni `use` possono anche essere opzionalmente seguite da un blocco, indicato con `{` e `}`. In tal caso, i qubit vengono deallocati alla fine del blocco anziché alla fine dell'operazione.

Q# offre anche un altro modo per allocare qubit, noto come *prestito* (`borrow`). A differenza di quando si assegnano i qubit tramite istruzioni `use`, l'istruzione `borrow` permette di prendere in prestito i qubit di proprietà di altre operazioni senza sapere in quale stato cominciano. Non parleremo molto di `borrow` in questo libro, ma l'istruzione funziona in modo molto simile all'istruzione `use`, in quanto rende impossibile dimenticare che abbiamo preso in prestito un qubit.

Per convenzione, tutti i qubit iniziano nello stato $|0\rangle$ appena dopo averli ottenuti, e promettiamo alla macchina target che li riporteremo nello stato $|0\rangle$ alla fine del blocco in modo che la macchina target possa concederli immediatamente alla prossima operazione che ne avrà bisogno.

Listato 7.2 Operation.qs: Q# per simulare un round del gioco di Morgana.

```
operation GetNextRandomBit() : Result { (1)
    use qubit = Qubit(); (2)
    H(qubit); (3)
    return M(qubit); (4)

}
```

(1) Dichiara un'operazione, poiché vogliamo utilizzare dei qubit e restituire la misurazione `Result` al chiamante.

(2) La parola riservata `use` in Q# chiede alla macchina target uno o più qubit. Qui, chiediamo un singolo valore di tipo `Qubit`, che memorizziamo nella nuova variabile `qubit`.

(3) Dopo aver richiamato `H`, `qubit` è nello stato $H|0\rangle = |+\rangle$.

(4) Utilizza l'operazione `M` per misurare il nostro qubit nella base Z. Il risultato sarà `Zero` o `One` con uguale probabilità. Dopo la misurazione, possiamo restituire al chiamante i dati classici.

Qui, usiamo l'operazione `M` per misurare il nostro qubit in base Z, salvando il risultato nella variabile `Result` che abbiamo dichiarato in precedenza. Poiché lo stato del qubit è in una sovrapposizione uguale di $|0\rangle$ e $|1\rangle$, il risultato sarà `Zero` o `One` con uguale probabilità.

Applicazioni Q# autonome

Possiamo anche scrivere applicazioni Q# come QRNG come applicazioni autonome, invece di richiamarle da notebook Python o IQ#. Per farlo, possiamo designare una delle operazioni Q# nell'applicazione quantistica come *punto di ingresso*, ed essa verrà quindi richiamata automaticamente quando eseguiremo la nostra applicazione dalla riga di comando.

Per questo libro, continueremo a utilizzare Jupyter e Python per lavorare con Q#. Ma osservate l'esempio su `https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp/tree/master/ch07/Qrng` per informazioni sulla scrittura di applicazioni Q# autonome.

Nel Listato 7.2, vediamo anche il nostro primo esempio di apertura di uno "spazio dei nomi", un *namespace* in Q#. Come i namespace in C++ e C# o i package in Java, i namespace Q# aiutano a mantenere separate le funzioni e le operazioni. Per esempio, l'operazione `H` richiamata nel Listato 7.2 è definita nella libreria standard Q# come `Microsoft.Quantum.Intrinsic.H`; ovvero, risiede nel namespace `Microsoft.Quantum.Intrinsic`. Per usare `H`, possiamo specificarne il nome completo, `Microsoft.Quantum.Intrinsic.H`, o usare un'istruzione `open` per rendere disponibili tutte le operazioni e le funzioni presenti in un namespace:

```
open Microsoft.Quantum.Intrinsic;
```

Rende tutte le funzioni e le operazioni fornite in `Microsoft.Quantum.Intrinsic` disponibili per l'uso all'interno di un notebook Q# o di un file sorgente senza dover specificare il loro nome completo.

SUGGERIMENTO

Quando si scrive codice Q# in Jupyter Notebook, vengono sempre aperti automaticamente i namespace `Microsoft.Quantum.Intrinsic` e `Microsoft.Quantum.Canon` della libreria standard Q#, che vengono utilizzati dalla maggior parte del codice Q#. Per esempio, quando abbiamo richiamato `Message`, in precedenza, quella funzione è stata fornita nel namespace

`Microsoft.Quantum.Intrinsic` che il kernel Q# ha aperto automaticamente. Per controllare quali namespace sono aperti in un notebook Q#, possiamo usare il comando magico `%lsopen`.

ESERCIZIO 7.2 GENERARE PIÙ BIT

Usa il comando magico `%simulate` per eseguire l'operazione `GetNextRandomBit` alcune volte. Ottenete i risultati che vi aspettereste?

Successivamente, vedremo quanti round occorreranno a Lancillotto per ottenere lo `Zero` che ha bisogno per tornare a casa. Scriviamo un'operazione per svolgere i round fino a quando non si ottiene uno `Zero`. Poiché questa operazione simula il gioco di Morgana, lo chiameremo `PlayMorganasGame`.

SUGGERIMENTO

In Q# tutte le variabili sono immutabili per default.

Listato 7.3 Operations.qs: simulazione di più round del gioco di Morgana usando Q#.

```
operation PlayMorganasGame() : Unit {
    mutable nRounds = 0; (1)
    mutable done = false;
    repeat { (2)
        set nRounds = nRounds + 1;
        set done = (GetNextRandomBit() == Zero); (3)
    }
    until done; (4)

    Message($"It took Lancelot {nRounds} turns to get home."); (5)
```

(1) Utilizza la parola riservata `mutable` per dichiarare una variabile che indica quanti round sono passati. Possiamo modificare il valore di questa variabile in un secondo momento con la parola riservata `set`.

(2) Q# consente alle operazioni di utilizzare un ciclo di ripetizione fino al successo.

(3) All'interno del nostro ciclo, richiamiamo il QRNG che abbiamo scritto in precedenza come l'operazione `GetNextRandomBit`. Controlliamo se il risultato è `Zero` e, in tal caso, impostiamo `done` su `true`.

(4) Se abbiamo `Zero`, possiamo fermare il ciclo.

❺ Usiamo di nuovo `Message` per mostrare il numero di round sullo schermo. Per fare ciò, usiamo stringhe `$""`, che ci consentono di includere variabili nel messaggio diagnostico impiegando segnaposto `{}` all'interno della stringa.

> **SUGGERIMENTO**
>
> Le stringhe Q# indicate con `$""` sono chiamate *stringhe interpolate* e funzionano in modo molto simile alle stringhe `f""` in Python.

Il Listato 7.3 include un flusso di controllo Q# chiamato ciclo *repeat-until-success* (*RUS*). A differenza di un ciclo `while`, i cicli RUS ci consentono anche di specificare una "correzione" che viene eseguita se la condizione per uscire dal ciclo non viene soddisfatta.

Quando dobbiamo eseguire il reset dei qubit?

In Q#, quando allochiamo un nuovo qubit con `use`, promettiamo alla macchina target che lo rimetteremo nello stato $|0\rangle$ prima di deallocarlo. A prima vista, questo non sembra necessario, poiché la macchina target potrebbe ripristinare con facilità lo stato dei qubit quando vengono deallocati: dopotutto, spesso richiamiamo l'operazione `Reset` alla fine di un'operazione o di un blocco `use`. In effetti, questo accade automaticamente quando viene misurato un qubit, subito prima che venga rilasciato.

È importante notare che l'operazione `Reset` funziona effettuando una misurazione nella base Z e invertendo il qubit con un'operazione `X` se la misurazione restituisce `One`. In molti dispositivi quantistici, la misurazione è molto più costosa di altre operazioni, quindi se possiamo evitare di richiamare `Reset`, possiamo ridurre il costo dei nostri programmi quantistici. Soprattutto visti i limiti dei dispositivi a medio termine, questo tipo di ottimizzazione può essere fondamentale per rendere davvero utile in pratica un programma quantistico.

Più avanti nel capitolo, vedremo esempi in cui conosciamo lo stato di un qubit quando deve essere deallocato, in modo tale da poter "de-preparare" il qubit invece di misurarlo. Non abbiamo una misurazione definitiva in questi casi, e in tal modo il compilatore # Q non aggiunge un reset automatico, cosa che ci evita la necessità di un'operazione di misurazione potenzialmente costosa.

Possiamo eseguire questa nuova operazione con il comando `%simulate` in un modo molto simile all'esempio `HelloWorld`. In tal modo, possiamo vedere quanto tempo deve rimanere da Morgana Lancillotto:

```
In []: %simulate PlayMorganasGame
It took Lancelot 1 turns to get home.
Out[]: ()
```

Sembra che Lancillotto sia stato fortunato questa volta! O forse sfortunato, se si annoiava a girare attorno alla tavola rotonda a Camelot.

Approfondimento: open e import

A prima vista, l'istruzione `open` di Q# potrebbe somigliare molto all'istruzione `import` di Python, JavaScript o TypeScript. Sia `open` sia `import` rendono disponibile nei nostri programmi e nelle nostre applicazioni il codice dalle librerie. La differenza principale è che l'apertura di un namespace con l'istruzione `open` fa solo questo: apre un namespace e lo rende disponibile, ma non esegue alcun codice e non cambia il modo in cui il nostro codice viene compilato e trova le librerie.

In linea di principio, potremmo scrivere ogni programma Q# senza nemmeno un'istruzione `open`, facendo riferimento esplicito a ciascuna funzione e operazione con il suo nome completo, incluso il suo namespace (cioè richiamando `Microsoft.Quantum.Intrinsic.H` invece della sola `H`). In questo senso, `open` è simile all'istruzione `using` di C++ o C#, all'istruzione `open` di F# e all'istruzione `import` di Java.

Al contrario, `import` in Python, JavaScript e TypeScript non solo rende disponibili al nostro codice i nomi di una libreria, ma lancia anche parti di tali librerie. Quando un modulo Python viene importato per la prima volta, l'interprete Python utilizza il suo nome per cercare dove è definito quel modulo e quindi esegue il modulo insieme a tutto il suo eventuale codice di inizializzazione. Spesso, per utilizzare questi moduli, è necessario prima installare uno o più package utilizzando uno strumento come pip o conda per i package Python o npm per JavaScript e TypeScript.

Lo stesso concetto può essere utilizzato per aggiungere nuove librerie ai nostri programmi Q# con il gestore di package NuGet. In un notebook Q#, il comando `%package` chiede al kernel di Q# di scaricare una data libreria Q# e di aggiungerla alla nostra sessione. Per esempio, quando apriamo un nuovo notebook, la libreria

standard Q# viene scaricata e installata automaticamente dal package `Microsoft.Quantum.Standard` da nuget.org (https://www.nuget.org/packages/Microsoft.Quantum.Standard). Allo stesso modo, quando si scrivono applicazioni Q# a riga di comando, l'esecuzione di `dotnet add package` aggiunge i metadati necessari al compilatore Q# del nostro progetto per trovare il package di cui abbiamo bisogno.
Per maggiori dettagli, consultate la documentazione del Quantum Development Kit in https://docs.microsoft.com/azure/quantum/user-guide/libraries/additional-libraries.

Passaggio di operazioni come argomenti

Supponiamo che, nel gioco di Morgana, fossimo interessati a campionare bit casuali con probabilità non uniforme. Dopotutto, Morgana non ha detto a Lancillotto *come* ha preparato il qubit che essi misurano; può farlo continuare a giocare più a lungo se impiega una moneta truccata.

Il modo più semplice per modificare il gioco di Morgana consiste, invece di richiamare direttamente `H`, nel prendere come input un'operazione che rappresenta ciò che Morgana fa per preparare il gioco. Per prendere un'operazione come input, dobbiamo scrivere il *tipo* di input, esattamente come possiamo scrivere `qubit: Qubit` per dichiarare un `qubit` di input di tipo `Qubit`. I tipi delle operazioni sono indicati da frecce spesse (`=>`) dal tipo di input al tipo di output. Per esempio, `H` ha il tipo `Qubit => Unit` poiché `H` accetta come input un singolo qubit e restituisce come output una tupla vuota.

SUGGERIMENTO

In Q#, le funzioni sono indicate da frecce sottili (`->`), mentre le operazioni sono indicate da frecce spesse (`=>`).

Listato 7.4 Usare le operazioni come input per prevedere il gioco di Morgana.

```
operation PrepareFairCoin(qubit : Qubit) : Unit {
    H(qubit);
}
operation GetNextRandomBit(
    statePreparation : (Qubit => Unit) (1)
) : Result {
    use qubit = Qubit();
    statePreparation(qubit); (2)
    return Microsoft.Quantum.Measurement.MResetZ(qubit); (3)
}
```

(1) Abbiamo aggiunto a `GetNextRandomBit` un nuovo input chiamato `statePreparation`, che rappresenta l'operazione che vogliamo utilizzare per preparare lo stato che usiamo come "moneta". In questo caso, `Qubit => Unit` è il tipo di qualsiasi operazione che richiede un singolo qubit e restituisce come tipo la tupla vuota `Unit`.

(2) In `GetNextRandomBit`, l'operazione passata come `statePreparation` può essere richiamata come qualsiasi altra operazione.

(3) Le librerie standard Q# forniscono `Microsoft.Quantum.Measurement.MResetZ` per misurare ed eseguire il reset di un qubit in un unico passaggio. In questo caso, l'operazione `MResetZ` fa la stessa cosa dell'istruzione `return M(qubit);` nell'esempio precedente. La differenza è che `MResetZ` esegue sempre il reset del suo qubit di input, non solo quando il qubit viene utilizzato prima di essere rilasciato. Vedremo meglio questa operazione più avanti nel capitolo, dove impiegheremo anche un nome più breve per richiamare questa operazione.

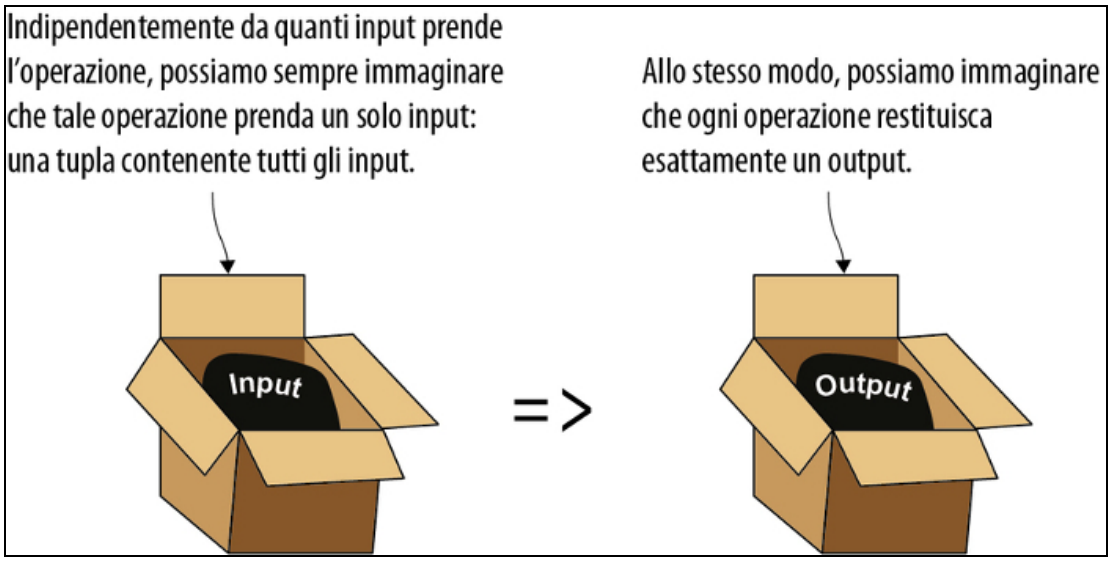
ESERCIZIO 7.3 IL TIPO DI GETNEXTRANDOMBIT

Qual è il tipo della nostra nuova definizione di `GetNextRandomBit`?

Tupla in, tupla out

Tutte le funzioni e le operazioni in Q# accettano in input una tupla e restituiscono in output una tupla. Per esempio, una funzione dichiarata come `function Pow(x:`

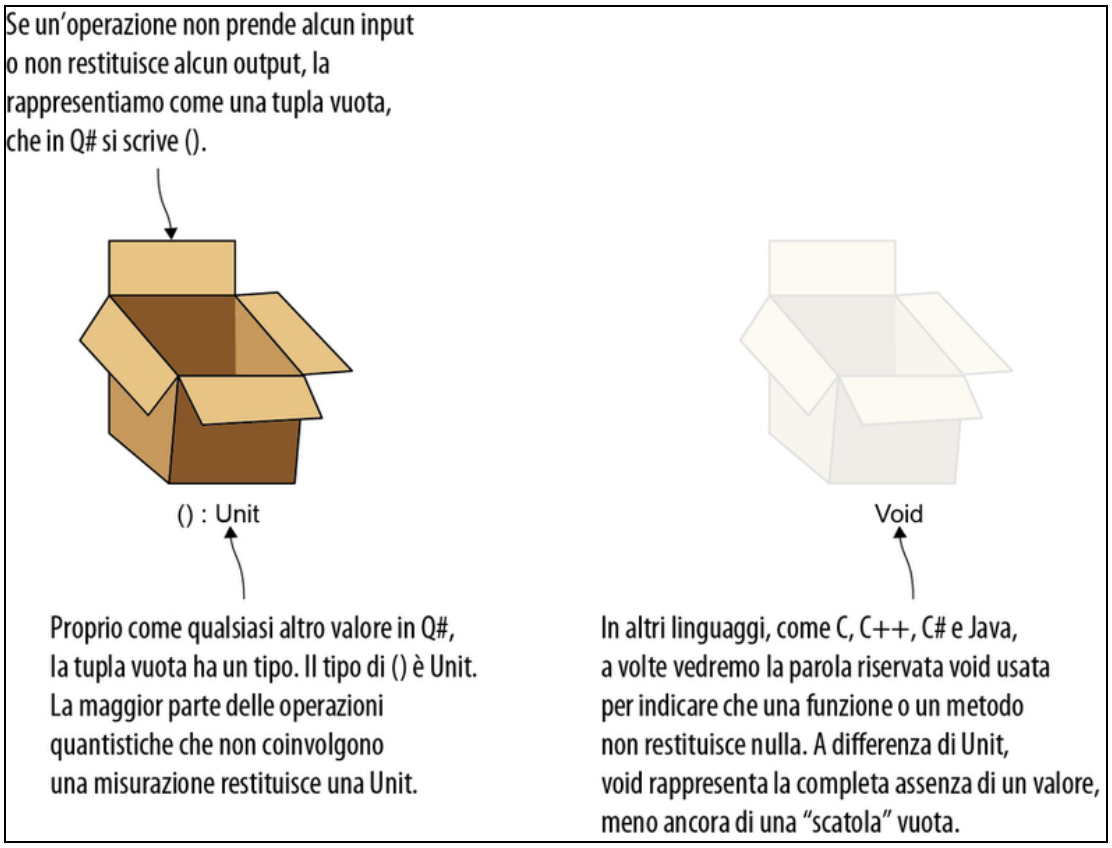
`Double, y: Double): Double {…}` prende come input una tupla `(Double, Double)` e restituisce in output una tupla `(Double)`. Funziona a causa di una proprietà nota come *equivalenza singleton-tuple*. Per qualsiasi tipo `'T`, la tupla `(' T)` contenente una singola `'T` è equivalente a `'T` stessa. Nell'esempio di `Pow`, possiamo pensare all'output come a una tupla `(Double)` che è equivalente a `Double`.



Rappresentazione di operazioni con un unico input e un unico output.

Detto questo, una funzione o un'operazione che non restituisce alcun output può essere considerata come se restituisse una tupla senza elementi, `()`. Il tipo di tali tuple è detto `Unit`, come in altri linguaggi basati su tuple, come F#. Se immaginiamo una tupla come una sorta di scatola, allora è qualcosa di diverso dal `void` usato in C, C++ o C#, perché è "qualcosa": è una scatola senza niente dentro.

In Q#, restituiamo sempre una "scatola", anche se è vuota. Non ha significato in Q# una funzione o un'operazione che non restituisce "niente". Per ulteriori dettagli, vedi il paragrafo 7.2 di *Get Programming with F#* di Isaac Abraham (Manning, 2018).



Unit e void.

Nel Listato 7.4, vediamo che `GetNextRandomBit` tratta il suo input `statePreparation` come una "scatola nera". L'unico modo per imparare qualcosa sulla strategia preparatoria di Morgana è *eseguirlo*.

In altre parole, non vogliamo fare nulla con `StatePreparation` che implichi che sappiamo che cosa fa o che cosa è. L'unico modo in cui `GetNextRandomBit` può interagire con `statePreparation` è richiamandolo, passandogli un `Qubit` su cui agire.

Questo ci permette di riutilizzare la logica contenuta in `GetNextRandomBit` per vari tipi di procedure di preparazione degli stati che Morgana potrebbe utilizzare per causare un po' di problemi a Lancillotto. Per esempio, supponiamo che lei desideri usare una

moneta truccata che restituisce uno `Zero` tre quarti delle volte e un `One` un quarto delle volte. Potremmo eseguire qualcosa di simile al seguente per prevedere questa nuova strategia.

Listato 7.5 Passaggio di diverse strategie di preparazione degli stati a PlayMorganasGame.

```
open Microsoft.Quantum.Math; (1)
operation PrepareQuarterCoin(qubit : Qubit) : Unit {
    Ry(2.0 * PI() / 3.0, qubit); (2)
}
```

(1) Le funzioni matematiche classiche come `Sin`, `Cos`, `Sqrt` e `ArcCos`, nonché le costanti come `PI()` sono fornite dal namespace `Microsoft.Quantum.Math`.

(2) L'operazione `Ry` implementa una rotazione sull'asse *Y*. Q# usa i radianti anziché i gradi per esprimere le rotazioni, quindi questa è una rotazione di 120° attorno all'asse *Y*.

L'operazione `Ry` implementa la rotazione attorno all'asse *Y* che abbiamo visto nel Capitolo 2. Pertanto, quando ruotiamo lo stato di `qubit` di 120° nel Listato 7.5, se `qubit` inizia nello stato $|0\rangle$, ciò prepara `qubit` nello stato $\text{Ry}(120°)\,|0\rangle = \sqrt{3/4}\,|0\rangle + \sqrt{1/4}\,|1\rangle$. Di conseguenza, la probabilità di osservare 1 quando misuriamo `qubit` è $\sqrt{1/4}^2 = 1/4$.

Possiamo rendere questo esempio ancora più generale, consentendo a Morgana di specificare un bias arbitrario per la sua moneta (che è implementata da un qubit condiviso).

Listato 7.6 Passaggio di operazioni per implementare bias arbitrari per la moneta.

```
operation PrepareBiasedCoin(
    morganaWinProbability : Double, qubit : Qubit
) : Unit {
    let rotationAngle = -2.0 * ArcCos(Sqrt(1.0 -
morganaWinProbability)); (1)
    Ry(rotationAngle, qubit);
}
operation PrepareMorganasCoin(qubit : Qubit) : Unit { (2)
    PrepareBiasedCoin(0.62, qubit);
```

```
}
```

(1) Trova di quale angolo ruotare il qubit di input per ottenere la giusta probabilità di vedere un `One` come risultato. Questo richiede un po' di trigonometria; vedi il riquadro successivo per tutti i dettagli.

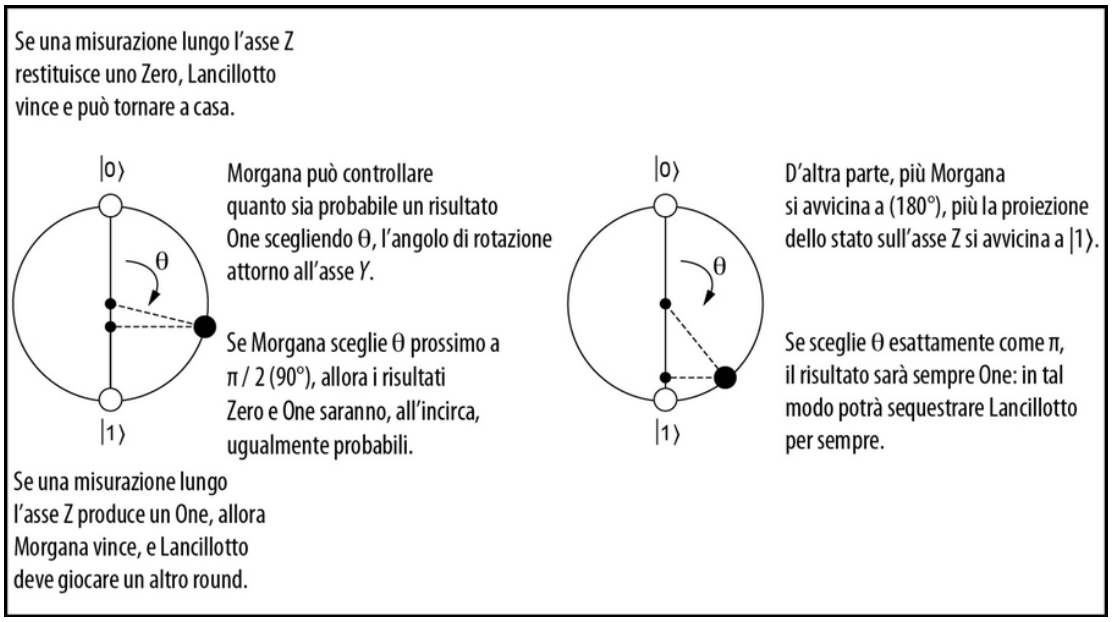
(2) Questa operazione ha il giusto tipo di *signature* (`Qubit => Unit`), e possiamo vedere che la probabilità che Morgana vinca ogni round è del 62%.

Un po' di trigonometria

Come abbiamo visto più volte, il quantum computing si occupa ampiamente di rotazioni. Per capire di quali angoli abbiamo bisogno per le nostre rotazioni, ci affidiamo alla trigonometria (letteralmente, "studio dei triangoli"), una branca della matematica che descrive proprio gli angoli di rotazione. Per esempio, come abbiamo visto nel Capitolo 2, ruotando $|0\rangle$ di un angolo q attorno all'asse *Y* si ottiene uno stato cos(-q / 2) $|0\rangle$ + sin(-q / 2) $|1\rangle$. Sappiamo di voler scegliere q tale che cos(-q / 2) = $\sqrt{100\% - 62\%}$ in modo da ottenere una probabilità del 62% di un risultato `One`. Ciò significa che abbiamo bisogno di "annullare" la funzione coseno di capire quanto deve valere q.

In trigonometria, l'inversa della funzione coseno è la funzione arcoseno, arccos. Prendendo l'arcocoseno di entrambi i lati di cos(-q / 2) = $\sqrt{100\% - 62\%}$ otteniamo arccos(cos(-q / 2)) = arccos($\sqrt{100\% - 62\%}$). Possiamo semplificare arccos e cos per trovare un angolo di rotazione che ci dia quello che ci serve, -q / 2 = arccos($\sqrt{100\% - 62\%}$).

Infine, moltiplichiamo entrambi i lati per -2 per ottenere l'equazione usata nella quarta riga del Listato 7.6. Possiamo vedere visivamente questo argomento nella figura seguente.



Come Morgana può scegliere q per controllare il gioco. Più l'angolo q da lei scelto si avvicina a π, più a lungo Morgana può tenere a giocare il povero Lancillotto.

Questo non è soddisfacente, però, perché l'operazione `PrepareMorganasCoin` svolge un sacco di lavoro solo per bloccare il valore `0.62` per l'argomento di input `morganaWinProbability` di `PrepareBiasCoin`. Se Morgana dovesse cambiare la sua strategia per avere un bias differente, usando questo approccio avremmo bisogno di un'altra operazione standard per rappresentarlo. Facendo un passo indietro, vediamo quello che *effettivamente fa* `PrepareMorganasCoin`. Inizia con un'operazione `PrepareBiasCoin : (Double, Qubit) => Unit` e la racchiude in un'operazione di tipo `Qubit => Unit` bloccando l'argomento `Double` a `0.62`. Cioè, rimuove uno degli argomenti di `PrepareBiasCoin` fissando il valore di quell'input a `0.62`.

Fortunatamente, Q# fornisce una comoda scorciatoia per creare nuove funzioni e operazioni bloccando alcuni (non tutti!) dei loro input. Usando questa scorciatoia, nota come *applicazione parziale*,

possiamo riscrivere `PrepareMorganasCoin` (vedi Listato 7.6) in una forma più leggibile:

```
let flip = GetNextRandomBit(PrepareBiasedCoin(0.62, _));
```

Il segno _ indica che *manca* una parte dell'input per `PrepareBiasCoin`. Diciamo che `PrepareBiasCoin` è stata applicata parzialmente. Mentre `PrepareBiasCoin` aveva il tipo `(Double, Qubit) => Unit`, poiché abbiamo specificato la parte `Double` dell'input, `PrepareBiasCoin(0.62, _)` ha il tipo `Qubit => Unit`, rendendola compatibile con le nostre modifiche a `GetNextRandomBit`.

SUGGERIMENTO

L'applicazione parziale del linguaggio Q# è simile a `functools.partial` di Python e a _ in Scala.

Un altro modo per pensare all'applicazione parziale: è un modo per creare nuove funzioni e operazioni specializzando funzioni e operazioni preesistenti.

Listato 7.7 Uso dell'applicazione parziale per specializzare un'operazione.

```
function BiasedPreparation(headsProbability : Double) : (Qubit => Unit)
{ (1)
    return PrepareBiasedCoin(headsProbability, _); (2)

}
```

(1) Il tipo di output è un'operazione che accetta un qubit e restituisce la tupla vuota. In pratica, `BiasPreparation` è una funzione che crea nuove operazioni!

(2) Esegue la nuova operazione passando `headsProbability` ma lasciando uno spazio vuoto (_) per il qubit target. Questo ci dà un'operazione che prende un singolo `Qubit` e lo sostituisce con lo spazio vuoto.

SUGGERIMENTO

Nel Listato 7.7, restituiamo `PrepareBiasCoin(headsProbability, _)` come un valore, proprio come possiamo far restituire `42` a una funzione o a un'operazione che ha `Int` come tipo di output. In Q#, le funzioni e le operazioni sono valori esattamente come `42`, `true` e `(3.14, "hello")`, e proprio come la funzione Python `(lambda x: x ** 2)` è un valore in Python. Formalmente, diciamo che le funzioni e le operazioni sono *valori di prima classe* in Q#.

Può sembrare un po' strano che `BiasPreparation` restituisca un'operazione da una funzione, ma questo è completamente coerente con la distinzione fra funzioni e operazioni descritta in precedenza, poiché `BiasPreparation` è ancora prevedibile. In particolare, `BiasPreparation(p)` restituisce sempre la stessa operazione per un dato `p`, indipendentemente da quante volte richiamiamo la funzione. Possiamo assicurarci della cosa notando che `BiasPreparation` applica solo parzialmente le operazioni, ma non le richiama mai.

ESERCIZIO 7.4 APPLICAZIONE PARZIALE

L'applicazione parziale funziona sia per le funzioni sia per le operazioni. Dimostratelo scrivendo una funzione `Plus` che somma due numeri interi, `n` e `m`, e un'altra funzione `PartialPlus` che accetta un input `n` e restituisce una funzione che somma `n` al suo input.

Suggerimento: come modello potete utilizzare il seguente frammento di codice:

```
function Plus(n : Int, m : Int) : Int {
// scrivete questa parte
}
function PartialPlus(n : Int) : (Int -> Int) {
// scrivete questa parte
}
```

Il gioco di Morgana in Q#

Ora che conosciamo le operazioni di prima classe e l'applicazione parziale, possiamo creare una versione più completa del gioco di Morgana.

Le librerie standard di Q#

Il Quantum Development Kit è dotato di varie librerie standard che esamineremo nel resto del libro. Nel Listato 7.8, per esempio, usiamo un'operazione `MResetZ` che misura un qubit (come `M`) e ne esegue il reset (come `Reset`). Questa operazione è offerta dal namespace `Microsoft.Quantum.Measurement`, una delle principali librerie standard fornite col Quantum Development Kit.

Un elenco completo delle operazioni e delle funzioni disponibili in tale namespace si trova su https://docs.microsoft.com/qsharp/api/qsharp/microsoft.quantum.measurement.

Per ora, però, non preoccupatevene troppo; vedremo altre librerie standard di Q# man mano che procederemo.

Listato 7.8 Listato Q# completo del PlayMorganasGame "truccato".

```
open Microsoft.Quantum.Math; (1)
open Microsoft.Quantum.Measurement;
operation PrepareBiasedCoin(winProbability : Double, qubit : Qubit) :
Unit {
    let rotationAngle = 2.0 * ArcCos(Sqrt(1.0 - winProbability)); (2)
    Ry(rotationAngle, qubit);
}
operation GetNextRandomBit(statePreparation : (Qubit => Unit)) : Result
{
    use qubit = Qubit();
    statePreparation(qubit); (3)
    return MResetZ(qubit); (4)
}
operation PlayMorganasGame(winProbability : Double) : Unit {
    mutable nRounds = 0;
    mutable done = false;
    let prep = PrepareBiasedCoin(winProbability, _); (5)
    repeat {
        set nRounds = nRounds + 1;
        set done = (GetNextRandomBit(prep) == Zero);
    }
    until done;
    Message($"It took Lancelot {nRounds} turns to get home.");
}
```

(1) Apre i namespace della libreria standard Q# per impiegare la matematica classica e la misurazione dei qubit.
(2) L'angolo di rotazione sceglie quanto è truccata la moneta.
(3) Usa l'operazione che abbiamo passato come `statePreparation` e la applica al `qubit`.
(4) L'operazione `MResetZ` è definita nel namespace `Microsoft.Quantum.Measurement`, aperto all'inizio. Misura il `qubit` nella base Z e poi applica le operazioni necessarie per riportare il qubit allo stato $|0\rangle$.
(5) Utilizza l'applicazione parziale per specificare quanto deve essere truccata la moneta per la nostra procedura di preparazione degli stati, ma non il `qubit` target. Mentre `PrepareBiasCoin` ha tipo `(Double, Qubit) => Unit`, `PrepareBiasCoin(0.2, _)` "riempie" uno dei due input, lasciando un'operazione di tipo `Qubit => Unit`, come previsto da `EstimateBias`.

La documentazione per le funzioni e le operazioni Q#

La documentazione per le funzioni e le operazioni Q# può essere fornita scrivendo piccoli documenti di testo formattati in modo speciale in commenti a tre barre (`///`) prima della dichiarazione di una funzione o di un'operazione. Questi documenti sono scritti in Markdown, un semplice linguaggio di formattazione del testo utilizzato su siti come GitHub, Azure DevOps, Reddit e Stack Exchange e da generatori di siti come Jekyll. Le informazioni contenute in commenti `///` vengono visualizzate quando passiamo il mouse sulle chiamate a quella funzione o operazione e possono essere utilizzate per creare riferimenti API simili a quelli presenti in https://docs.microsoft.com/qsharp/api.

Le varie parti differenti dei commenti `///` sono indicate con intestazioni di sezione come `/// #Summary`. Per esempio, possiamo documentare l'operazione `PrepareBiasCoin` del Listato 7.8 come segue:

```
/// # Summary
/// Prepares a state representing a coin with a given bias.
///
/// # Description
/// Given a qubit initially in the |0⟩ state, applies operations
/// to that qubit such that it has the state √p |0⟩ + √(1 − p) |1⟩,
/// where p is provided as an input.
```

```
/// Measurement of this state returns a One Result with probability p.
///
/// # Input
/// ## winProbability
/// The probability with which a measurement of the qubit should return One.
/// ## qubit
/// The qubit on which to prepare the state √p |0⟩ + √(1 − p) |1⟩.
operation PrepareBiasedCoin(winProbability : Double, qubit : Qubit) : Unit {
    let rotationAngle = 2.0 * ArcCos(Sqrt(1.0 - winProbability));
    Ry(rotationAngle, qubit);
}
```

Quando si utilizza IQ#, è possibile cercare i commenti di documentazione utilizzando il comando `?`. Per esempio, possiamo cercare la documentazione per l'operazione `X` eseguendo `X?` in una cella di input.

Per un riferimento completo sull'argomento, consultate https://docs.microsoft.com/azure/quantum/user-guide/language/statements/iterations#documentation-comments.

Per stimare il bias (il "trucco") di una determinata operazione di preparazione degli stati, possiamo eseguire ripetutamente l'operazione `PlayMorganasGame` e contare quante volte otteniamo uno `Zero`. Scegliamo un valore per `winProbability` ed eseguiamo l'operazione `PlayMorganasGame` con quel valore per vedere quanto tempo Lancillotto resterà bloccato:

```
In []: %simulate PlayMorganasGame winProbability=0.9 (1)

It took Lancelot 5 turns to get home.
```

(1) Possiamo passare degli input alle operazioni utilizzando il comando `%simulate` specificandoli dopo il nome dell'operazione che vogliamo simulare.

Provate a giocare con diversi valori di `winProbability`. Notate che se Morgana trucca seriamente il gioco, a Lancillotto servirà davvero molto tempo per tornare da Ginevra:

```
In []: %simulate PlayMorganasGame winProbability=0.999

It took Lancelot 3255 turns to get home.
```

Nel prossimo capitolo, svilupperemo le competenze apprese per provare il nostro primo esempio di algoritmo quantistico: l'algoritmo di

Deutsch-Jozsa.

Riepilogo

- Q# è un linguaggio di programmazione quantistico fornito con il Quantum Development Kit open source di Microsoft.
- I programmi quantistici in Q# sono suddivisi in funzioni, che rappresentano la logica classica e deterministica, e operazioni, che possono avere effetti collaterali come l'invio di istruzioni a dispositivi quantistici.
- Le funzioni e le operazioni sono valori di prima classe in Q# e possono essere passati come input ad altre funzioni e operazioni. Possiamo sfruttare questo fatto per combinare insieme le varie parti diverse dei programmi quantistici.
- Passando operazioni Q# nel nostro programma, possiamo estendere il nostro esempio di generatore quantistico di numeri casuali sviluppato nel Capitolo 2, per consentire il passaggio di operazioni che preparano stati differenti da $|+\rangle$; questo, a sua volta, ci consente di creare generatori truccati di numeri casuali.

Che cos'è un algoritmo quantistico?

Un'importante applicazione degli algoritmi quantistici consiste nell'ottenere accelerazioni nella risoluzione di problemi in cui abbiamo bisogno di cercare input per una funzione che stiamo cercando di conoscere. Tali funzioni potrebbero essere offuscate (come le funzioni *hash*) o computazionalmente difficili da valutare (cosa piuttosto comune nello studio di problemi matematici). In entrambi i casi, l'applicazione dei computer quantistici a tali problemi ci obbliga a capire come si programma e come si fornisce un input agli algoritmi quantistici. Per imparare a farlo, programmeremo ed eseguiremo un'implementazione dell'*algoritmo di Deutsch-Jozsa*, che ci permetterà di scoprire rapidamente le proprietà di funzioni sconosciute, utilizzando dispositivi quantistici.

Algoritmi classici e quantistici

> Algoritmo (sostantivo): una procedura passo a passo per risolvere un problema o ottenere un risultato.
> *- Dizionario Merriam-Webster*

Quando parliamo di programmazione classica, a volte diciamo che un programma implementa un *algoritmo*: cioè una sequenza di passi che può essere utilizzata per risolvere un problema. Per esempio, se vogliamo ordinare una lista, possiamo impiegare l'algoritmo *quicksort*,

indipendentemente dal linguaggio o dal sistema operativo che stiamo utilizzando. Spesso specifichiamo questi passaggi ad alto livello. Nell'esempio del quicksort, potremmo elencare i passaggi come segue.

1. Se la lista da ordinare è vuota o ha un solo elemento, restituiscila così com'è.
2. Scegli un elemento della lista da ordinare, chiamato *pivot*.
3. Separa tutti gli altri elementi della lista in quelli più piccoli e quelli più grandi del pivot.
4. Applica il quicksort a ogni nuova lista, in modo ricorsivo.
5. Restituisci la prima lista, poi il pivot e infine la seconda lista.

Questi passi servono come una guida per scrivere un'implementazione in un determinato linguaggio. Supponiamo di voler scrivere l'algoritmo quicksort in Python.

Listato 8.1 Un esempio di implementazione di quicksort.

```
def quicksort(xs):
    if len(xs) > 1: (1)
        pivot = xs[0] (2)
        left = [x in xs[1:] if x <= pivot] (3)
        right = [x in xs[1:] if x > pivot]
        return quicksort(left) + [pivot] + quicksort(right) (4)
    else:
        return xs
```

(1) Controlla il caso base, verificando se la lista contiene almeno due elementi.
(2) Sceglie il primo elemento che funge da pivot per il Passo 2.
(3) Codice Python che crea due nuove liste come descritto nel Passo 3.
(4) Concatena tutto di nuovo insieme, come descritto nei Passi 4 e 5

Un algoritmo ben scritto può aiutare molto a scrivere le implementazioni, rendendo chiari i passaggi che devono essere eseguiti. Gli algoritmi quantistici non fanno eccezione, sotto questo aspetto: elencano i passaggi che dobbiamo eseguire in qualsiasi implementazione.

DEFINIZIONE

Un *programma quantistico* è un'implementazione di un algoritmo quantistico costituito da un *programma classico* che invia istruzioni a un *dispositivo quantistico* per preparare un determinato stato o risultato della misurazione.

Come abbiamo visto nel Capitolo 7, quando scriviamo un programma Q#, stiamo scrivendo un programma classico che invia per noi delle istruzioni a una delle diverse macchine target, come illustrato nella Figura 8.1; esse poi restituiranno le loro misurazioni al nostro programma classico.

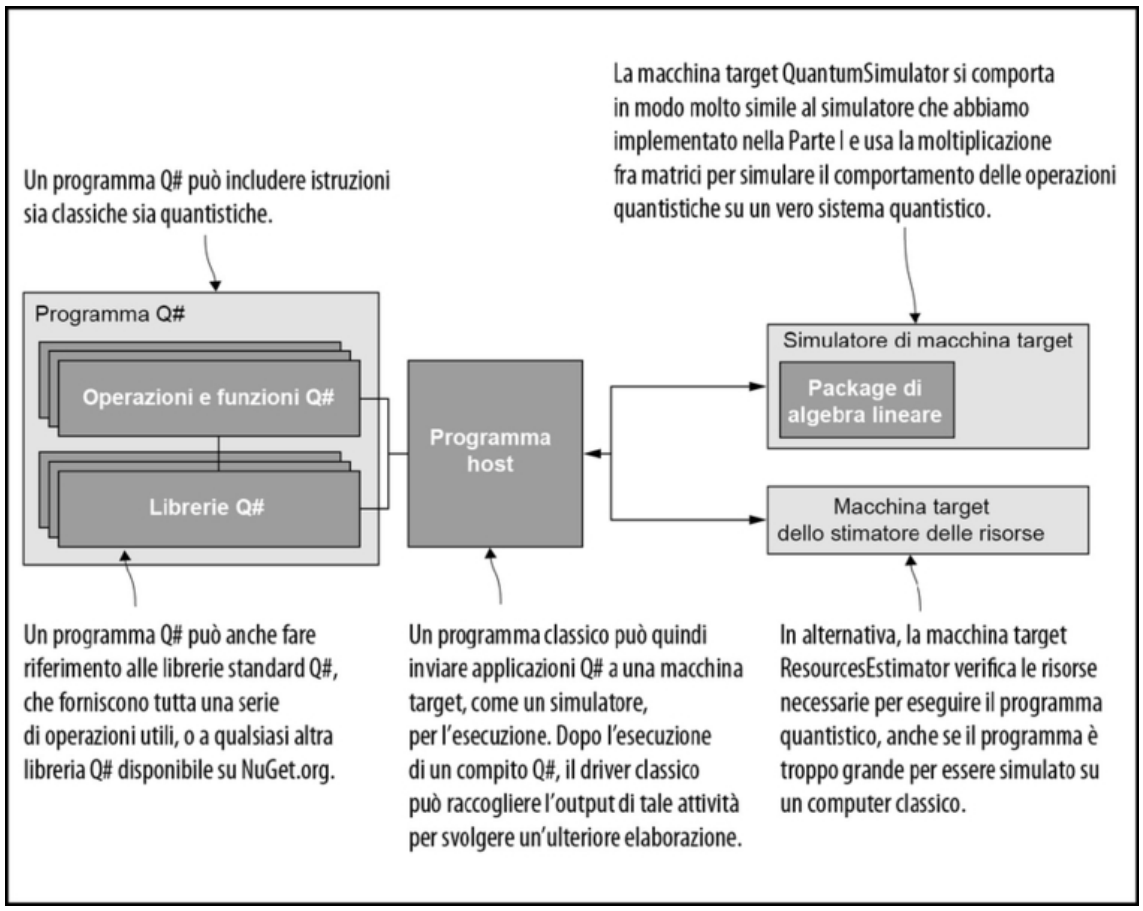


Figura 8.1 Stack software Microsoft Quantum Development Kit su un computer classico. Possiamo scrivere un programma Q# composto da funzioni e operazioni e che fa riferimento a qualsiasi libreria Q# che vogliamo includere. Un programma host può quindi coordinare le comunicazioni tra il nostro programma Q# e la macchina

target (che, per esempio, può essere un simulatore in esecuzione localmente sul nostro computer).

L'arte della programmazione quantistica

Non possiamo copiare gli stati quantistici, ma se sono il risultato dell'esecuzione di un programma, possiamo dire a qualcun altro quali passi devono compiere per preparare quegli stessi stati. Come abbiamo visto in precedenza, i programmi quantistici sono un tipo speciale di programma classico, quindi possiamo copiarli con una certa tranquillità. Come vedremo nel resto del libro, qualsiasi stato quantistico può essere approssimato o scritto esattamente in base all'output di un programma quantistico che inizia con copie dello stato $|0\rangle$. Per esempio, nel Capitolo 2, abbiamo preparato lo stato iniziale $|+\rangle$ di un generatore quantistico di numeri casuali con un programma costituito da una singola istruzione `H`.

In altre parole, possiamo considerare un programma come una ricetta che spiega come preparare un qubit. Dato un qubit, non possiamo determinare quale ricetta è stata utilizzata per prepararlo, ma possiamo copiare tale ricetta quanto vogliamo.

Mentre i passaggi nell'esecuzione di `quicksort` chiedono all'interprete Python di confrontare i valori e spostarli in memoria, i passaggi in un programma Q# chiedono alle macchine target di applicare rotazioni e misurazioni ai qubit in un dispositivo. Come potete vedere nella Figura 8.1, possiamo usare un programma host per inviare a ciascuna macchina target le applicazioni Q# da eseguire. Per ora, continueremo a usare come programma host il plugin IQ# per Jupyter Notebook; nel prossimo capitolo vedremo come usare C# per scrivere anche i nostri programmi host.

Il più delle volte in questo libro, siamo interessati a simulare programmi quantistici, quindi utilizziamo la macchina target `QuantumSimulator`. Questo simulatore funziona in modo molto simile a quelli che abbiamo sviluppato nei Capitoli 2 e 4, poiché esegue le istruzioni, come quella di Hadamard `H`, moltiplicando gli stati quantistici con operatori unitari come H.

SUGGERIMENTO

Come nei capitoli precedenti, usiamo stili di caratteri differenti per distinguere le istruzioni, come `H`, dalle matrici unitarie, come H, che usiamo per simulare tali istruzioni.

La macchina target `ResourcesEstimator` ci consente di non eseguire un programma quantistico, ma di ottenere stime di quanti qubit sarebbero necessari per eseguirlo. Questo è utile per programmi più grandi, che non possono essere simulati in modo classico o eseguiti sull'hardware disponibile, per aiutarci a capire quanti qubit richiederà la loro esecuzione; vedremo meglio questa macchina target più avanti.

Poiché le applicazioni Q# inviano istruzioni alle macchine target che utilizziamo per eseguirle, è facile riutilizzare il codice Q# in un secondo momento su altre macchine target che condividono lo stesso set di istruzioni. La macchina target `QuantumSimulator`, per esempio, utilizza le stesse istruzioni che ci aspettiamo accetterà l'hardware quantistico, una volta che sarà disponibile; quindi possiamo sottoporre a test i programmi Q# sui simulatori anche utilizzando piccole istanze di problemi, e poi eseguire quegli stessi programmi su hardware quantistico, in un secondo momento.

Ciò che rimane in comune tra queste diverse macchine target e applicazioni è che per raggiungere un obiettivo abbiamo bisogno di scrivere il programma che invii le istruzioni alla macchina target. Il nostro compito come programmatori quantistici è quindi quello di assicurarci che queste istruzioni abbiano l'effetto di risolvere un problema utile.

SUGGERIMENTO

Il modo in cui utilizziamo i simulatori per sottoporre a test i programmi Q# è un po' simile a come utilizziamo i simulatori, per sottoporre a test i programmi per altri hardware specializzati come gli FPGA (field-programmable gate array), o gli emulatori, per sottoporre a test le app per dispositivi mobili dai nostri desktop e laptop. La differenza principale è che

possiamo usare un computer classico per simulare un computer quantistico solo per un numero molto piccolo di qubit e per programmi limitati.

Il tutto è molto più facile quando abbiamo un algoritmo che ci guida per organizzare i passaggi che devono avvenire nei dispositivi classici e quantistici. Nello sviluppo di nuovi algoritmi quantistici, possiamo utilizzare gli effetti quantistici, come l'*entanglement*, che abbiamo visto nel Capitolo 4.

SUGGERIMENTO

Per sfruttare il nostro hardware quantistico, *dobbiamo* utilizzare le proprietà quantistiche specifiche dell'hardware. Altrimenti, avremo solo un computer classico più costoso e più lento.

L'algoritmo di Deutsch-Jozsa: piccoli miglioramenti per la ricerca

Quale potrebbe essere un buon esempio di *algoritmo quantistico* che sfrutta il nostro nuovo e magnifico hardware quantistico? Nei Capitoli 4 e 7 abbiamo imparato che immaginare dei giochi spesso aiuta. Per trovare un gioco per questo capitolo, facciamo un nuovo viaggio a Camelot, dove Merlino si trova ad affrontare una prova.

La Dama del Lago (quantistico)

Merlino, il famoso e saggio mago, ha appena incontrato Viviana, la Dama del Lago. Viviana, alla ricerca di un mentore adatto per il prossimo re d'Inghilterra, ha deciso di mettere alla prova Merlino per scoprire se è all'altezza del compito. Due acerrimi rivali, Artù e Mordred, sono in lizza per il trono, e se Merlino accetta il compito di Viviana, deve scegliere a chi fare da mentore.

Da parte sua, a Viviana non importa chi diventa re, a patto che Merlino possa dare loro un saggio consiglio. Viviana *deve però*

scoprire se Merlino, l'istruttore nominato per il nuovo re, sarà affidabile e coerente nella sua leadership.

Dal momento che Viviana condivide la nostra passione per i giochi, ha deciso di sottoporre un gioco a Merlino, per verificare se sarà un buon mentore. Il gioco di Viviana, *Kingmaker*, verifica se Merlino è coerente nel suo ruolo di consigliere del re. Per giocare a Kingmaker, Viviana dà a Merlino il nome di uno dei due acerrimi rivali, e Merlino deve rispondere se il candidato di Viviana debba essere il vero erede al trono. Ecco le regole.

- A ogni round, Viviana pone a Merlino una singola domanda del tipo: "Il *potenziale erede* dovrebbe essere re?".
- Merlino deve rispondere "sì" o "no", senza fornire ulteriori informazioni.

Ogni round fornisce a Viviana ulteriori informazioni sul regno dei mortali, pertanto intende fare tutte le domande necessarie per scoprire se Merlino è degno di fiducia. I suoi obiettivi sono i seguenti.

- Verificare che Merlino sia un buon mentore per il nuovo re d'Inghilterra.
- Porre il minor numero possibile di domande per verificarlo.
- Evitare di sapere a chi Merlino vorrà fare da mentore.

A questo punto, Merlino ha quattro possibili strategie.

1. Rispondere "sì" quando gli viene chiesto se Artù debba essere re e "no" altrimenti (buon mentore).
2. Rispondere "sì" quando gli viene chiesto se Mordred debba essere re e "no" altrimenti (buon mentore).
3. Rispondere "sì" indipendentemente dalla domanda di Viviana (cattivo mentore).
4. Rispondere "no" indipendentemente dalla domanda di Viviana (cattivo mentore).

Possiamo considerare le strategie di Merlino usando ancora una volta il concetto di tavola di verità. Supponiamo, per esempio, che Merlino abbia deciso di essere del tutto inutile e di negare il trono a entrambi i candidati. Potremmo trascriverlo usando la tabella di verità presentata nella Tabella 8.1.

Tabella 8.1 Tabella di verità per una possibile strategia di Kingmaker: Merlino dice sempre no.

Input (Viviana)	Output (Merlino)
"Mordred dovrebbe essere re?"	"No."
"Artù dovrebbe essere re?"	"No."

A questo punto, Viviana avrebbe ragione di lamentarsi della saggezza di Merlino come mentore! Merlino non è coerente con il suo incarico di scegliere tra Artù e Mordred. Anche se a Viviana non interessa chi sceglierà Merlino, sicuramente deve scegliere di fare da mentore a uno di loro e prepararli per il trono.

Viviana ha bisogno di una strategia per determinare nel minor numero di round di gioco possibile se Merlino applica la strategia 1 o 2 (buon mentore) oppure 3 o 4 (cattivo mentore). Potrebbe semplicemente porre entrambe le domande: "Mordred dovrebbe essere re?" e "Artù dovrebbe essere re?" e poi confrontare le risposte, ma questo porterebbe Viviana a sapere con certezza chi ha scelto Merlino come re. A ogni domanda, Viviana impara una cosa di più sugli affari mortali del regno, e non va bene.

Anche se sembrerebbe che il gioco di Viviana sia destinato a costringerla a conoscere la scelta di Merlino, in realtà è fortunata. Trattandosi di un lago quantistico, come vedremo in tutto il resto di questo capitolo Viviana può porre anche *una sola domanda* che le dirà *solo* se Merlino si è impegnato nel suo ruolo di mentore ma non chi ha scelto.

Dal momento che non abbiamo a disposizione un lago quantistico, proviamo a modellare ciò che Viviana sta facendo con le istruzioni quantistiche in Q# sul nostro computer classico e poi simuliamolo. Rappresentiamo la strategia di Merlino con una funzione classica *f*, che prende come input *x* la domanda di Viviana. Cioè, scriveremo *f*(Artù) per intendere "Che cosa risponde Merlino quando gli viene chiesto se Artù debba essere re?". Notate che poiché Viviana porrà una sola delle due domande, la domanda che farà è un esempio di bit. A volte è utile scrivere quel bit utilizzando le etichette "0" e "1", mentre altre volte è utile etichettare il bit di input di Viviana utilizzando i valori booleani "False" e "True". Dopotutto, "1" sarebbe una risposta piuttosto ambigua a una domanda come "Mordred dovrebbe essere re?".

Usando i bit, scriviamo *f*(0) = 0 per indicare che se Viviana domanda a Merlino, "Mordred dovrebbe essere re?" e la sua risposta è no. La Tabella 8.2 mostra come possiamo mappare le domande di Viviana ai valori booleani.

Tabella 8.2 Codifica della domanda di Viviana come un bit.

Domanda di Viviana	Rappresentazione a bit	Rappresentazione a valore booleano
"Mordred dovrebbe essere re?"	0	False
"Artù dovrebbe essere re?"	1	True

Se non avesse risorse quantistiche, per conoscere la strategia di Merlino, Viviana dovrebbe provare entrambi gli input di *f*, cioè dovrebbe porre a Merlino entrambe le domande. Ma provando tutti gli input, Viviana conoscerebbe l'intera strategia di Merlino e, come abbiamo detto, Viviana non è interessata a conoscerla.

Invece di chiedere a Merlino sia di Mordred sia di Artù, possiamo implementare un algoritmo quantistico in Q#, che sfrutta gli effetti quantistici per capire se Merlino è un buon mentore ponendogli una sola domanda. Utilizzando i simulatori forniti con il Quantum Development Kit, possiamo persino eseguire il nostro nuovo programma Q# sul nostro laptop o desktop. Nel resto del capitolo, vedremo un esempio del modo in cui scrivere questo algoritmo quantistico, chiamato algoritmo di Deutsch-Jozsa (vedi Figura 8.2).

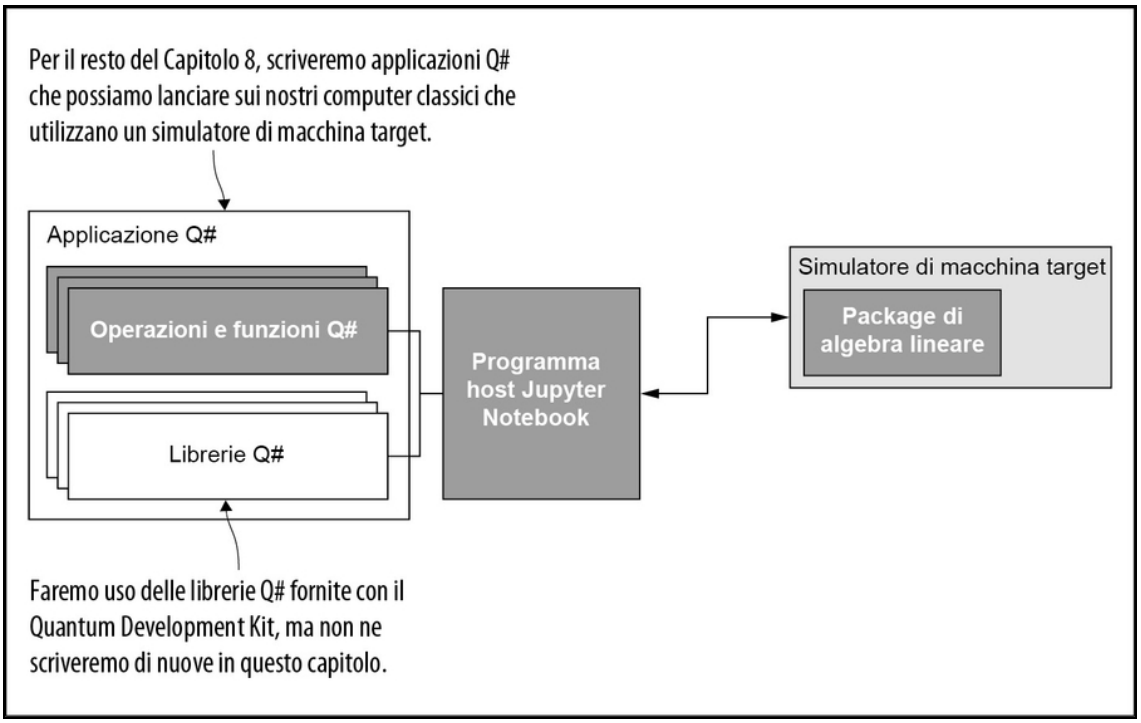


Figura 8.2 Per questo capitolo lavoreremo nello stack software del Microsoft Quantum Development Kit, scrivendo programmi Q# che vengono eseguiti tramite un host Jupyter Notebook su un simulatore di macchina target.

Vediamo come sarà il nostro programma quantistico. I possibili input e output per *f* (strategia di Merlino) sono `True` e `False`. Possiamo scrivere una tabella di verità per *f* usando gli input e gli output che otteniamo quando richiamiamo *f*. Per esempio, se *f* è l'operazione NOT classica (spesso indicata con ¬), allora osserveremo che *f*(`True`) è `False`,

e viceversa. Come vediamo nella Tabella 8.3, usare una classica operazione NOT come strategia nel nostro gioco corrisponde a scegliere Mordred come re.

Tabella 8.3 Tabella di verità per l'operazione NOT classica.

Input	Output
`True` ("Artù dovrebbe essere re?")	`Falso` ("No.")
`False` ("Mordred dovrebbe essere re?")	`True` ("Sì")

Ci sono quattro possibili opzioni per la definizione della nostra funzione *f*, ognuna delle quali rappresenta una delle quattro strategie a disposizione di Merlino, riassunte nella Figura 8.3.

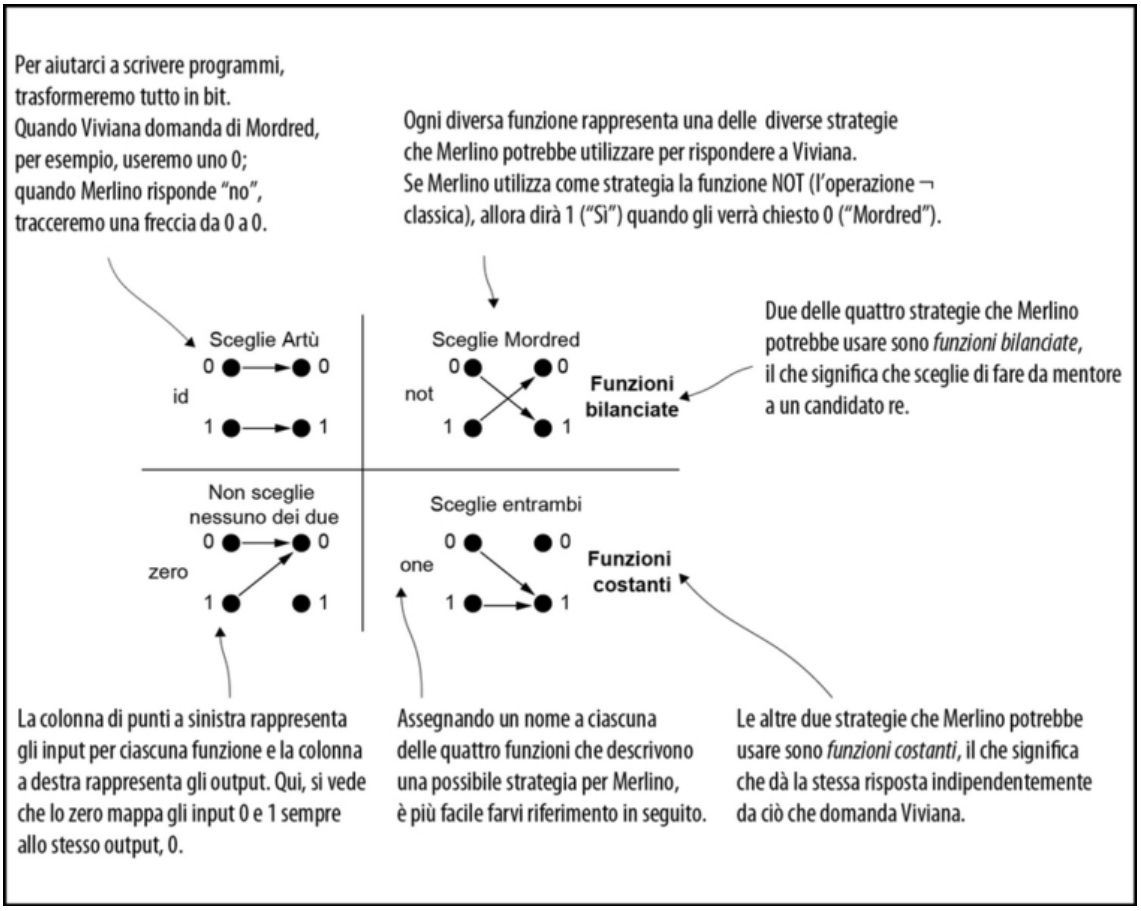


Figura 8.3 Quattro diverse funzioni da un bit a un bit. Chiamiamo bilanciate le due funzioni della riga superiore, perché gli input mappati a 0 equivalgono agli input

mappati a 1. Chiamiamo costanti le due funzioni della riga inferiore, poiché tutti gli input sono mappati a un singolo output.

Due di queste funzioni, etichettate `id` e `not` per comodità, inviano gli input `0` e `1` a output differenti; chiamiamo queste funzioni *bilanciate*. Nel nostro piccolo gioco, rappresentano i casi in cui Merlino sceglie esattamente una persona come re. La Tabella 8.4 elenca tutti i casi.

Tabella 8.4 Classificazione delle strategie di Merlino come costanti o equilibrate.

Stategia di Merlino	Funzione	Tipo	Supera la sfida di Viviana?
Sceglie Artù	`id`	Bilanciato ($f(0) \neq f(1)$)	Sì
Sceglie Mordred	`not`	Bilanciato ($f(0) \neq f(1)$)	Sì
Non sceglie	`zero`	Costante ($f(0) = f(1)$)	No
Sceglie entrambi	`one`	Costante ($f(0) = f(1)$)	No

Al contrario, le funzioni etichettate con `zero` e `one` sono *costanti*, poiché inviano entrambi gli input allo stesso output. Le funzioni costanti rappresentano le strategie in cui Merlino è decisamente inutile, in quanto ha scelto come re entrambi (un buon modo per iniziare una guerra) o nessuno dei due.

Classicamente, per determinare se una funzione è costante o bilanciata (quindi, rispettivamente, se Merlino è un cattivo o un buon mentore), dobbiamo determinare l'intera funzione costruendo la sua tabella di verità. Ricordate: Viviana vuole assicurarsi che Merlino sia un buon mentore. Se Merlino adotta una strategia rappresentata da una funzione costante, non sarà un buon mentore. Osservando le tabelle di verità delle funzioni `id` e `one`, rispettivamente le Tabelle 8.5 e 8.6, possiamo vedere come esse descrivano se Merlino sta adottando una

strategia che gli permetterà di essere un buon mentore o un cattivo mentore.

Tabella 8.5 Tabella di verità per la funzione id, un esempio di funzione bilanciata.

Input	Output
`True` (“Artù dovrebbe essere re?”)	`True` (“Sì”)
`False` (“Mordred dovrebbe essere re?”)	`False` (“No”)

Tabella 8.6 Tabella di verità per la funzione one, un esempio di funzione costante.

Input	Output
`True` (“Artù dovrebbe essere re?”)	`True` (“Sì”)
`False` (“Mordred dovrebbe essere re?”)	`True` (“Sì”)

La difficoltà di Viviana nel cercare di capire se Merlino è un buon mentore o un cattivo mentore (cioè, se f è bilanciata o costante) è che la qualità di mentoring di Merlino è una sorta di *proprietà globale* della sua strategia. Non c’è modo di guardare un singolo output di f e concludere qualcosa su ciò che f produrrebbe per più input. Se abbiamo accesso solo a f, allora Viviana è bloccata: è costretta a costruire l’intera tabella di verità per decidere se la strategia di Merlino è costante o bilanciata.

Ma se possiamo rappresentare la strategia di Merlino con un programma quantistico, possiamo usare gli effetti quantistici che abbiamo imparato finora. Utilizzando il quantum computing, Viviana può anche scoprire *solo* se la strategia di Merlino è costante o bilanciata, senza scoprire quale strategia sta utilizzando. Dal momento che non siamo interessati alle informazioni fornite dalla tabella di verità, ma solo al fatto che Merlino sia o non sia un buon mentore, l’uso degli effetti quantistici può aiutarci a imparare ciò che ci interessa. Con il nostro algoritmo quantistico, possiamo farlo con una

sola chiamata a funzione, senza bisogno di apprendere nessun'altra informazione aggiuntiva, che non ci interessa. Non chiedendo tutti i dettagli della tabella di verità ma solo cercando le proprietà più generali della nostra funzione, possiamo utilizzare al meglio le nostre risorse quantistiche.

La potenza del quantum computing

Se vogliamo usare un computer *classico* per scoprire se una funzione è costante o bilanciata, dobbiamo risolvere un problema difficile: identificare esattamente quali funzioni abbiamo. Al contrario, la meccanica quantistica ci consente di risolvere anche solo il problema che ci interessa (costante o bilanciata) senza risolvere il problema più impegnativo che un computer classico deve necessariamente risolvere.

Questo è un esempio di un modello che vedremo in tutto il libro, in cui la meccanica quantistica ci consente di specificare algoritmi meno potenti di quelli che possiamo esprimere in modo classico.

Per fare ciò, useremo l'algoritmo di Deutsch-Jozsa, che utilizza una *singola domanda* per la nostra rappresentazione quantistica della strategia di Merlino per scoprire se è o non è un buon mentore. Il vantaggio non è molto pratico (un risparmio di una sola domanda), ma per ora va bene così; vedremo algoritmi più pratici più avanti nel libro. Per ora, l'algoritmo di Deutsch-Jozsa è un ottimo punto di partenza per imparare a implementare algoritmi quantistici e, ancora più importante, quali strumenti possiamo usare per capire che cosa fanno gli algoritmi quantistici.

Gli oracoli: come rappresentare le funzioni classiche negli algoritmi quantistici

Vediamo come vanno le cose nel lago quantistico di Viviana. Mentre ci tuffiamo per una nuotata, affrontiamo una domanda alquanto immediata: come possiamo usare i qubit per implementare la funzione *f* che rappresenta la strategia di Merlino? Dal paragrafo precedente, abbiamo visto che la funzione classica *f* è la nostra descrizione di una strategia impiegata da Merlino per giocare ogni round di Kingmaker. Poiché *f* è una funzione classica, è facile tradurla in un insieme di azioni che Merlino intraprenderà: Viviana dà a Merlino un singolo bit classico (la sua domanda) e Merlino restituisce a Viviana un bit classico (la sua risposta).

Per evitare di immischiarsi negli affari dei mortali, Viviana ora vuole invece utilizzare l'algoritmo di Deutsch-Jozsa. Dal momento che abita nelle profondità di un intero lago quantistico, Viviana può assegnare con facilità un qubit a Merlino. Fortunatamente per noi, Merlino sa comunicare tramite qubit, ma dobbiamo ancora capire che cosa farà Merlino con i qubit di Viviana per attuare la sua strategia.

Il problema è che non possiamo passare qubit alla funzione *f* che usiamo per rappresentare la strategia di Merlino: *f* accetta e restituisce bit classici, non qubit. Affinché Merlino attui la sua strategia per guidare ciò che fa con i qubit di Viviana, dobbiamo trasformare la strategia di Merlino, *f*, in un tipo di programma quantistico detto *oracolo*. Fortunatamente per noi, Merlino gioca il ruolo di un oracolo piuttosto bene.

NOTA

Dalla descrizione di Merlino fatta da T.H. White, apprendiamo che vive la propria vita procedendo a ritroso nel tempo. Lo rappresenteremo assicurandoci che tutto ciò che Merlino fa sia *unitario*. Come abbiamo visto nel Capitolo 2, una conseguenza è che le trasformazioni applicate da Merlino sono *reversibili*. In particolare, Merlino non sarà in grado di misurare i qubit di Viviana, poiché la misurazione non è reversibile. Tale privilegio è solo di Viviana.

Per capire quello che dobbiamo fare per modellare le azioni di Merlino come un oracolo, dobbiamo conoscere due cose:

1. quale trasformazione dovrebbe applicare Merlino ai qubit di Viviana in base alla sua strategia;
2. quali operazioni quantistiche dovrà applicare Merlino per implementare tale trasformazione.

Matrici unitarie e tavole di verità

Un altro modo per dire quello che dobbiamo fare nel Passo 1 è che dobbiamo trovare una matrice unitaria che rappresenti ciò che fa Merlino, in modo simile a come abbiamo usato le funzioni classiche, come f, per rappresentare ciò che ha fatto Merlino quando Viviana gli ha dato dei bit classici. Come abbiamo visto nel Capitolo 2, le matrici unitarie stanno al quantum computing come le tabelle di verità stanno al calcolo classico: ci dicono l'effetto di un'operazione quantistica per ogni possibile input. Una volta che troviamo la giusta matrice unitaria, nel Passo 2 scopriremo la sequenza di operazioni quantistiche che possiamo fare e che sarà descritta da tale matrice unitaria.

Le trasformazioni di Merlino

Per completare il Passo 1, dobbiamo trasformare le funzioni come f in matrici unitarie, quindi iniziamo ricapitolando che cosa può essere f. Le possibili strategie che Merlino può usare sono rappresentate dalle funzioni `id`, `not`, `zero` e `one` (vedi Figura 8.4).

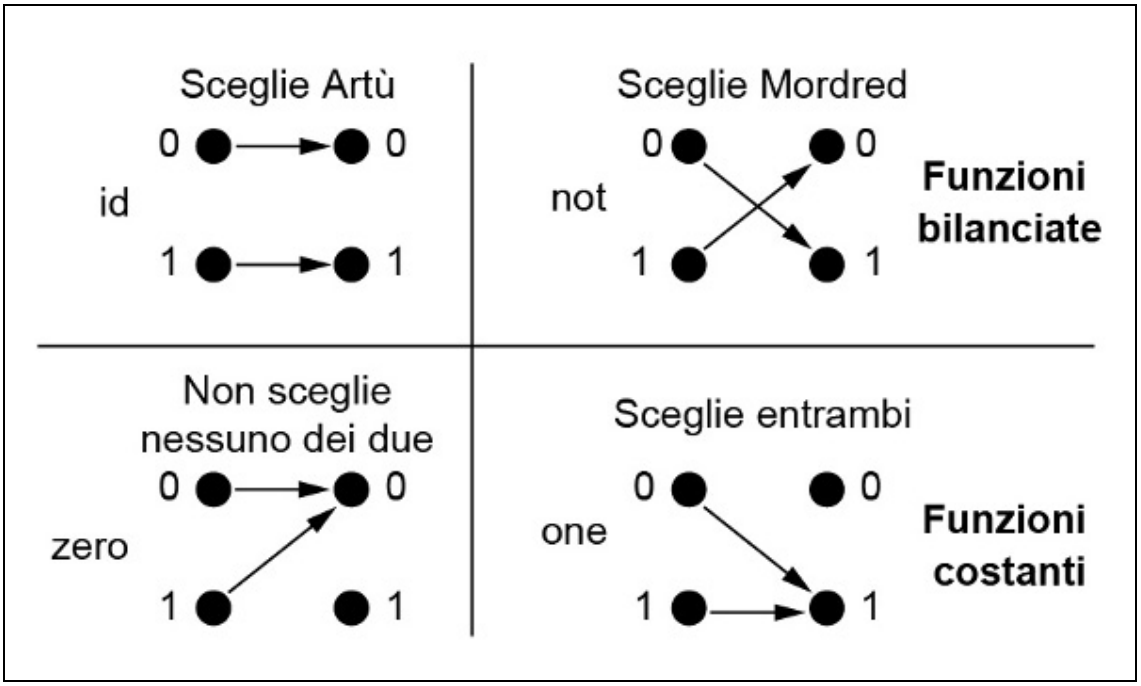


Figura 8.4 Quattro diverse funzioni da un bit a un bit.

Per le due funzioni bilanciate `id` e `not` nella Figura 8.4 possiamo rispondere facilmente alla prima domanda. I programmi quantistici per `id` e `not` possono essere implementati come operazioni di rotazione, cosa che semplifica la loro trasformazione in operazioni quantistiche. L'operazione NOT quantistica, per esempio, è una rotazione di 180° attorno all'asse X, che scambia gli stati $|0\rangle$ e $|1\rangle$.

SUGGERIMENTO

Come ricordiamo dal Capitolo 3, l'operazione quantistica X, rappresentata dalla matrice unitaria $X = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$, applica una rotazione di 180° attorno all'asse X. Questa operazione implementa un NOT quantistico: dal momento che $X|0\rangle = |1\rangle$ e $X|1\rangle = |0\rangle$, possiamo scriverla utilizzando l'operatore ¬ (NOT) del Capitolo 2, come $X|x\rangle = |\neg x\rangle$.

Sebbene qualsiasi rotazione possa essere annullata ruotando della stessa entità nella direzione opposta, incontriamo maggiori difficoltà con le funzioni costanti `zero` e `one`. Né `zero` né `one` possono essere

implementate direttamente come rotazioni, quindi abbiamo ancora un po' di lavoro da fare. Per esempio, se *f* è `zero`, allora gli output di *f*(0) e *f*(1) sono entrambi `0`. Se abbiamo solo l'output `0`, non possiamo dire se abbiamo ottenuto quell'output dall'input `0` o `1` (vedi Figura 8.5).

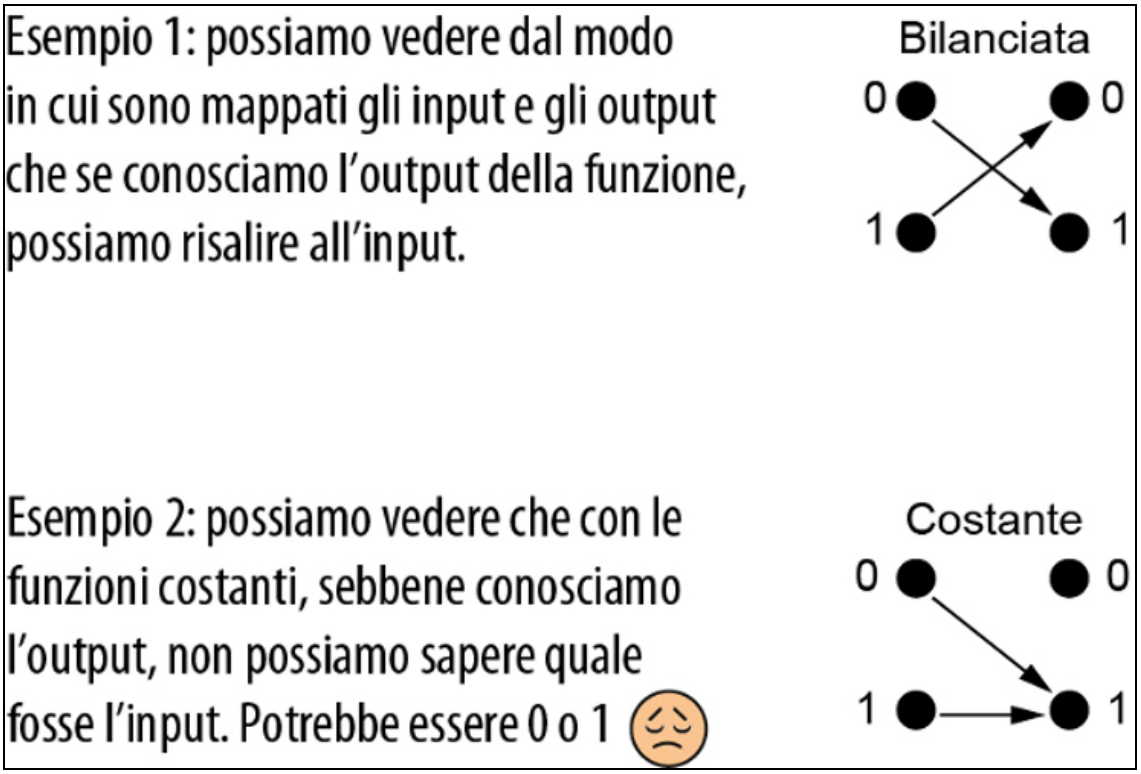


Figura 8.5 Perché non possiamo invertire le funzioni costanti zero o one? Fondamentalmente, perdiamo le informazioni sull'input impiegato, dal momento che tutti gli input vengono mappati su un singolo output.

NOTA

Una volta applicato `zero` o `one`, perdiamo tutte le informazioni sull'input.

Poiché `one` e `zero` sono entrambe irreversibili e le operazioni valide sui qubit devono essere reversibili, Merlino ha bisogno di un altro modo per rappresentare funzioni come *f* in algoritmi quantistici come quello necessario per la sfida di Viviana. D'altra parte, se possiamo rappresentare la strategia di Merlino con una funzione classica

reversibile, invece di f, sarà molto più facile scrivere una rappresentazione quantistica della sua strategia. Ecco la nostra strategia per rappresentare le funzioni classiche come oracoli quantistici.

1. Trova un modo per rappresentare la nostra funzione classica irreversibile con una funzione classica reversibile.
2. Scrivi una trasformazione sugli stati quantistici usando la funzione classica reversibile.
3. Scopri quali operazioni quantistiche possiamo fare per ottenere tale trasformazione.

Usiamo l'approccio collaudato, di indovinare e poi controllare se siamo in grado di progettare una funzione classica reversibile valida. Il modo più semplice per capire se abbiamo ricevuto uno `0` o un `1` come input consiste nel registrarlo da qualche parte. Quindi creiamo una nuova funzione che restituisca due bit invece di uno.

Per il nostro primo tentativo, cerchiamo di registrare e conservare l'input: $g(x) = (x, f(x))$.

Per esempio, se Merlino usa la strategia *one* (cioè, dice "sì" a Viviana indipendentemente da ciò che chiede), allora $f(x) = 1$ e $g(x) = (x, f(x)) = (x, 1)$.

Questo ci avvicina molto al risultato, poiché ora possiamo dire se abbiamo iniziato con un input 0 o 1. Ma non ci siamo ancora, dal momento che g ha due output e un input (vedi Figura 8.6).

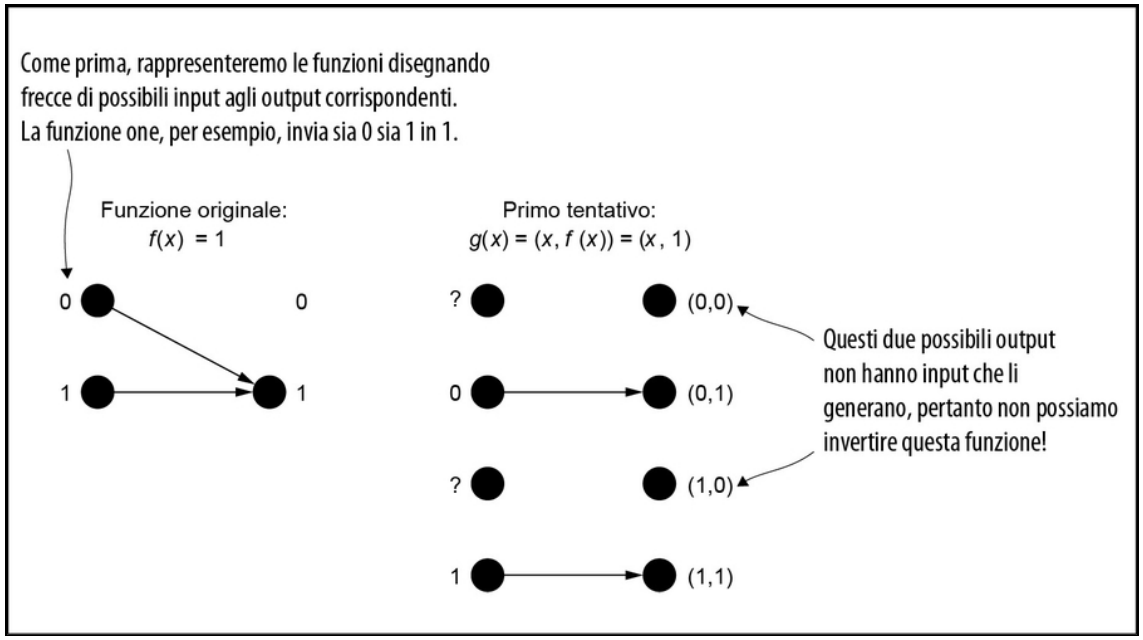


Figura 8.6 Primo tentativo: conservare input e output con g(x). Utilizzando questo approccio, alcune combinazioni di output non possono essere raggiunte da nessun input (per esempio, nessun input produce l'output (1,0)). Quindi è impossibile invertire la funzione, poiché non c'è alcun input corrispondente a quegli output.

Per usare *g* come strategia, Merlino dovrebbe restituire a Viviana più qubit di quanti lei gli ha dato, ma lei è la custode sia delle spade sia dei qubit. Più tecnicamente, invertire *g* distruggerebbe le informazioni, poiché richiederebbe due input e restituirebbe un solo output.

Provando ancora una volta, definiamo una nuova funzione classica *h* che prende due input e restituisce due output, $h(x, y)$. Consideriamo ancora una volta l'esempio in cui descriviamo la strategia di Merlino con la funzione $f(x) = 1$. Poiché *g* ci ha quasi portato alla soluzione, scegliamo *h* tale che $h(x, 0) = g(x)$. Abbiamo visto dal nostro primo tentativo che quando Merlino usa la strategia $f(\mathrm{x}) = 1$, allora $g(x) = (x, 1)$, pertanto abbiamo che $h(x, 0) = (x, 1)$. Ora dobbiamo solo definire che cosa succede quando passiamo $y = 1$ a *h*. Se vogliamo che *h* sia reversibile, abbiamo bisogno che a $h(x, 1)$ sia assegnato qualcosa di diverso da $(x, 1)$. Un modo per farlo è far sì che $h(x, y) = (x, \neg y)$ in

modo che $h(x, 1) = (x, 0) \neq (x, 1)$. Questa scelta è particolarmente utile, poiché applicando h due volte si ottiene il nostro input originale, $h(h(x, y)) = h(x, \neg y) = (x, \neg\neg y) = (x, y)$. Se la cosa vi sembra un po' ingarbugliata, date un'occhiata alla sua rappresentazione visuale nella Figura 8.7.

Ora che sappiamo come fare una funzione classica *reversibile* per ogni strategia, cerchiamo di terminare scrivendo un programma quantistico per la nostra funzione reversibile. Nel caso di `one`, abbiamo visto che h inverte il suo secondo input, $h(x, y) = (x, \neg y)$. Possiamo quindi scrivere un programma quantistico che fa la stessa cosa della nostra funzione classica reversibile semplicemente invertendo il secondo di due qubit di input. Come abbiamo visto nel Capitolo 4, possiamo farlo usando l'istruzione `x`, in quanto $X|x\rangle = |\neg x\rangle$.

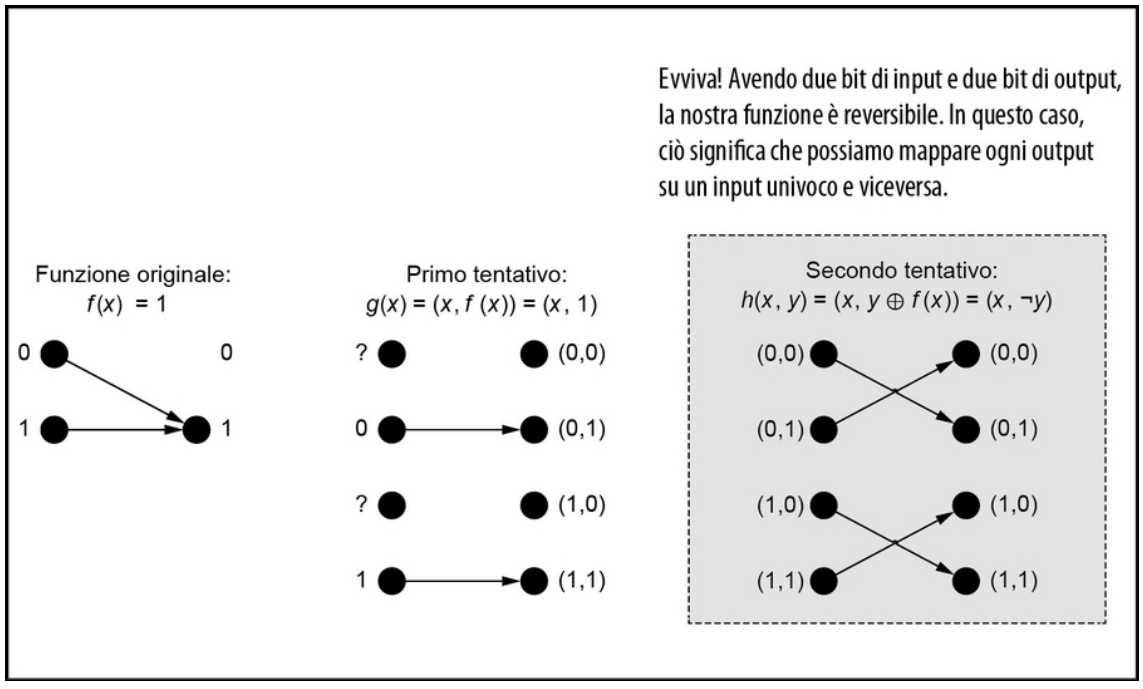


Figura 8.7 Secondo tentativo: h(x, y) è reversibile e ha lo stesso numero di input e output.

Generalizzare i risultati

Più in generale, possiamo eseguire un'operazione quantistica reversibile esattamente nello stesso modo in cui abbiamo reso reversibile la funzione classica *h* invertendo un bit di output sulla base dell'output della funzione irreversibile *f*. Possiamo definire la matrice unitaria (ovvero l'analogo quantistico di una tabella di verità) *Uf* per *f* per ogni stato di input esattamente allo stesso modo. La Figura 8.8 mostra un confronto di come possiamo creare questa definizione.

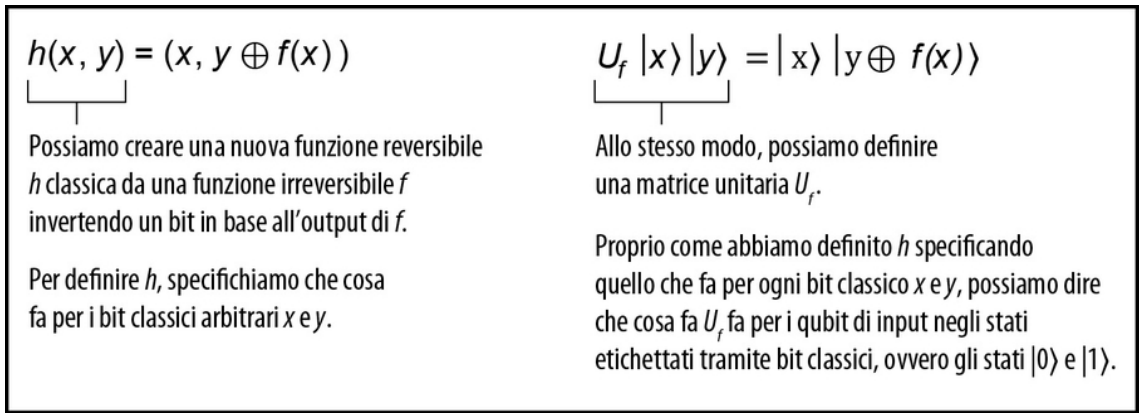


Figura 8.8 Costruzione di funzioni classiche reversibili e matrici unitarie a partire da funzioni classiche irreversibili. A sinistra, possiamo vedere che possiamo costruire una funzione classica reversibile da una irreversibile se teniamo traccia dell'input che stiamo dando alla funzione irreversibile. Possiamo fare lo stesso per una matrice unitaria che descrive un'operazione quantistica impiegando due registri di qubit: un registro tiene traccia dell'input e l'altro contiene l'output della nostra funzione irreversibile.

Definire *Uf* in questo modo permette di annullare la chiamata a *f*, poiché applicare *Uf* due volte ci fornisce l'identità (cioè la matrice unitaria per l'istruzione "non fare nulla"). Quando definiamo una matrice unitaria in questo modo applicando una funzione *f* in modo condizionale alle etichette per gli stati dei qubit, chiamiamo questa nuova operazione un *oracolo*.

DEFINIZIONE

Un *oracolo* è un'operazione quantistica rappresentata da una matrice unitaria *U*f che trasforma il suo stato di input come:

$$U_f \left|x\right\rangle \left|y\right\rangle = \left|x\right\rangle \left|y \oplus f(x)\right\rangle .$$

Il simbolo $\oplus$ rappresenta l'operatore OR esclusivo della normale logica booleana.

Non rimane che definire la sequenza di istruzioni che abbiamo bisogno di inviare per implementare ogni matrice unitaria *Uf*. Abbiamo visto le istruzioni di cui abbiamo bisogno per implementare un oracolo per `one`: un'istruzione `x` sul secondo qubit. Ora vediamo come scrivere oracoli per altre funzioni *f*. In questo modo, Merlino saprà che cosa fare, indipendentemente dalla sua strategia.

Approfondimento: perché si chiama oracolo?

Finora, abbiamo visto alcuni esempi di nomi stravaganti che il quantum computing trae dalle sue radici nella fisica, come bra, ket e teletrasporto. Ma non solo ai fisici piace divertirsi un po'! Una branca della computer science teorica, chiamata *teoria della complessità*, esplora ciò che è possibile fare in modo efficiente, anche in linea di principio, dati diversi tipi di macchine. Potreste aver sentito, per esempio, del problema "P e NP", un classico enigma nella teoria della complessità che chiede se i problemi in P sono difficili da risolvere come quelli in NP. La classe di complessità P è il gruppo di domande per le quali esiste un modo per rispondere con un algoritmo che richiede un tempo polinomiale. Al contrario, NP è il gruppo di domande per le quali possiamo verificare una potenziale risposta in tempo polinomiale, ma non sappiamo se possiamo trovare una risposta in tempo polinomiale.

Molti altri problemi nella teoria della complessità vengono posti dalla definizione di piccoli giochi o storie per aiutare i ricercatori a ricordare quali definizioni usare e dove. La nostra piccola storia con Merlino e Viviana segue proprio questa tradizione. In effetti, una delle storie più celebri del quantum computing si chiama MA (Merlino-Artù). I problemi di classe MA impiegano una storia in cui Artù deve porre a Merlino, un mago onnipotente ma inaffidabile, una serie di domande. Un problema decisionale sì/no è di classe MA se ogni volta che la risposta è "sì"

esiste una prova che Merlino può dare ad Artù e che Artù può facilmente controllare usando una macchina P e un generatore di numeri casuali.
Il nome *oracolo* si inserisce in questo tipo di racconto, in quanto qualsiasi classe di complessità A può essere trasformata in una nuova classe di complessità AB consentendo alle macchine A di risolvere un problema B in un unico passaggio, come se stessero consultando un oracolo. Gran parte della storia di problemi come quello di Deutsch-Jozsa deriva dal tentativo di capire in che modo il quantum computing influisce sulla complessità computazionale; pertanto, nel quantum computing sono state adottate molte delle convenzioni di denominazione e gran parte della terminologia.
Per ulteriori informazioni sulla teoria della complessità e sulle sue relazioni con il quantum computing, i buchi neri, il libero arbitrio e la filosofia greca, consultate *Quantum Computing Since Democritus* di Scott Aaronson (Cambridge University Press, 2013).

In generale, trovare una sequenza di istruzioni partendo da una matrice unitaria è un problema matematicamente difficile, noto come *sintesi unitaria*. Detto questo, in questo caso possiamo capirlo sostituendo ciascuna delle strategie di Merlino, f, nella nostra definizione di U_f e identificando quali istruzioni avranno quell'effetto: possiamo indovinare e controllare nello stesso modo in cui abbiamo trasformato la funzione `one` in un oracolo. Proviamo ad applicare la funzione `zero`.

ESERCIZIO 8.1 PROVATE A SCRIVERE UN ORACOLO

Quale sarebbe l'operazione oracolo (U_f) se f fosse `zero`?

Soluzione: risolviamolo un passo per volta.

1. Dalla definizione di U_f, sappiamo che $U_f|xy\rangle = |x\rangle\ |y \oplus f(x)\rangle$.
2. Sostituendo `zero` per f, $f(x) = 0$, otteniamo che $U_f|xy\rangle = |x\rangle\ |y \oplus 0\rangle$.
3. Possiamo usare che $y \oplus 0 = y$ per semplificare ulteriormente, ottenendo che $U_f|xy\rangle = |x\rangle\ |y\rangle$.
4. A questo punto, notiamo che U_f non fa nulla al suo stato di input, quindi possiamo implementarla… non facendo nulla.

SOLUZIONI DEGLI ESERCIZI

Tutte le soluzioni degli esercizi di questo libro si trovano nel suo archivio di codice: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Aprite la cartella del capitolo in cui vi trovate e poi il notebook Jupyter con il nome che menziona le soluzioni degli esercizi.

La funzione f = `id` è leggermente più complessa dei casi `zero` e `one`, perché $y \oplus f(x)$ non può essere semplificata in modo da non dipendere da x. Come potete vedere nella Tabella 8.7, abbiamo bisogno di $U_f|x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle = |x\rangle |y \oplus x\rangle$. Vale a dire, abbiamo bisogno che l'azione dell'oracolo sullo stato di input ($|x\rangle |y\rangle$) lasci stare x e sostituisca a y l'OR esclusivo di x e y.

Un altro modo per considerare questo output è quello di ricordare che $y \oplus 1 = \neg y$, quindi quando $x = 1$, dobbiamo invertire y. Questo è esattamente il modo in cui abbiamo definito l'istruzione di NOT controllato (CNOT) nel Capitolo 6, quindi riconosciamo che quando f è `id`, U_f può essere implementata applicando un CNOT.

Non ci rimane che definire l'oracolo per f = `not`. Proprio come l'oracolo per `id` inverte il qubit di output (target) quando il qubit di input (control) è nello stato $|1\rangle$, lo stesso argomento ci dice che abbiamo bisogno che il nostro oracolo per `not` inverta il secondo qubit quando il qubit di input è $|0\rangle$. Il modo più semplice per farlo è invertire prima il qubit di input con un'istruzione X, poi applicare un'istruzione CNOT e infine annullare la prima inversione con un'altra istruzione X.

SUGGERIMENTO

Per scoprire di più su come potete costruire gli altri oracoli e usare QuTiP per dimostrare che essi fanno quanto desiderato, consultate l'Appendice D.

Per ricapitolare gli oracoli che abbiamo imparato a definire, abbiamo raccolto tutto il lavoro di questo paragrafo nella Tabella 8.7.

Tabella 8.7 Output degli oracoli per ciascuna funzione a un bit f.

Nome funzione	Funzione	Output dell'oracolo

`zero`	$f(x) = 0$	$\|x\rangle\|y \oplus 0\rangle = \|x\rangle\|y\rangle$
`one`	$f(x) = 1$	$\|x\rangle\|y \oplus 1\rangle = \|x\rangle\|\neg y\rangle$
`id`	$f(x) = x$	$\|x\rangle\|y \oplus x\rangle$
`not`	$f(x) = \neg x$	$\|x\rangle\|y \oplus \neg x\rangle$

Il trucco senza calcolo: trasformare le funzioni in oracoli quantistici

Allo stato attuale, può sembrare che sia necessario molto lavoro per progettare *U*f per ciascuna funzione *f*. Fortunatamente, c'è un bel "trucco" che ci permette di costruire un oracolo partendo da un requisito un po' più semplice.

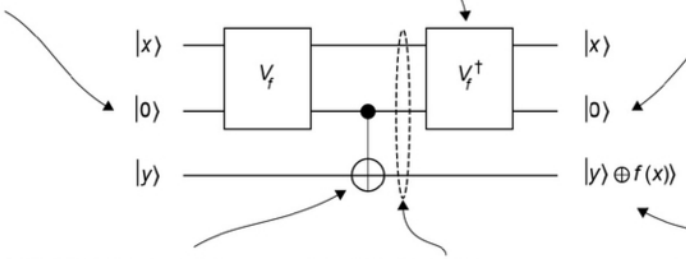
Ricordate che, in precedenza, abbiamo tentato di definire una versione reversibile di *f* restituendo in output (x, $f(x)$) dato in input (x, 0). Allo stesso modo, supponiamo che ci venga data un'operazione quantistica *V*f che trasforma correttamente $|x\rangle|0\rangle$ in $|x\rangle$ $|f(x)\rangle$. Possiamo sempre creare un oracolo *U*f usando un qubit aggiuntivo e richiamando *V*f due volte utilizzando una tecnica chiamata *uncompute trick*, illustrata nella Figura seguente.

L'*uncompute trick*: se aggiungiamo un altro input |0⟩, allora possiamo fare un'operazione che ci dà un modo per rendere reversibile l'operazione da |x⟩|y⟩ a |x⟩|f(x)⟩. Questo funziona per qualsiasi funzione *f*(*x*).

Il primo passo consiste nell'utilizzare un qubit temporaneo insieme al circuito del tentativo 2. La matematica è più semplice se partiamo dallo stato |0⟩.

Possiamo applicare l'inverso del circuito per eliminare lo spreco di memoria, ripulendo il registro temporaneo che abbiamo usato.

Alla fine, il nostro registro temporaneo non "ricorda" nulla di x, quindi non abbiamo alcuno spreco di memoria.



Poiché l'informazione *f*(*x*) è classica, possiamo usare un CNOT per copiare questa informazione nel registro |y⟩.

Lo stato qui sarebbe |x⟩ |*f*(*x*)⟩ |y ⊗ *f*(*x*)⟩. Questo è un problema, perché il secondo qubit spreca informazioni sul nostro sistema (è un qubit in più che ricicliamo dopo l'uso, quindi, come un disco rigido, dovremmo restituirlo vuoto).

A questo punto è semplice verificare che il nostro nuovo circuito è reversibile: in realtà è il suo inverso ed è nella stessa forma degli altri oracoli di questo capitolo.

Uso dell'uncompute trick per trasformare un'operazione che funziona solo quando aggiungiamo un qubit di input $|0\rangle$ in un'operazione che può essere usata come un oracolo.

Anche se questo non aiuta particolarmente nel caso di Deutsch-Jozsa, mostra che il concetto di oracolo è molto generale, poiché spesso è molto più facile iniziare con un'operazione della forma U_f.

NOTA

La costruzione di un oracolo funziona anche per funzioni a più qubit. Come esercizio mentale, se abbiamo una funzione $f(x0, x1) = x0$ AND $x1$, come trasformerà l'oracolo U_f lo stato di input $|x0\ x1\rangle$? Vedremo nei prossimi capitoli come codificare questo oracolo.

Pertanto, abbiamo utilizzato la rappresentazione a oracolo per risolvere un problema: che funzioni come `zero` e `one` non possono essere rappresentate come rotazioni. Detto questo, possiamo continuare a scrivere il resto dell'algoritmo che Viviana usa per sfidare Merlino.

Approfondimento: altri modi per rappresentare le funzioni come oracoli

Questo non è l'unico modo in cui avremmo potuto definire U_f. Merlino potrebbe invertire il segno dell'input di Viviana x quando $f(x)$ è `one`:

$$U_f\,|x\rangle = (-1)^{f(x)}\,|x\rangle\,.$$

Questa risulta essere una rappresentazione più utile in alcuni casi, come negli algoritmi di discesa del gradiente. Questi algoritmi sono comuni nel machine learning e minimizzano le funzioni effettuando la ricerca lungo le direzioni in cui una funzione cambia più velocemente. Per ulteriori informazioni, consultate il paragrafo 4.10 di *Grokking Deep Learning* di Andrew Trask (Manning, 2019).

Scegliere il modo giusto con cui una particolare applicazione può rappresentare le informazioni classiche come le chiamate di subroutine all'interno di un algoritmo quantistico è una parte dell'arte della programmazione quantistica. Per ora useremo la definizione di *oracolo* introdotta in precedenza.

Avendo a disposizione la rappresentazione dell'oracolo di f, i primi passi dell'algoritmo di Deutsch-Jozsa possono essere scritti nello stesso

tipo di pseudocodice che abbiamo usato per scrivere `quicksort` in precedenza.

1. Prepara due qubit di nome `control` e `target` nello stato $|0\rangle \otimes |0\rangle$.
2. Applica le operazioni ai qubit `control` e `target` per preparare il seguente stato: $|+\rangle \otimes |-\rangle$.
3. Applica l'oracolo *Uf* allo stato di input $|+\rangle \otimes |-\rangle$. Ricorda che $Uf|x\rangle|y\rangle = |x\rangle|y \otimes f(x)\rangle$.
4. Misurare il qubit `control` nella base *X*. Se osservi uno `0`, la funzione è costante; in caso contrario, la funzione è bilanciata.

> **SUGGERIMENTO**
>
> Misurare un qubit in base *X* restituisce sempre uno 0 o un 1, proprio come se fosse misurato in base *Z*. Ricordiamo dal Capitolo 3 che se lo stato del qubit è $|+\rangle$, otteniamo sempre 0 quando lo misuriamo in base *X*, mentre otteniamo sempre 1 se il qubit è in $|-\rangle$.

La Figura 8.9 illustra questi passaggi. Vedremo alla fine del capitolo *perché* questo algoritmo funziona, ma per ora iniziamo a programmarlo. Per farlo, dovremo usare il linguaggio Q# fornito dal Quantum Development Kit, dal momento che questo aiuta molto a vedere la struttura di un algoritmo quantistico partendo dal suo codice sorgente.

Simulazione dell'algoritmo di Deutsch-Jozsa in Q#

Nel Capitolo 7, abbiamo provato a passare le operazioni come argomenti nei programmi Q#. Possiamo usare lo stesso approccio per passare le operazioni come input agli oracoli, per aiutare a prevedere come andrà a finire la sfida di Viviana. Per fare ciò, ricordate che possiamo considerare quattro possibili funzioni per questo problema,

ognuna delle quali rappresenta una possibile strategia utilizzabile da Merlino (vedi Tabella 8.8).

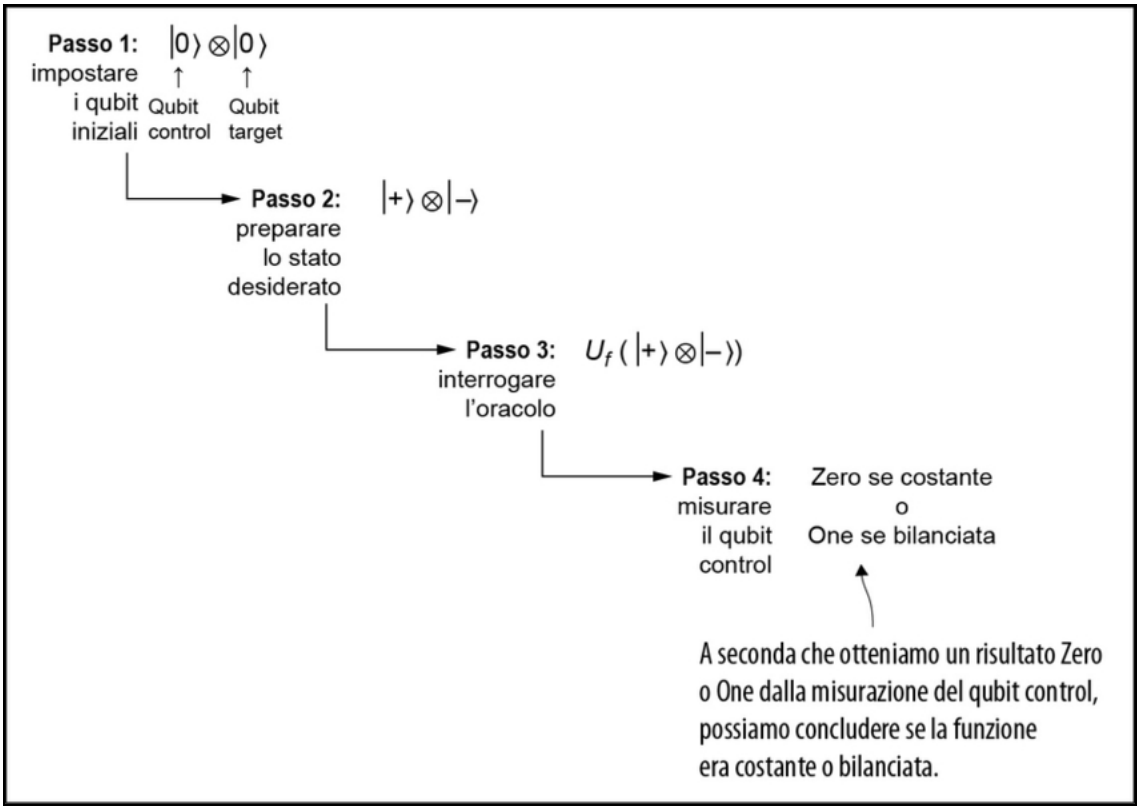


Figura 8.9 Passaggi dell'algoritmo di Deutsch-Jozsa. Iniziamo preparando lo stato |+-⟩, quindi interroghiamo l'oracolo (cioè, poniamo a Merlino la domanda), poi misuriamo il qubit control per sapere se l'oracolo rappresenta una funzione costante o bilanciata.

Tabella 8.8 Rappresentazione delle funzioni a un bit come oracoli a due qubit.

Nome funzione	Funzione	Output dell'oracolo	Operazione Q#
`zero`	$f(x) = 0$	$\|x\rangle \|y \oplus 0\rangle = \|x\rangle\|y\rangle$	`NoOp (control, target);`
`one`	$f(x) = 1$	$\|x\rangle \|y \oplus 1\rangle = \|x\rangle\|\neg y\rangle$	`X(target);`

`id`	$f(x) = x$	$\|x\rangle \|y \oplus x\rangle$	`CNOT(control, target)`
`not`	$f(x) = \neg x$	$\|x\rangle \|y \oplus \neg x\rangle$	`X(control); CNOT(control, target); X (control);`

Se rappresentiamo ogni funzione $f(x)$ con un oracolo (un'operazione quantistica) che mappa $|x\rangle|y\rangle$ in $|x\rangle|y \oplus f(x)\rangle$, allora possiamo identificare ciascuna delle funzioni `zero`, `one`, `id` e `not` della Figura 8.3.

Ciascuno dei quattro oracoli si traduce immediatamente in Q#:

```
namespace DeutschJozsa {
    open Microsoft.Quantum.Intrinsic;
    operation ApplyZeroOracle(control : Qubit, target : Qubit) : Unit {
    }
    operation ApplyOneOracle(control : Qubit, target : Qubit) : Unit {
        X(target);
    }
    operation ApplyIdOracle(control : Qubit, target : Qubit) : Unit {
        CNOT(control, target);
    }
    operation ApplyNotOracle(control : Qubit, target : Qubit) : Unit {
        X(control);
        CNOT(control, target);
        X(control);
    }
}
```

Non potremmo semplicemente guardare il codice sorgente?

In `Oracles.qs`, abbiamo scritto il codice sorgente di ciascuno dei quattro oracoli mono-qubit: `ApplyZeroOracle`, `ApplyOneOracle`, `ApplyIdOracle` e `ApplyNotOracle`. Osservando il codice sorgente, potremmo dire se ognuna di esse è costante o bilanciata, senza doverla richiamare; quindi perché dovremmo occuparci dell'algoritmo di Deutsch-Jozsa? Pensatela dal punto di vista di Viviana: lei non ha il "codice sorgente" impiegato da Merlino per applicare le operazioni ai suoi qubit. Ma anche se lo avesse, le scelte di Merlino sono imperscrutabili, quindi potrebbe non essere in grado di prevedere facilmente ciò che deciderà Merlino, anche avendo il suo codice sorgente.

In pratica, sebbene sia difficile offuscare così tanto un oracolo a due qubit, l'algoritmo di Deutsch-Jozsa dimostra una tecnica che è utile anche più in generale. Per esempio, potremmo avere accesso al codice sorgente di un'operazione, ma estrarre la risposta a una domanda su quell'operazione potrebbe essere un

problema matematicamente o computazionalmente difficile. Per definizione, tutte le funzioni hash crittografiche hanno questa proprietà di base, indipendentemente dal fatto che vengano utilizzate per garantire che un file sia stato scaricato correttamente, per verificare se un'applicazione è stata firmata da uno sviluppatore o per far crescere una blockchain tramite mining.
Nel Capitolo 10, vedremo un esempio che utilizza tecniche come quelle sviluppate nell'algoritmo di Deutsch-Jozsa per porre domande più rapidamente su tali funzioni.

Ora che abbiamo gli oracoli implementati in Q#, possiamo scrivere l'intero algoritmo di Deutsch-Jozsa (e anche la strategia di Viviana per Kingmaker). La Figura 8.10 ricorda i passaggi di Deutsch-Jozsa.

Listato 8.2 Algorithm.qs: operazione Q# per eseguire Deutsch-Jozsa.

```
operation CheckIfOracleIsBalanced(oracle : ((Qubit, Qubit) => Unit)) :
Bool {
    use control = Qubit();
    use target = Qubit(); (1)
    H(control); (2)
    X(target);
    H(target);
    oracle(control, target); (3)
    H(target); (4)
    X(target);
    return MResetX(control) == One; (5)
}
```

(1) Chiede alla macchina target di darci due qubit, `control` e `target`, entrambi nello stato |0⟩.

(2) Prepara lo stato di input |+-⟩ = (|00⟩ - |01⟩ + |10⟩ - |11⟩) / 2 su `control` e `target`, come nel Passo 2 della Figura 8.10.

(3) Richiama l'oracolo fornito come argomento di input. Notate che l'oracolo viene richiamato una sola volta.

(4) Sappiamo che il qubit `target` è ancora |-⟩, quindi possiamo annullare la sequenza di operazioni `X(target); H(target);` per eseguire il reset del qubit `target`.

(5) Misura se il qubit `control` è in |+⟩ o |-⟩, corrispondente ai risultati `Zero` o `One` in base *X*.

Come l'operazione `MResetZ` fornita dalle librerie standard Q#, l'operazione `MResetX` esegue la misurazione in base *X* desiderata e ripristina il qubit misurato allo stato |0⟩. Ora vogliamo assicurarci che la nostra implementazione sia buona, quindi proviamola!

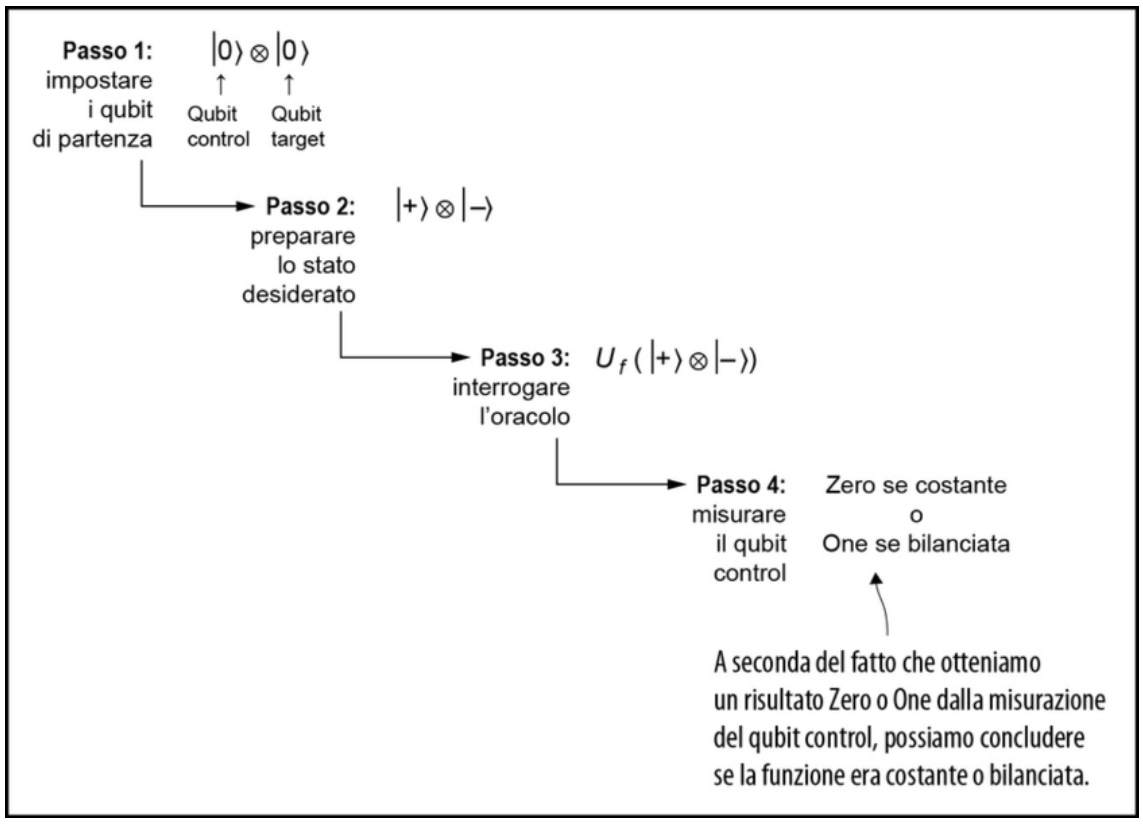


Figura 8.10 I passaggi dell'algoritmo di Deutsch-Jozsa. Iniziamo preparando lo stato |+-⟩, quindi interroghiamo l'oracolo (ovvero poniamo la domanda a Merlino) e quindi misuriamo il qubit control.

Risultati della misurazione in Q#

Abbiamo visto entrambe le operazioni `MResetX` e `MResetZ` di Q#, che misurano e ripristinano un qubit rispettivamente nelle basi *X* e *Z*. Entrambe queste operazioni restituiscono un valore `Result`, che sembra un po' confuso, in un primo momento. Dopotutto, una misurazione in base *X* ci dice se eravamo nello stato |+⟩ o |-⟩, quindi perché Q# usa le etichette `Zero` e `One`?

Convenzioni utilizzate per i risultati delle misurazioni su base X e Z in Q#.

Valore di Result	Base *X*	Base Z.
`Zero`	⟨+|	⟨0|
`One`	⟨-|	⟨1|

Vedremo meglio tutto questo più avanti, ma la "versione breve" è che un valore di tipo `Result` ci dice quante fasi di (-1) vengono applicate a uno stato da istruzioni differenti. Per esempio, $Z|1\rangle = -|1\rangle = (-1)1\ |1\rangle$, mentre $X|-\rangle = (-1)1\ |-\rangle$. Poiché in entrambi i casi eleviamo (-1) alla potenza 1, $|1\rangle$ e $|-\rangle$ sono assegnati al risultato `One`, quando misuriamo rispettivamente nelle basi *Z* e *X*. Allo stesso modo, poiché $Z|0\rangle = (-1)0\ |0\rangle$, assegniamo $|0\rangle$ al risultato `Zero` quando misuriamo in *Z*.

Abbiamo detto che Viviana vorrebbe conoscere il meno possibile sulle cose degli esseri umani. Quindi, domanda a Merlino di fare qualcosa con i suoi qubit una sola volta, dove richiamiamo `oracle(control, target)`. Viviana ottiene solo un bit classico dalla chiamata a `MResetX`, che non è sufficiente per consentirle di distinguere tra la strategia `id` (Merlino sceglie di fare da mentore ad Artù) e la strategia `not` (Merlino sceglie di fare da mentore a Mordred).

Per assicurarsi di conoscere l'unica cosa che le interessa (se la strategia di Merlino è costante o bilanciata) possiamo usare la funzione `Fact` fornita dalle librerie standard Q# per verificare che la nostra implementazione funzioni. `Fact` prende come primi due argomenti due variabili booleane, controlla se sono uguali, e, se non lo sono, emette un messaggio.

SUGGERIMENTO

Più avanti vedremo come usare queste affermazioni per scrivere gli *unit test* per le librerie quantistiche.

La prima cosa che facciamo è passare l'operazione `ApplyZeroOracle` che abbiamo scritto in precedenza come oracolo per la funzione `zero`. Poiché `zero` non è una funzione bilanciata, ci aspettiamo che

`CheckIfOracleIsBalanced(ApplyZeroOracle)` restituisca `false` in output; questa aspettativa può essere verificata utilizzando la funzione `Fact`.

Listato 8.3 Algorithm.qs: operazioni Q# per il controllo di Deutsch-Jozsa.

```
operation RunDeutschJozsaAlgorithm() : Unit {
    Fact(not CheckIfOracleIsBalanced(ApplyZeroOracle),
        "Test failed for zero oracle."); (1)
    Fact(not CheckIfOracleIsBalanced(ApplyOneOracle),
        "Test failed for one oracle.");
    Fact(CheckIfOracleIsBalanced(ApplyIdOracle),
        "Test failed for id oracle."); (2)
    Fact(CheckIfOracleIsBalanced(ApplyNotOracle),
        "Test failed for not oracle.");
    Message("All tests passed!"); (3)
}
```

(1) Esegue l'algoritmo di Deutsch-Jozsa per il caso in cui Merlino utilizza la strategia `zero`.

(2) Fa esattamente la stessa cosa per la strategia `one`, questa volta richiamando `CheckIfOracleIsBalanced(ApplyOneOracle)`.

(3) Se tutte e quattro le asserzioni vengono superate, possiamo essere certi del funzionamento del nostro programma per l'algoritmo di Deutsch-Jozsa, indipendentemente dalla strategia adottata da Merlino.

Se lo eseguiamo utilizzando il comando "magico" `%simulate`, possiamo confermare che, utilizzando l'algoritmo di Deutsch-Jozsa, Viviana può conoscere solo ed esattamente ciò che vuole effettivamente conoscere della strategia di Merlino.

```
In [ ]: %simulate RunDeutschJozsaAlgorithm
All tests passed!
```

Le tecniche degli algoritmi quantistici

Abbiamo ormai compiuto un paio di grandi passi!

- Abbiamo usato delle funzioni reversibili classiche per modellare la strategia di Merlino, in modo da poterla scrivere come un oracolo quantistico.

- Abbiamo usato il linguaggio Q# e il Quantum Development Kit per implementare l'algoritmo di Deutsch-Jozsa e per verificare che possiamo conoscere la strategia di Merlino con una singola chiamata all'oracolo.

A questo punto, è utile riflettere su ciò che abbiamo imparato facendo un tuffo nel lago quantistico di Viviana, poiché le tecniche che abbiamo usato in questo capitolo saranno utili per tutto il resto del libro.

Una questione di calze e scarpe: applicare e annullare le operazioni quantistiche

Il primo schema su cui è utile riflettere è quello che abbiamo potuto notare in `Algorithm.qs`. Osserviamo meglio l'ordine in cui le operazioni vengono applicate al qubit `target`.

Listato 8.4 Istruzioni di Deutsch-Jozsa per target.

```
// ...
X(target);
H(target);
oracle(control, target);
H(target);
X(target);
// ...
```

Un modo per considerare questa sequenza è che le istruzioni `X(target); H(target);` preparano il target nello stato |-⟩, mentre le istruzioni `H(target); X (target);` "de-preparano" |-⟩, riportando il target allo stato |0⟩. Dobbiamo invertire l'ordine a causa di quello che è spesso chiamato principio *shoes and socks*, letteralmente "scarpe e calze". Se vogliamo indossare calze e scarpe, ma è molto meglio se indossiamo prima le calze; se poi vogliamo toglierle, dobbiamo partire dalle scarpe. La Figura 8.11 illustra questa procedura.

Figura 8.11 Non possiamo toglierci le calze prima delle scarpe.

Il linguaggio Q# aiuta a eseguire trasformazioni di tipo “calze e scarpe” nel codice impiegando una sua funzionalità: i *funtori*. I funtori consentono di descrivere facilmente nuove varianti di un’operazione che abbiamo già definito. Facciamo un esempio e introduciamo una nuova operazione, `PrepareTargetQubit`, che incapsula la sequenza

`X(target); H(target);.`

Listato 8.5 La preparazione degli stati dal Listato 8.4.

```
operation PrepareTargetQubit(target : Qubit) : Unit is Adj { (1)
    X(target);
    H(target);
}
```

(1) Scrivendo `is Adj` nella signature, diciamo al compilatore Q# di calcolare automaticamente l’operazione inversa, cioè l’*aggiunto*, di questa operazione.

Possiamo richiamare l'operazione inversa generata dal compilatore utilizzando `Adjoint`, uno dei due funtori forniti da Q# (vedremo l'altro nel Capitolo 9).

Listato 8.6 Utilizzo della parola riservata Adjoint per applicare le istruzioni.

```
PrepareTargetQubit(target);
oracle(control, target);
Adjoint PrepareTargetQubit(target); (1)
```

(1) `Adjoint PrepareTargetQubit` applica il funtore `Adjoint` a `PrepareTargetQubit`, restituendo un'operazione che "annulla" `PrepareTargetQubit`. Seguendo il principio "calze e scarpe", questa nuova operazione funziona richiamando prima `Adjoint H(target);` e poi `Adjoint X(target);`.

Operazioni self-adjoint

In questo caso, `X` e `Adjoint X` sono la stessa operazione, poiché invertendo un bit e poi invertendolo di nuovo torniamo sempre al punto di partenza. In altre parole, `Adjoint X` annulla `X`. Allo stesso modo, `Adjoint H` è uguale a `H`, quindi il frammento precedente ci dà la sequenza `H(target); X(target);`. Diciamo che le istruzioni `X` e `H` sono self-adjoint.

Tuttavia, non tutte le operazioni sono le loro adjoint! Per esempio, `Adjoint Rz(theta, _)` è la stessa operazione di `Rz(-theta, _)`.

In termini più pratici, il funtore `Adjoint` sull'operazione *U* è lo stesso dell'operazione che inverte o annulla gli effetti di *U*. Il nome *aggiunta* si riferisce alla coniugata trasposta *U*+ di una matrice unitaria *U*. In parole, *U*+ è chiamata *aggiunta* di *U*. La parola riservata `Adjoint` di Q# garantisce che se un'operazione *U* è descritta dalla matrice unitaria *U*, allora se esiste `Adjoint U`, è descritta da *U*+.

Il modello di esecuzione di un'istruzione è così comunemente utilizzato che le librerie standard Q# forniscono l'operazione `ApplyWith`

per esprimere questo modello di esecuzione e successivo annullamento di un'operazione.

> **NOTA**
>
> L'operazione `ApplyWith` è fornita dal namespace `Microsoft.Quantum.Canon` nella libreria standard Q#. Come avviene per molti altri linguaggi, la libreria standard Q# fornisce molti strumenti di base necessari per scrivere programmi Q#. Nel resto del libro, troverete molti modi in cui la libreria standard Q# può essere utile per semplificarvi la vita come sviluppatori quantistici.

Utilizzando `ApplyWith` e l'applicazione parziale, possiamo riscrivere l'operazione `Check IfOracleIsBalanced` in modo compatto.

Listato 8.7 Utilità di ApplyWith e dell'applicazione parziale per il principio "calze e scarpe".

```
H(control);
ApplyWith(PrepareTargetQubit, oracle(control, _), target);
set result = MResetX(control);
```

L'operazione `ApplyWith` di questo esempio applica automaticamente l'adjoint di `PrepareTargetQubit` dopo aver eseguito `oracle(control, _)`. Notate che _ viene utilizzato per applicare parzialmente l'oracolo al qubit `control`.

Espandiamo il Listato 8.7 passo dopo passo per vedere come funziona. La chiamata ad `ApplyWith` viene applicata al primo argomento, poi al secondo argomento, poi applica l'aggiunta del primo argomento al qubit fornito nell'ultimo argomento.

Listato 8.8 Espansione di ApplyWith nel Listato 8.7.

```
H(control);
PrepareTargetQubit(target);
(oracle(control, _))(target);
Adjoint PrepareTargetQubit(target);
set result = MResetX(control);
```

L'applicazione parziale sulla terza riga può essere riscritta sostituendo `target` a _.

Listato 8.9 Risoluzione dell'applicazione parziale nel Listato 8.8.

```
H(control);
PrepareTargetQubit(target);
oracle(control, target);
Adjoint PrepareTargetQubit(target);
set result = MResetX(control);
```

L'uso di operazioni come `ApplyWith` aiuta a riutilizzare schemi molto comuni nella programmazione quantistica, in particolare per essere sicuri di non dimenticare di prendere un `Adjoint` in un grosso programma quantistico.

Q# fornisce anche un altro modo per rappresentare il modello "calze e scarpe" utilizzando blocchi di dichiarazioni invece di passare le operazioni. Per esempio, potremmo scrivere il Listato 8.7 utilizzando le parole riservate `within` e `apply`, come mostrato di seguito.

Listato 8.10 Uso di within e apply per il principio "calze e scarpe".

```
H(control);
within {
    PrepareTargetQubit(target);
} apply {
    oracle(control, target);
}
set result = MResetX(control);
```

Entrambe le forme possono essere utili in diversi contesti, quindi sentitevi liberi di usare quello che vi sembra funzionare meglio.

Usare le istruzioni di Hadamard per invertire control e target

Possiamo usare ragionamenti alla "calze e scarpe" del paragrafo precedente per cambiare quali qubit svolgono i ruoli `control` e `target` in istruzioni come `CNOT`. Per capire come funziona, è importante tenere presente che le istruzioni quantistiche trasformano l'intero stato dei registri su cui agiscono. In casi come l'algoritmo di Deutsch-Jozsa, il

qubit `control` può essere influenzato dall'applicazione di porte logiche ai qubit `control` e `target`, insieme, non solo al qubit `target`. Questo è un esempio di un modello più generale: `control` e `target` di un'operazione `CNOT` si scambiano i ruoli quando applichiamo un'istruzione `CNOT` nella base *X* invece che nella base *Z* (computazionale).

Per vederlo in azione, esaminiamo l'operatore unitario (l'analogo quantistico alle tabelle di verità classiche) per quello che succede se usiamo istruzioni `H` per trasformare in base *X*, poi applichiamo un'istruzione `CNOT` e quindi usiamo più istruzioni `H` per tornare alla base *Z*.

Listato 8.11 Verifica che H inverta control e target in CNOT.

```
>>> import qutip as qt
>>> from qutip.qip.operations import hadamard_transform
>>> H = hadamard_transform()
>>> HH = qt.tensor(H, H)
>>> HH (1)
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper,
isherm = True
Qobj data = (2)
[[ 0.5  0.5  0.5  0.5]
 [ 0.5 -0.5  0.5 -0.5]
 [ 0.5  0.5 -0.5 -0.5]
 [ 0.5 -0.5 -0.5  0.5]]
>>> HH * qt.cnot(2, 0, 1) * HH (3)
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper,
isherm = True
Qobj data = (4)
[[1. 0. 0. 0.]
 [0. 0. 0. 1.]
 [0. 0. 1. 0.]
 [0. 1. 0. 0.]]
>>> qt.cnot(2, 1, 0) (5)
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper,
isherm = True
Qobj data =
[[1. 0. 0. 0.]
 [0. 0. 0. 1.] (6)
 [0. 0. 1. 0.]
 [0. 1. 0. 0.]]
```

(1) È utile definire una scorciatoia per l'operatore unitario $H \otimes H$ che simula la sequenza di istruzioni `H(control); H(target);`.

(2) Osservando l'operatore unitario $H \otimes H$, vediamo che $|00\rangle$ viene trasformato in $(|00\rangle + |01\rangle + |10\rangle + |11\rangle) / 2$, una sovrapposizione uniforme su tutti e quattro gli stati computazionali di base.
(3) Fornisce l'operatore unitario che rappresenta `H(control); H(target); CNOT(control, target); H(control); H(target);`. Possiamo considerare questa sequenza di istruzioni come l'applicazione di un `CNOT` nella base X invece che nella base Z.
(4) L'operatore unitario per questa sequenza somiglia un po' a `CNOT`, ma con alcune righe invertite. Che cosa è successo?
(5) Per cercare di capire che cosa ha fatto all'istruzione `CNOT` l'applicazione di istruzioni `H` a ciascun qubit, vediamo l'operatore unitario per `CNOT(target, control)`.
(6) Invertire il ruolo dei qubit `control` e `target` in una chiamata all'istruzione `CNOT` ci fornisce esattamente lo stesso operatore unitario dell'uso delle istruzioni `H` per applicare un'istruzione `CNOT` nella base X.

La Figura 8.12 fornisce una rappresentazione visuale del codice Python che abbiamo appena eseguito.

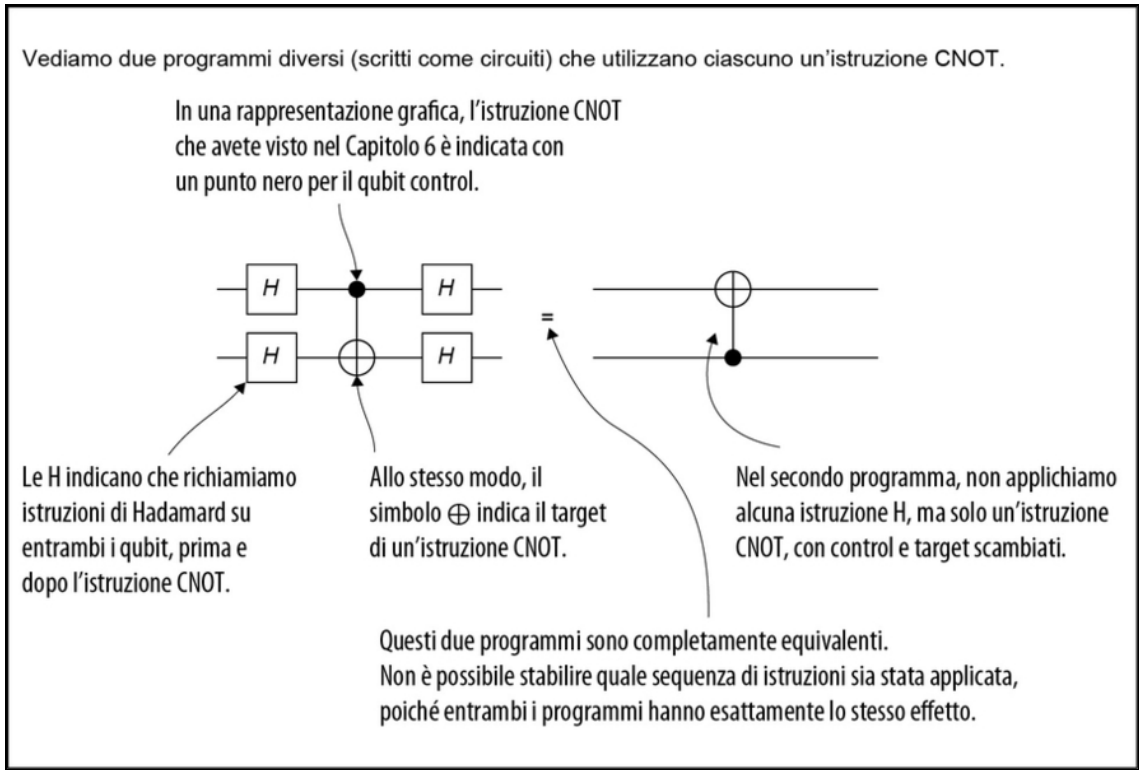


Figura 8.12 Scambio dei qubit control e target di un'istruzione CNOT utilizzando istruzioni di Hadamard. Applicando l'Hadamard su ogni qubit, sia prima sia dopo il CNOT, possiamo invertire i ruoli dei qubit control e target.

Dal calcolo precedente, possiamo concludere che `CNOT(target, control)` fa esattamente la stessa cosa di `H(control); H(target); CNOT(control, target); H(control); H(target);`. Allo stesso modo in cui `H` inverte il ruolo delle basi *X* e *Z*, le istruzioni `H` possono scambiare i qubit `control` e `target`.

Phase kickback: la chiave del nostro successo

Ora che conosciamo queste tecniche, siamo attrezzati per esplorare ciò che rende così utile l'algoritmo di Deutsch-Jozsa: una tecnica di

programmazione quantistica chiamata *phase kickback*. Questa tecnica ci consente di scrivere l'operazione `CheckIfOracleIsBalanced` in modo che funzioni per più oracoli differenti, rivelando solo quello che vogliamo sapere (se Merlino intende agire come un buon mentore).

Per vedere come l'algoritmo di Deutsch-Jozsa utilizza il phase kickback per funzionare *in generale*, torniamo ai nostri tre modi di pensare e usiamo la matematica per prevedere che cosa succede quando richiamiamo un oracolo. Ricordiamo che abbiamo definito l'oracolo *Uf* che abbiamo costruito da ciascuna funzione classica *f* tale che, per tutti i bit classici *x* e *y*, $Uf|xy\rangle = |x\rangle\,|f(x) \oplus y\rangle$.

SUGGERIMENTO

Qui usiamo *x* e *y* per rappresentare i bit classici che sono le etichette degli stati di due qubit. Questo è un altro esempio di utilizzo della base computazionale per ragionare su come si comportano i programmi quantistici.

Cominciamo nello stesso modo in cui abbiamo fatto nei nostri programmi QuTiP espandendo lo stato di input e $|+-\rangle = |+\rangle \otimes |-\rangle$ nella base computazionale. Partendo dall'espansione dello stato del qubit `control`, abbiamo che $|+-\rangle = |+\rangle \otimes |-\rangle = (|0\rangle + |1\rangle) / \sqrt{2} \otimes |-\rangle = (|0-\rangle + |1-\rangle) / \sqrt{2}$. Come prima, possiamo controllarne la correttezza impiegando QuTiP.

Listato 8.12 Usare QuTiP per controllare $(|0-\rangle + |1-\rangle) / \sqrt{2} = |+-\rangle$.

```
>>> import qutip as qt
>>> from qutip.qip.operations import hadamard_transform
>>> from numpy import sqrt
>>> H = hadamard_transform()
>>> ket_0 = qt.basis(2, 0) (1)
>>> ket_1 = qt.basis(2, 1)
>>> ket_plus = H * ket_0
>>> ket_minus = H * ket_1
>>> qt.tensor(ket_plus, ket_minus)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.5]
 [-0.5]
 [ 0.5]
 [-0.5]]
```

```
>>> (
...     qt.tensor(ket_0, ket_minus) +
...     qt.tensor(ket_1, ket_minus)
... ) / sqrt(2)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.5] (2)
 [-0.5]
 [ 0.5]
 [-0.5]]
```

(1) Inizia con un'utile notazione abbreviata.

(2) Entrambi i vettori sono gli stessi, il che ci dice che (|0-⟩ + |1-⟩) / √2 è un altro modo di scrivere |+-⟩.

Successivamente, come abbiamo visto nel Capitolo 2, possiamo usare la linearità per prevedere come *Uf* trasforma questo stato di input.

La matrice rivisitata

Abbiamo utilizzato implicitamente la linearità in precedenza in questo paragrafo, quando abbiamo utilizzato le matrici per modellare il funzionamento dell'algoritmo di Deutsch-Jozsa. Come descritto nel Capitolo 2, le matrici sono un modo per scrivere funzioni lineari.

Poiché *Uf* è una matrice unitaria, sappiamo che per *ogni* stato |ψ⟩ e |φ⟩ e per *ogni* numero α e β, $Uf(\alpha|\psi\rangle + \beta|\varphi\rangle) = \alpha Uf|\psi\rangle + \beta\, Uf|\varphi\rangle$]. Utilizzando questa proprietà con la base computazionale, abbiamo che, allo stesso modo |+-⟩ e (|0-⟩ + |1-⟩) / √2 sono lo stesso stato e anche *Uf*|+-⟩ e *Uf*(|0-⟩ + |1-⟩) / √2 sono lo stesso stato.

SUGGERIMENTO

Usando la nostra scorciatoia per gli stati multi-qubit, |+-⟩ = |+⟩ ⊗ |-⟩, |0-⟩ = |0⟩ ⊗ |-⟩ e |1-⟩ = |1⟩ ⊗ |-⟩.

La Figura 8.13 fornisce una rappresentazione visuale della linearità.

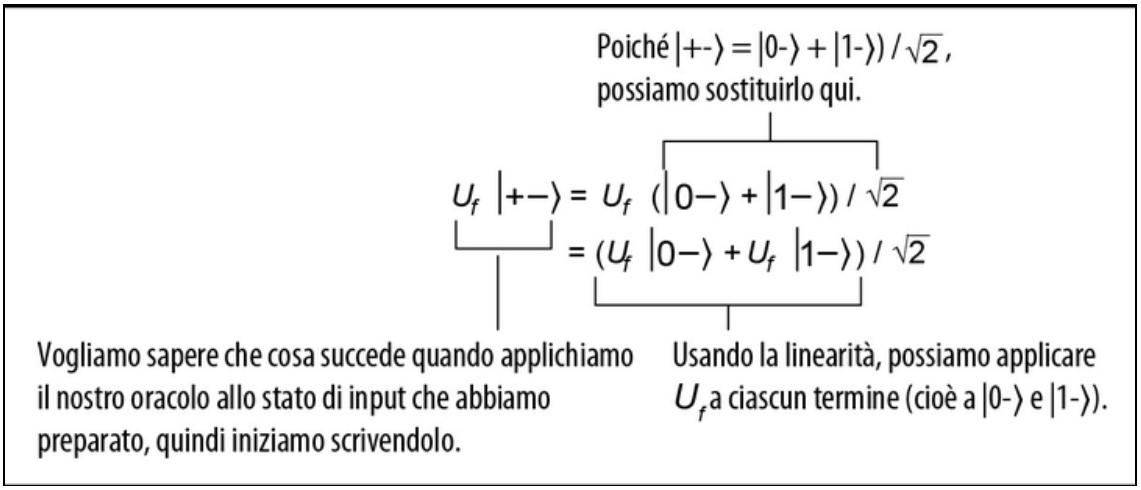


Figura 8.13 Applicazione della linearità per comprendere come il nostro oracolo trasforma lo stato di input.

Così come è scritto, non è immediatamente evidente quale vantaggio abbiamo ottenuto mediante l'applicazione di *Uf* a |0-⟩ e a |1-⟩. Vediamo come l'operazione `oracle` si applica al primo termine fattorizzando il (primo) qubit `control` per considerare l'effetto sul qubit `target`.

Nel farlo, useremo di nuovo la linearità per capire come funziona *Uf* passando uno stato alla volta al nostro oracolo. Come abbiamo appreso nel Capitolo 2, la linearità è uno strumento molto potente, che ci consente di scomporre anche algoritmi quantistici piuttosto complicati in frammenti che possiamo comprendere e analizzare più facilmente. In questo caso, possiamo capire come opera *Uf* su |0-⟩ utilizzando la linearità (ovvero, suddividendo |0-⟩ in una sovrapposizione tra |00⟩ e |01⟩).

Iniziamo usando che …

$$U_f \left|0-\right\rangle = U_f(|00\rangle - |01\rangle)/\sqrt{2}$$

Possiamo applicare *Uf* a ciascun termine, per linearità.

$$= (U_f \left|00\right\rangle - U_f \left|01\right\rangle)/\sqrt{2}$$

La definizione di oracolo ci dice come *Uf* trasforma gli stati nella base computazionale.

$$= \left(\left|0(0 \oplus f(0))\right\rangle - \left|0(1 \oplus f(0))\right\rangle\right)/\sqrt{2}$$

Applicando lo XOR fra qualcosa e 0 non facciamo nulla.

$$= (\left|0f(0)\right\rangle - \left|0(1 \oplus f(0))\right\rangle)/\sqrt{2}$$

Applicando lo XOR fra qualcosa e 1 è come se applicassimo un NOT.

$$= (\left|0f(0)\right\rangle - \left|0(\neg f(0))\right\rangle)/\sqrt{2}$$

Poiché il qubit `control` è |0-⟩ in entrambi i termini, possiamo estrarlo.

$$= \left|0\right\rangle \otimes \left(\left|f(0)\right\rangle - \left|\neg f(0)\right\rangle\right)/\sqrt{2}$$

Per esempio, se stiamo considerandola funzione `zero`, allora *f*(0) = 0. Pertanto, |*f*(0)⟩ = |0⟩ e |¬*f*(0)⟩ = |1⟩, così *Uf* |0-⟩ = |0-⟩.

D'altra parte, se *f*(0) = 1, allora *Uf*|0-⟩ = |0⟩ ⊕ (|1⟩ - |0⟩) / √2 = -|0-⟩. Cioè, *Uf* inverte il segno di |0-⟩.

SUGGERIMENTO

(|1⟩ - |0⟩) / √2 può anche essere scritto come -|-⟩ o come *X*|-⟩.

Notate quindi che *Uf* ruota il qubit `target` di *X* o meno a seconda del valore di *f*(0):

$$U_f \left|0-\right\rangle = (-1)^{f(0)} \left|0-\right\rangle/\sqrt{2}$$

Possiamo usare la stessa tecnica che abbiamo usato prima per capire che cosa fa *Uf* se invece il qubit `control` è nello stato $|1\rangle$. In questo modo, otteniamo una fase di $(-1)^{f}(1)$ invece di $(-1)^{f}(0)$, in modo che $Uf|1\text{-}\rangle = (-1)^{f}(1)\,|1\text{-}\rangle$.

Usando di nuovo la linearità per combinare i termini per i due stati del qubit `control`, ora sappiamo come *Uf* trasforma lo stato di entrambi i qubit quando il qubit `control` è nello stato $|+\rangle$:

$$U_f\,|+-\rangle = \frac{1}{\sqrt{2}}((-1)^{f(0)}\,|0-\rangle + (-1)^{f(1)}\,|1-\rangle).$$

L'ultimo passo è notare che, come abbiamo visto nel Capitolo 4, non possiamo osservare le *fasi globali*. Pertanto, possiamo escludere $(-1)^{f}(0)$ per esprimere lo stato di output in termini di $f(0) \oplus f(1)$, la domanda cui eravamo inizialmente interessati, come illustrato nella Figura 8.14.

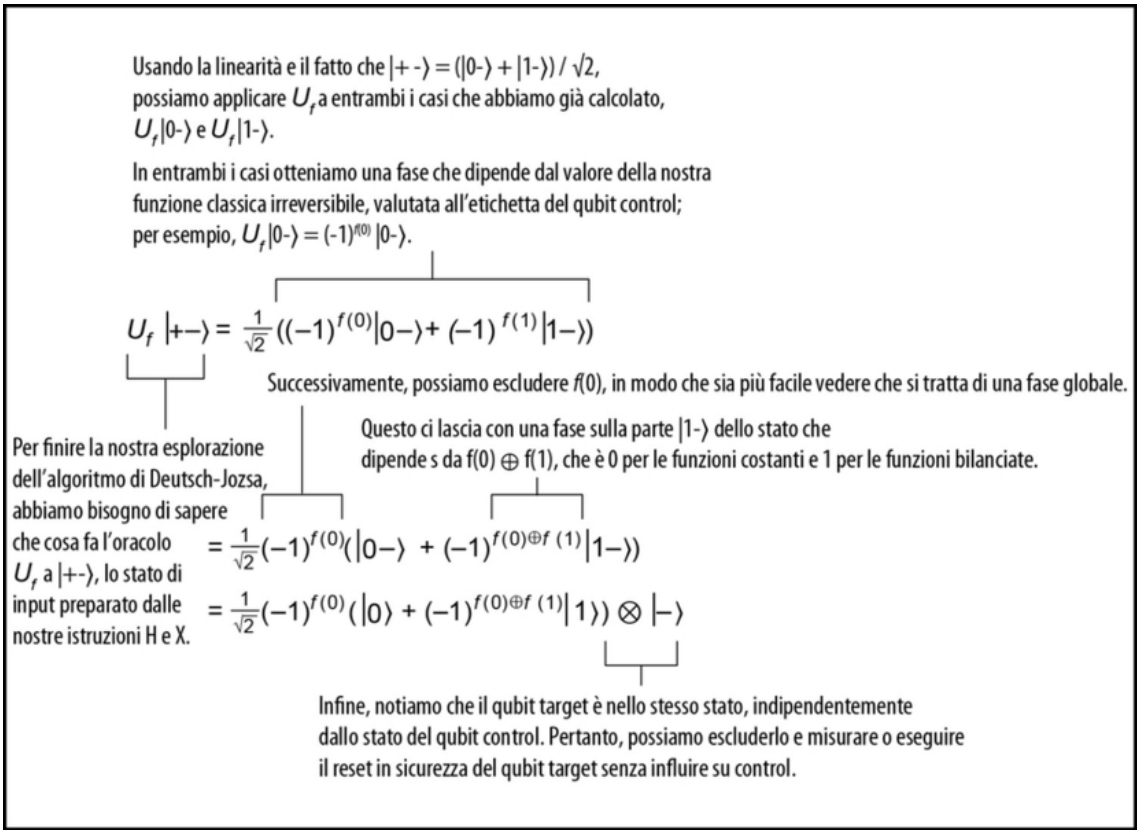


Figura 8.14 Elaborazione degli ultimi due passaggi dell'algoritmo di Deutsch-Jozsa. Scrivendo l'azione dell'oracolo sullo stato |+-⟩, possiamo vedere come il fatto di misurare il qubit control ci dice se l'oracolo rappresenta una funzione costante o bilanciata.

Se $f(0) \oplus f(1) = 0$ (costante), lo stato dell'output è |+-⟩; ma se $f(0) \oplus f(1) = 1$ (bilanciata), lo stato dell'output è |-⟩. Con una chiamata a *Uf*, scopriamo se *f* è costante o bilanciata anche se non sappiamo che cosa sia $f(x)$ per un determinato input x.

Un modo per considerare cosa succede quando applichiamo *Uf* con il qubit di input nello stato |+⟩ è che lo stato del qubit di input rappresenta la domanda che stiamo ponendo a proposito di *f*. Se domandiamo |0⟩, otteniamo la risposta $f(0)$, mentre se domandiamo |1⟩, otteniamo la risposta $f(1)$. La domanda |+⟩ quindi ci parla di $f(0) \oplus f(1)$ senza dirci nulla di $f(0)$ o di $f(1)$.

Quando poniamo domande in sovrapposizione come questa, tuttavia, i ruoli di "input" e "output" non sono così immediatamente chiari come sono in ambito classico. In particolare, entrambi gli input $|0\rangle$ e $|1\rangle$ causano l'inversione del qubit di output, mentre l'input $|+\rangle$ provoca l'inversione del qubit di input, ammesso che iniziamo il qubit di output nello stato $|-\rangle$. In generale, le operazioni a due qubit come *Uf* trasformano l'intero spazio dei qubit su cui agiscono: la nostra suddivisione fra input e output è solo il modo in cui interpretiamo l'azione di *Uf*.

Il fatto che lo stato del qubit di input cambi in base alle trasformazioni definite nel qubit di output è un esempio di *phase kickback*, l'effetto quantistico utilizzato dall'algoritmo di Deutsch-Jozsa. Nei prossimi due capitoli, utilizzeremo il *phase kickback* per esplorare nuovi algoritmi come quelli utilizzati nelle simulazioni di sensori quantistici e di chimica quantistica.

Approfondimento: estendere Deutsch-Jozsa

Sebbene qui abbiamo considerato solo funzioni con input di un bit, l'algoritmo di Deutsch-Jozsa necessita di una sola domanda per input/output di qualsiasi dimensione della nostra funzione.

Per codificare una funzione a due qubit $f(x0, x1)$, possiamo introdurre un oracolo di tre qubit $U\text{f}|x0x1\text{y}\rangle = |x0x1\rangle \otimes |f(x0, x1) \oplus y\rangle$. Per esempio, considerate $f(x0, x1) = x0 \oplus x1$. Questa funzione è bilanciata, poiché $f(0, 0) = f(1, 1) = 0$, ma $f(0, 1) = f(1, 0) = 1$. Quando applichiamo Uf allo stato a tre qubit $|{+}{+}{-}\rangle = (|00\rangle + |01\rangle + |10\rangle + |11\rangle) \otimes |-\rangle$, otteniamo $(|00\rangle - |01\rangle - |10\rangle + |11\rangle) \otimes |-\rangle = |{-}{-}{-}\rangle$. Usando una misurazione in base *X*, possiamo distinguerla da una funzione costante come $f(x0, x1) = 0$, che ci darà un output $|-\rangle$.

Finché ci viene promesso che *f* è costante o bilanciata, lo stesso modello vale indipendentemente dal numero di bit che *f* accetta in input: possiamo apprendere un singolo bit di dati su come si comporta *f* con una singola chiamata a *U*f. Parliamo di O(1)! Se non avete familiarità con la notazione Big O, consultate *Grokking Algorithms* di Aditya Bhargava (Manning, 2016).

Nel prossimo capitolo, estenderemo le competenze che abbiamo imparato osservando a come l'*algoritmo di stima della fase* permette di ottenere tecnologie come i sensori quantistici.

Riepilogo

- Gli algoritmi quantistici sono sequenze di passaggi che possiamo seguire per risolvere un problema utilizzando computer quantistici. Possiamo implementare algoritmi quantistici scrivendo programmi quantistici in Q#.
- Quello di Deutsch-Jozsa è un esempio di algoritmo quantistico che ci consente di risolvere un problema computazionale con meno risorse rispetto a qualsiasi algoritmo classico possibile.
- Se vogliamo incorporare funzioni classiche in un algoritmo o programma quantistico, dobbiamo farlo in modo *reversibile*. Possiamo costruire tipi speciali di operazioni quantistiche chiamate oracoli, che ci consentono di rappresentare le funzioni classiche applicate ai dati quantistici.
- L'algoritmo di Deutsch-Jozsa ci permette di verificare se entrambi gli output di un oracolo a un bit sono uguali o differenti usando una sola chiamata a quell'oracolo; non veniamo a conoscere alcun output specifico, ma apprendiamo direttamente la proprietà globale che ci interessa.
- L'algoritmo di Deutsch-Jozsa mostra l'uso dei modelli "calze e scarpe" che si presentano comunemente anche in altri algoritmi quantistici. Spesso avremo bisogno di applicare un'operazione esterna, poi di applicare un'operazione interna e quindi annullare (prendere l'aggiunta de) l'operazione esterna.
- La tecnica del *phase kickback* ci permette di associare la fase applicata da un'operazione quantistica con un qubit `control` invece

del qubit `target`. Lo rivedremo negli algoritmi che scopriremo più avanti.

Sensori quantistici: non solo una fase

Nel capitolo precedente, abbiamo implementato il nostro primo algoritmo quantistico, Deutsch-Jozsa, in Q#. Aiutando Viviana e Merlino a giocare a Kingmaker, abbiamo visto come le tecniche di programmazione quantistica come *phase kickback* possano offrirci vantaggi nella risoluzione dei problemi. In questo capitolo daremo uno sguardo agli *algoritmi di stima della fase* che possiamo utilizzare nei nostri programmi quantistici per risolvere diversi tipi di problemi. Di nuovo, torneremo a Camelot; questa volta il gioco sarà tra Lancillotto e Dagonet.

Stima della fase: utilizzo di proprietà utili dei qubit per le misurazioni

In tutto il libro, abbiamo visto che i giochi possono essere un modo utile per apprendere i concetti del quantum computing. Per esempio, nel capitolo precedente, il gioco di Viviana con Merlino ci ha permesso di esplorare il nostro primo algoritmo quantistico: l'algoritmo di Deutsch-Jozsa. In questo capitolo, useremo un altro gioco per scoprire come conoscere le fasi degli stati quantistici utilizzando il *phase kickback*, la tecnica di sviluppo quantistico utilizzata dal Deutsch-Jozsa e da molti altri algoritmi quantistici.

Per il gioco di questo capitolo, torniamo indietro e vediamo che cosa ha combinato Lancillotto. Mentre Viviana e Merlino stanno decidendo il destino del re, troviamo Lancillotto e il buffone di corte, Dagonet, che giocano agli indovinelli. Da quando hanno più tempo per giocare, Dagonet si annoia e vuole "prendere in prestito" alcuni degli strumenti quantistici di Viviana per rendere il loro gioco un po' più interessante.

Applicazione parziale

Per il nuovo gioco di Dagonet, invece di far indovinare a Lancillotto un numero, gli chiede di indovinare che cosa fa un'operazione quantistica a un singolo qubit, consentendo a Lancillotto di chiamarlo con input differenti. Dal momento che tutte le operazioni mono-qubit sono rotazioni, questo gioco funziona. Dagonet sceglie un angolo di rotazione attorno a un determinato asse e Lancillotto inserisce un numero per l'operazione di Dagonet, che cambia la scala della rotazione applicata da Dagonet. L'asse scelto da Dagonet non ha molta importanza, poiché il gioco consiste nell'indovinare l'angolo di rotazione. Per comodità qui, le rotazioni di Dagonet sono sempre attorno all'asse Z. Infine, Lancillotto può misurare il qubit e utilizzare la sua misurazione per indovinare l'angolo di rotazione originale di Dagonet. La Figura 9.1 mostra un diagramma di flusso dei seguenti passaggi.

1. Dagonet sceglie un angolo segreto per un'operazione di rotazione mono-qubit.
2. Dagonet prepara un'operazione per Lancillotto che nasconde l'angolo segreto e consente a Lancillotto di specificare come input aggiuntivo un numero (una "scala") di cui verrà moltiplicato l'angolo segreto per dare l'angolo di rotazione totale dell'operazione.

3. La migliore strategia di Lancillotto per il gioco consiste nel selezionare molti valori di scala e stimare la probabilità di misurare `One` per ogni valore. Per fare ciò, deve svolgere le seguenti operazioni molte volte per ciascuno dei valori di scala.
   - Preparare lo stato $|+\rangle$ e inserire il valore della scala nella rotazione di Dagonet. Usare lo stato $|+\rangle$ perché sa che Dagonet sta ruotando attorno all'asse Z, e per questo stato le rotazioni produrranno un cambiamento di fase locale che lui può misurare.
   - Dopo aver preparato ogni stato $|+\rangle$, Lancillotto può ruotarlo con l'operazione segreta, misurare il qubit e registrare la misurazione.
4. Lancillotto ora dispone dei dati relativi al suo fattore di scala e alla probabilità di aver misurato un `One` per quel fattore di scala. Può valutare questi dati e ottenere l'angolo di Dagonet in base ai parametri adattati (in fin dei conti è il primo dei cavalieri). Possiamo usare Python per aiutarci.

Notate che questo *è* un gioco, poiché Lancillotto non ha alcun modo di misurare questa rotazione direttamente con una sola misurazione. Se potesse farlo, violerebbe il teorema di non-clonazione e trascenderebbe le leggi della fisica. In qualità di Cavaliere della Tavola Rotonda, Lancillotto è vincolato non solo dal dovere e dall'onore, ma anche dalle leggi della fisica, quindi deve giocare al gioco di Dagonet secondo le regole.

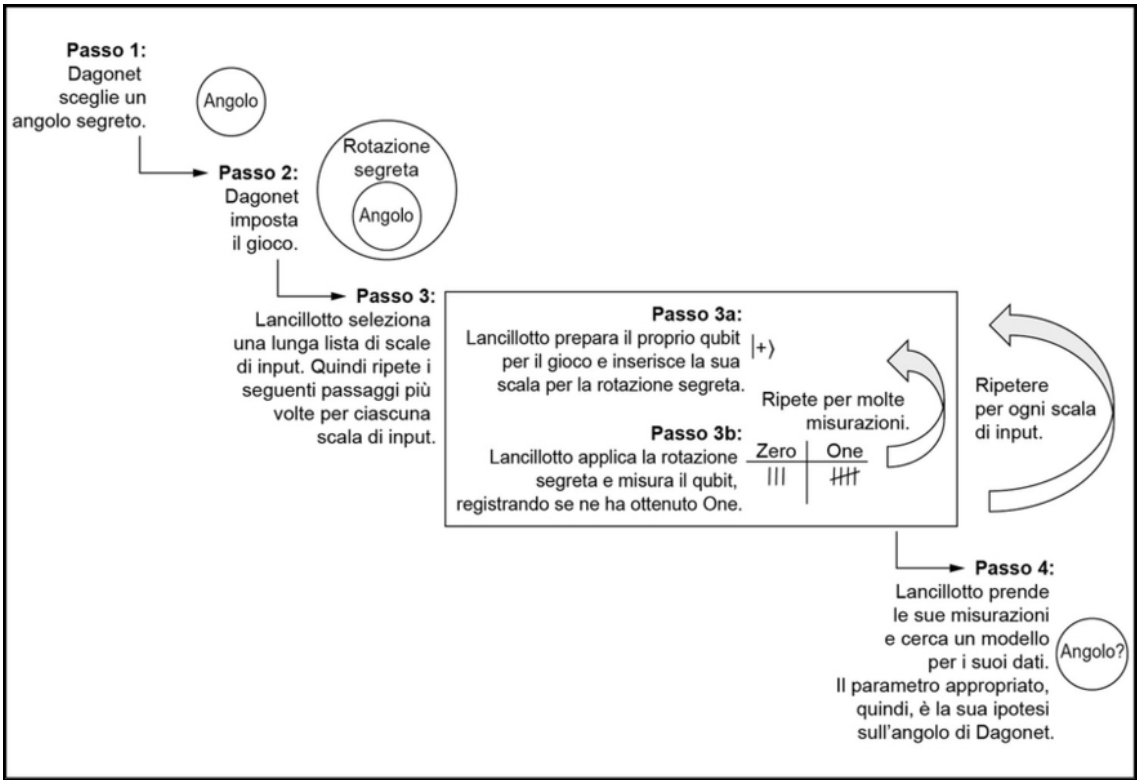


Figura 9.1 Passi nel gioco di Dagonet e Lancillotto. Dagonet applica un angolo di rotazione segreto in un'operazione e Lancillotto deve indovinarlo.

Approfondimento: apprendimento dell'asse con l'apprendimento hamiltoniano

In questo capitolo, ci concentriamo sul caso in cui l'asse di rotazione di Dagonet è noto, mentre dobbiamo scoprire l'angolo. Questo caso corrisponde a un problema comune in fisica, in cui abbiamo il compito di apprendere la *precessione di Larmor* di un qubit in un campo magnetico. La conoscenza delle precessioni di Larmor non è utile solo per costruire qubit; consente anche di rilevare campi magnetici molto piccoli e di costruire sensori molto precisi.

Più in generale, però, possiamo imparare molto di più di un semplice angolo di rotazione. Il caso in cui l'asse è noto è un esempio di un tipo generale di problema chiamato *apprendimento hamiltoniano*, un ricco ambito di ricerca nel quantum computing. Nell'apprendimento hamiltoniano, ricostruiamo un modello fisico per un qubit o un registro di qubit utilizzando un gioco molto simile a quello esplorato in questo capitolo.

Passiamo alla prototipazione di questo gioco in Q#. Sarà utile avere accesso a diverse parti delle librerie standard Q#, quindi possiamo iniziare aggiungendo le seguenti istruzioni `open` all'inizio del nostro file Q#, `operations.qs`.

Listato 9.1 operations.qs: apertura dei namespace Q# per il gioco.

```
namespace PhaseEstimation { (1)
open Microsoft.Quantum.Intrinsic; (2)
open Microsoft.Quantum.Convert as Convert; (3)
open Microsoft.Quantum.Measurement as Meas; (4)
open Microsoft.Quantum.Arrays as Arrays; (5)

// ...
```

(1) Tutte le istruzioni `open` di un file Q# vengono subito dopo la dichiarazione `namespace`.

(2) Come in precedenza, aprendo `Microsoft.Quantum.Intrinsic` avremo accesso a tutte le istruzioni fondamentali (`R1`, `Rz`, `X` e così via) che possiamo inviare a un dispositivo quantistico.

(3) Possiamo anche dare un alias ai namespace quando li apriamo, così come possiamo dare alias a package e moduli Python quando li importiamo. Qui abbreviamo `Microsoft.Quantum.Convert` in modo che in seguito possiamo utilizzare le funzioni di conversione di tipo in tale namespace usando solo il prefisso `Convert`.

(4) Possiamo rendere disponibile l'operazione `MResetZ` dei capitoli precedenti come `Meas.MResetZ` per documentare l'origine dell'operazione.

(5) L'ultimo spazio dei nomi che dobbiamo aprire in questo capitolo è `Microsoft.Quantum.Arrays`, che fornisce funzioni e operazioni utili per lavorare con gli array.

Il Listato 9.2 mostra un esempio dell'operazione quantistica di cui Dagonet ha bisogno per implementare la rotazione con cui giocheranno lui e Lancillotto. Come in altre rotazioni, la nostra nuova operazione di

rotazione restituisce `Unit` (il tipo della tupla vuota `()`), indicando che non vi è alcun output significativo dall'operazione. Per il corpo effettivo dell'operazione, possiamo trovare l'angolo di rotazione moltiplicando l'angolo segreto di Dagonet, `angle`, per il fattore di scala di Lancillotto, `scale`.

Listato 9.2 operations.qs: un'operazione per impostare il gioco.

```
operation ApplyScaledRotation(
    angle : Double, scale : Double, (1)
    target : Qubit)
: Unit is Adj + Ctl { (2)
    R1(angle * scale, target); (3)

}
```

(1) Per indovinare, abbiamo bisogno di un'operazione quantistica che richiede due argomenti classici: uno passato da Dagonet e uno passato da Lancillotto.

(2) La parte `Adj + Ctl` della signature indica che questa operazione supporta il funtore `Adjoint` che abbiamo visto per la prima volta nel Capitolo 8 così come il funtore Controlled che vedremo più avanti in questo capitolo.

(3) L'operazione di rotazione `R1` qui è quasi identica all'operazione `Rz` che abbiamo visto alcune volte finora. La differenza tra `R1` e `Rz` diventerà più importante quando aggiungeremo il funtore Controlled.

> **NOTA**
>
> Quando si scrive il codice Q# in un proprio file (cioè, non in un notebook Jupyter), tutte le operazioni e le funzioni devono essere definite all'interno di un namespace. Questo aiuta a mantenere il nostro codice organizzato ed evita che il nostro codice entri in conflitto con il codice presente nelle varie librerie che usiamo nella nostra applicazione quantistica. Per brevità, spesso non mostriamo le dichiarazioni del namespace, ma il codice completo può sempre essere trovato nell'archivio di esempi di questo libro: https://github.com/crazy4-pi314/learn-qc-with-python-and-qsharp.

La Figura 9.2 mostra una rappresentazione visuale della configurazione del gioco di Dagonet.

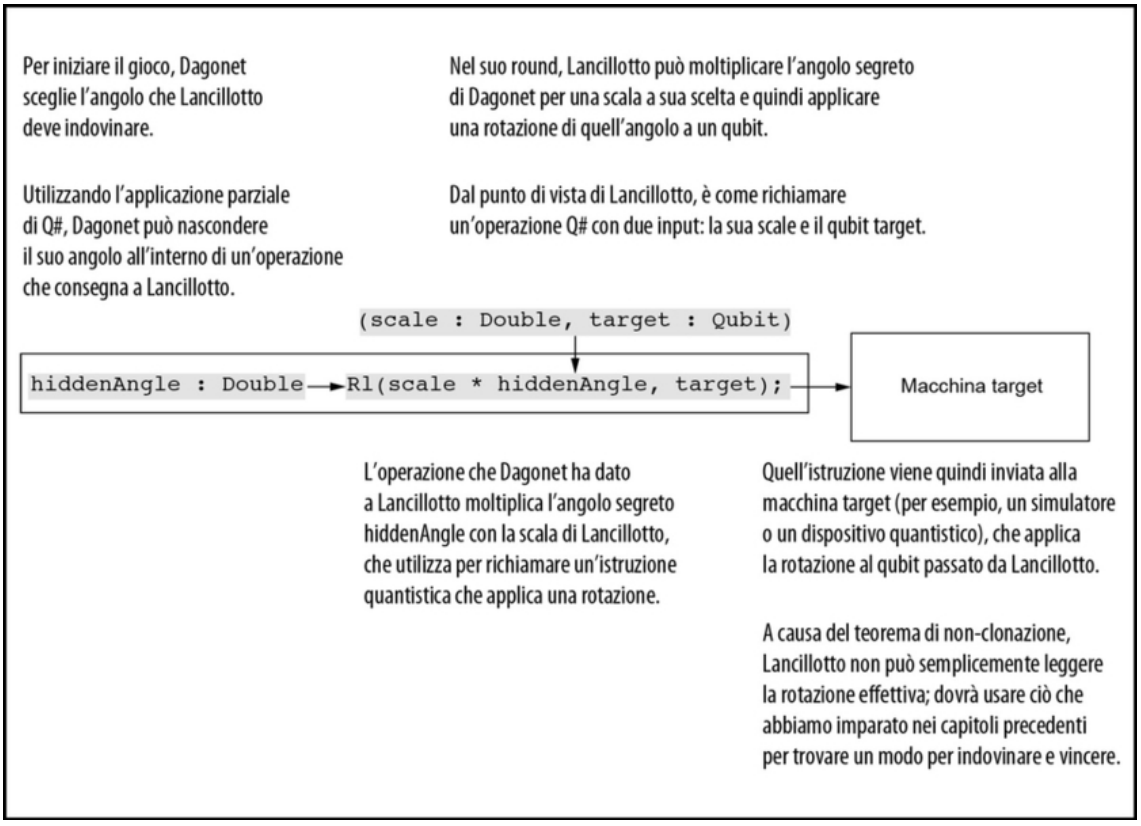


Figura 9.2 Utilizzo dell'applicazione parziale per nascondere l'angolo segreto durante il gioco di indovinelli di Dagonet. L'oracolo di Lancillotto ha un input per il suo parametro di scala, e può quindi scegliere la macchina target con cui usare l'operazione, ma non può "sbirciare" dentro l'operazione per conoscere l'angolo segreto.

Approfondimento: perché non misurare solo l'angolo?

Può sembrare che Lancillotto debba fare molti "salti mortali" per indovinare l'angolo segreto di Dagonet. Dopotutto, che cosa oltre al dovere e all'onore impedisce a Lancillotto di passare la scala `1.0` e leggere, semplicemente, l'angolo dalla fase applicata al suo qubit? Ma il teorema di non-clonazione interviene anche in questo caso: Lancillotto non può scoprire una fase con una singola misurazione.

Il modo più semplice per capirlo è fingere per un momento che Lancillotto possa farlo, e poi vedere che cosa va storto. Supponiamo che Dagonet nasconda un

angolo $\pi/2$ e che Lancillotto prepari il suo qubit nello stato $|+\rangle = 1 / \sqrt{2} (|0\rangle + |1\rangle)$. Se Lancillotto passa la scala `1.0`, il suo qubit finisce nello stato $1 / \sqrt{2} (|0\rangle + i|1\rangle)$. Se Lancillotto potesse misurare direttamente la fase $e^{i}\pi/2 = +i$ per indovinare l'angolo di Dagonet con un'unica misurazione, potrebbe usarla per preparare un'altra copia dello stato $1 / \sqrt{2} (|0\rangle + i|1\rangle)$, anche se Lancillotto non conoscesse la base corretta per eseguire la misurazione. In effetti, il dispositivo di misurazione magica di Lancillotto dovrebbe funzionare anche se Dagonet nascondesse l'angolo π, nel qual caso, Lancillotto finirebbe con un qubit nello stato $1 / \sqrt{2} (|0\rangle - |1\rangle)$.
In altre parole, se Lancillotto potesse conoscere l'angolo di Dagonet misurando direttamente le fasi, potrebbe anche fare copie degli stati, nella forma $1 / \sqrt{2} (|0\rangle + e^{i\varphi} |1\rangle)$ per ogni angolo φ, senza prima conoscere φ. Ciò viola piuttosto gravemente il teorema di non-clonazione, quindi possiamo tranquillamente concludere che Lancillotto dovrà fare un po' più di lavoro per vincere il gioco di Dagonet.

Una volta che abbiamo definito un'operazione in questo modo, Dagonet può utilizzare la funzione di applicazione parziale di Q# che abbiamo visto nel Capitolo 8 per nascondere il suo input. Quindi Lancillotto riceve un'operazione che può applicare ai suoi qubit, ma non in un modo che non gli consente di vedere direttamente l'angolo che sta cercando di indovinare.

Utilizzando `ApplyScaledRotation`, Dagonet può creare facilmente un'operazione che Lancillotto può poi richiamare. Per esempio, se Dagonet sceglie l'angolo `0.123`, può "nasconderlo" assegnando a Lancillotto l'operazione `ApplyScaledRotation(0.123, _, _)`. Come per gli esempi di applicazione parziale nel Capitolo 7, _ indica uno slot per i futuri input.

Come illustrato nella Figura 9.3, poiché `ApplyScaledRotation` ha tipo `(Double, Double, Qubit) => Unit is Adj + Ctl`, fornendo solo il primo input si ottiene un'operazione di tipo `(Double, Qubit) => Unit is Adj + Ctl`. Ciò significa che Lancillotto può fornire un input di tipo `Double`, un qubit cui vuole applicare la sua operazione, e usa `Adjoint` e il funtore che abbiamo visto nel Capitolo 6.

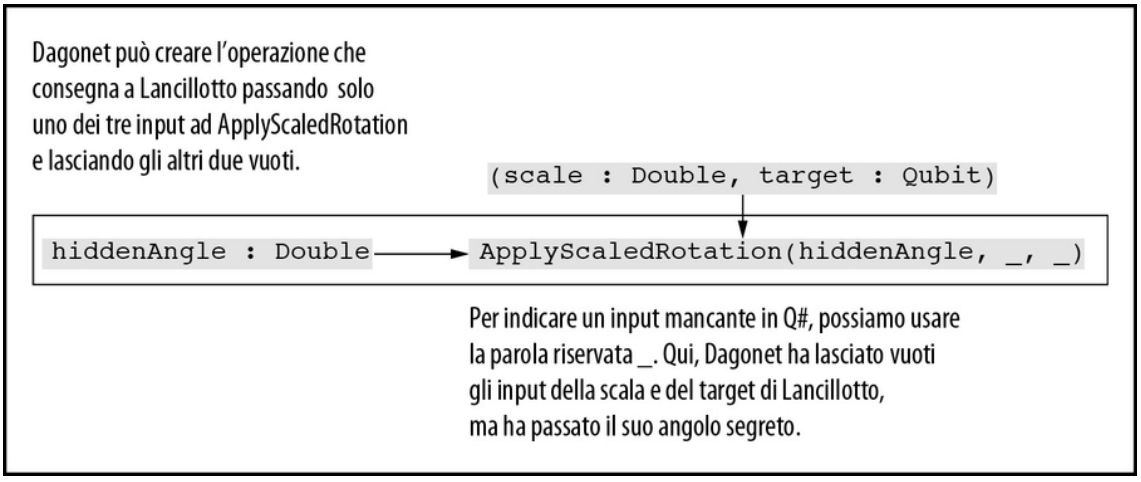


Figura 9.3 Applicazione parziale di ApplyScaledRotation per creare un'operazione per Lancillotto.

Il fatto che possiamo vedere il valore dell'angolo nella sintassi *non significa* che Lancillotto possa farlo. In effetti, le uniche cose che Lancillotto può fare con un'operazione o una funzione parzialmente applicata è richiamarla, applicarla parzialmente ulteriormente o passarla a un'altra funzione o operazione. Dal punto di vista di Lancillotto, `ApplyScaled Rotation(0.123, _, _)` è una scatola nera. Grazie a questo trucco dell'applicazione parziale, riceverà solo un'operazione che può prendere il suo valore di scala e può essere utilizzato per ruotare un qubit.

Possiamo semplificarci la vita come sviluppatori Q# dando un nome al tipo di operazione di Lancillotto, che sia un po' più facile da leggere rispetto a `(Double, Qubit) => Unit is Adj + Ctl`.

Nel prossimo paragrafo, vedremo come Q# ci permette di annotare le signature dei tipi che usiamo per il gioco di indovinelli di Dagonet e Lancillotto.

Tipi definiti dall'utente

Come abbiamo visto, i tipi giocano un loro ruolo in Q#, in particolare nelle signature di funzioni e operazioni. Abbiamo anche visto che sia le funzioni sia le operazioni accettano tuple e restituiscono tuple. In questo paragrafo, impareremo a costruire nuovi tipi in Q# e scopriremo perché ciò può essere utile.

In Q# (così come in molti altri linguaggi), vari tipi sono definiti dal linguaggio stesso: tipi come `Int`, `Qubit` e `Result`, che abbiamo già visto.

> **SUGGERIMENTO**
>
> Per un elenco completo dei tipi base, consultate la documentazione del linguaggio Q# su https://docs.microsoft.com/azure/quantum/user-guide/language/typesystem/#primitive-types.

Partendo da questi tipi base, possiamo creare tipi per array aggiungendo la coppia di parentesi `[]` dopo il nome del tipo. Per esempio, nel gioco di questo capitolo, dovremo probabilmente inserire un array di `Double` per rappresentare i molteplici input di Lancillotto per l'operazione di Dagonet. Possiamo usare `Double[]` per indicare un array di valori `Double`.

Listato 9.3 Definizione di un tipo per array di Double di lunghezza 10.

```
let scales = EmptyArray<Double>(10); (1)
```

(1) `EmptyArray` proviene dal namespace `Microsoft.Quantum.Arrays`. Assicuratevi di aprirlo prima di impiegare questo codice.

In Q# possiamo definire i nostri tipi anche con l'istruzione `newtype`, che ci permette di dichiarare nuovi *tipi definiti dall'utente*. Ci sono due ragioni principali per usare nuovi tipi:

- per comodità;
- per comunicare un intento.

Il primo motivo è una questione di *comodità*. A volte la signature di un tipo per una funzione o per un'operazione può diventare piuttosto

lunga, quindi possiamo definire il nostro tipo per usarlo come una sorta di scorciatoia. Un altro motivo per cui potremmo voler definire un nostro tipo è per comunicare l'intento. Supponiamo che la nostra operazione richieda una tupla di due valori `Double` che rappresentano un numero complesso. La definizione di un nuovo tipo `Complex` può ricordare a noi e ai membri del nostro team che cosa rappresenta quella tupla. Il Quantum Development Kit fornisce diverse funzioni, operazioni e tipi con le librerie Q#; il seguente caso, per esempio, definisce il tipo `Complex`.

Listato 9.4 Come vengono definiti i numeri complessi nel runtime Q#.

```
namespace Microsoft.Quantum.Math { (1)
    newtype Complex = ( (2)
        Real: Double,
        Imag: Double
    );

}
```

(1) Il tipo per i numeri complessi è implementato nel namespace `Microsoft.Quantum.Math`. Possiamo utilizzare questo tipo includendo l'istruzione `open Microsoft.Quantum.Math;` nella nostra applicazione quantistica.

(2) Il tipo `Complex` è definito come una tupla di due valori `Double`: il primo elemento è denominato `Real` e il secondo è denominato `Imag`.

> **SUGGERIMENTO**
>
> Il Quantum Development Kit è open source, quindi se siete curiosi, potete sempre controllare come funzionano le varie parti del linguaggio Q#, del runtime, del compilatore e delle librerie standard. Per esempio, la definizione del tipo `Complex` è contenuta nel file `src/Simulation/QSharpFoundation/Math/Types.qs` nell'archivio runtime di Q# in https://github.com/microsoft/qsharp-runtime.

Come illustrato nella Figura 9.4, ci sono due modi per estrarre i diversi elementi da un tipo definito dall'utente. Potete concatenare il

nome del tipo e del suo elemento con l'operatore `::` o "espandere" il tipo con l'operatore `!` per tornare al tipo originale racchiuso nel nuovo tipo:

```
function TakesComplex(complex : Complex) : Unit {
    let realFromNamedItem = complex::Real; (1)
    let (real, imag) = complex!; (2)

}
```

(1) Poiché il tipo `Complex` è definito con un elemento denominato `Real`, possiamo accedere a quell'elemento con `::Real` per recuperare la parte reale del nostro input.

(2) In alternativa, poiché `Complex` è definito come una tupla di tipo `(Double, Double)`, l'operatore `!` ci riporta alle parti reale e immaginaria del numero complesso, senza l'involucro del tipo `Complex`.

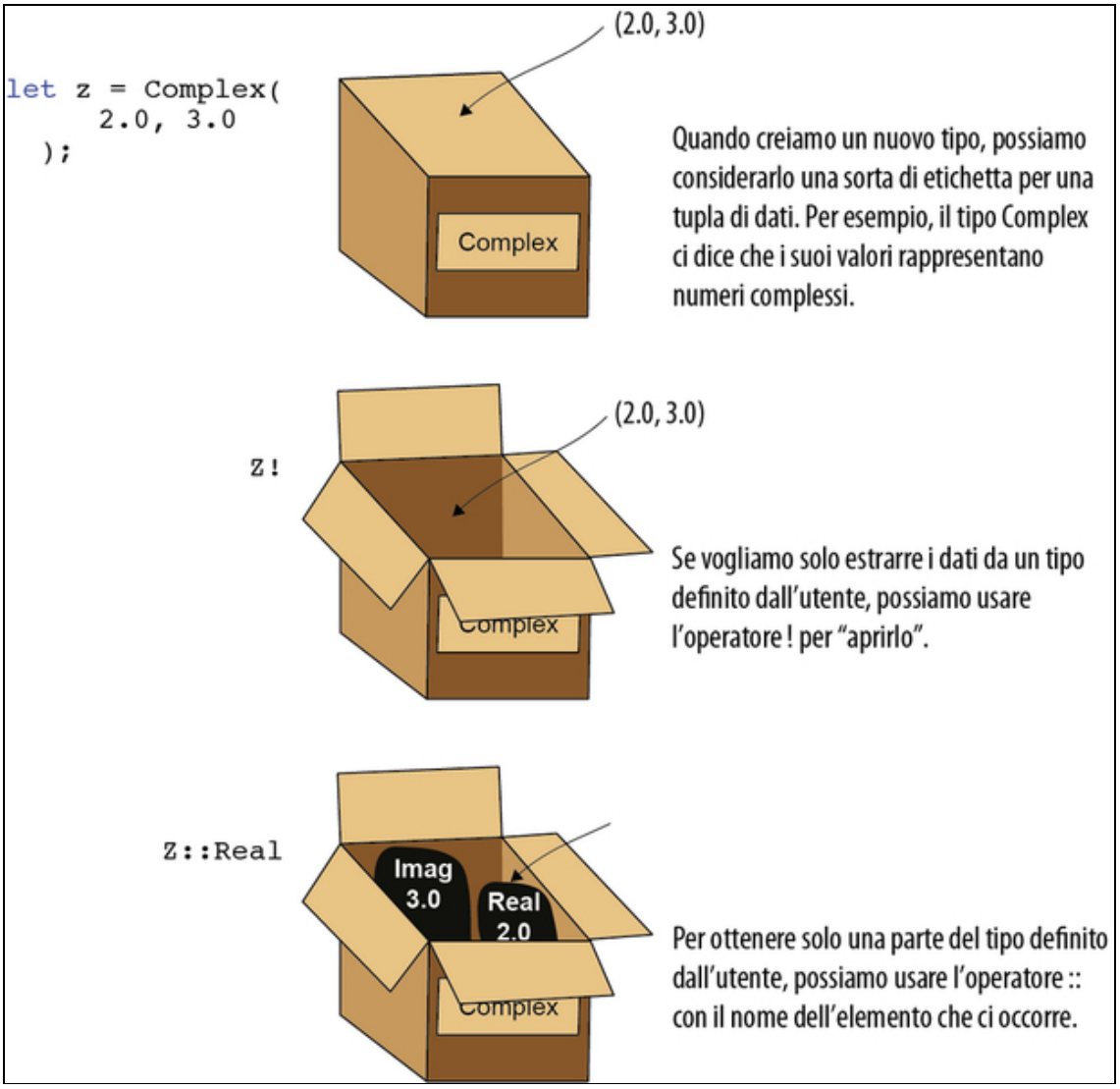


Figura 9.4 Utilizzo degli operatori :: e ! con i tipi definiti dall'utente. Possiamo considerare un tipo come una tupla di dati dotata di un'etichetta. L'operatore :: ci permette di accedere ai dati in base al nome, mentre l'operatore ! espande un tipo definito dall'utente nel suo tipo base.

SUGGERIMENTO

Entrambi i modi di lavorare con tipi definiti dall'utente sono utili in casi diversi, ma la maggior parte delle volte in questo libro continueremo a usare gli elementi per nome e con l'operatore `::`.

Una volta definito un nuovo tipo, questo può anche servire per creare una nuova istanza di quel tipo. Per esempio, il tipo `Complex` si comporta come una funzione che crea un nuovo numero complesso ricevendo in input una tupla di due valori `Double`. La situazione è simile a quella di Python, dove i tipi sono anche funzioni che creano istanze di quel tipo.

Listato 9.5 Creazione di un numero complesso con il tipo Complex.

```
let imaginaryUnit = Complex(0.0, 1.0); (1)
```

(1) La definizione di un tipo con `newtype` definisce anche una nuova funzione avente lo stesso nome del tipo, la quale restituisce valori del nuovo tipo. Per esempio, possiamo richiamare `Complex` come una funzione con due input `Double` che rappresentano le parti reale e immaginaria del nuovo valore `Complex`. Qui, definiamo un valore `Complex` che rappresenta $0 + 1.0i$ (`0 + 1j` in notazione Python), noto anche come unità immaginaria.

ESERCIZIO 9.1 TIPI PER LE STRATEGIE

Nel Capitolo 4, abbiamo usato annotazioni di tipo Python per rappresentare il concetto di *strategia* nel gioco CHSH. I tipi definiti dall'utente di Q# possono essere utilizzati in modo simile. Dimostratelo definendo un nuovo tipo per le strategie CHSH, quindi usate il nuovo tipo per implementare la strategia costante del Capitolo 4.

Suggerimento: le parti nostra e di Eve della strategia possono essere rappresentate come operazioni che prendono un `Result` e producono un `Result`, cioè come operazioni di tipo `Result => Result`.

SOLUZIONI DEGLI ESERCIZI

Tutte le soluzioni degli esercizi di questo libro si trovano nel suo archivio di codice: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Aprite la cartella del capitolo in cui vi trovate e poi il notebook Jupyter con il nome che menziona le soluzioni degli esercizi.

Per il gioco in questo capitolo, abbiamo definito un nuovo tipo, sia per spiegare come intendiamo usarlo sia come comoda scorciatoia per

il tipo di operazione che Lancillotto ottiene come parte del gioco. Nel seguente listato, possiamo vedere la definizione di questo nuovo tipo, chiamato `ScalableOperation`, come una tupla con un input denominato `Apply`.

Listato 9.6 operations.qs: predisposizione del gioco.

```
newtype ScalableOperation = ( (1)
    Apply: ((Double, Qubit) => Unit is Adj + Ctl) (2)
);
function HiddenRotation(hiddenAngle : Double) : ScalableOperation {
    return ScalableOperation( (3)
        ApplyScaledRotation(hiddenAngle, _, _) (4)
    );

}
```

(1) Possiamo dichiarare un nuovo tipo con l'istruzione `newtype`, specificandone il nome e definendo il tipo sottostante su cui si basa il nuovo tipo.

(2) Possiamo dare un nome ai vari elementi di un nuovo tipo usando la stessa sintassi della dichiarazione della signature per un'operazione o una funzione. Qui, il nuovo tipo definito dall'utente ha un singolo elemento denominato `Apply` che consente di richiamare l'operazione racchiusa in `ScalableOperation`.

(3) Una volta definito, possiamo utilizzare un nuovo tipo come faremmo con qualsiasi altro tipo. Qui definiamo una funzione che restituisce valori di tipo `ScalableOperation`.

(4) Possiamo facilmente creare valori di output richiamando `ScalableOperation` con l'operazione da inserire nel nostro nuovo tipo. In questo esempio, possiamo creare nuove istanze di `ScalableOperation` utilizzando la stessa applicazione parziale di `ApplyScaledRotation` che abbiamo visto in precedenza nel capitolo.

> **SUGGERIMENTO**
>
> Quando definiamo gli input per le funzioni e le operazioni in Q#, questi input hanno nomi che iniziano con lettere minuscole. Nel Listato 9.6, tuttavia,

l'elemento `Apply` di `ScalableOperation` inizia con una lettera maiuscola. Questo perché gli input di una funzione o operazione hanno significato solo all'interno di quell'ambito di chiamata, mentre gli elementi con nome hanno una visibilità più ampia. Possiamo usare le lettere maiuscole per gli input e per gli elementi con nome per chiarire meglio dove cercare le loro definizioni.

La funzione `HiddenRotation` definita nel Listato 9.6 ci aiuta a implementare il gioco di Lancillotto e Dagonet, in quanto ci dà un modo per consentire a Dagonet di nascondere il proprio angolo. Richiamando `HiddenRotation` con l'angolo di Dagonet otteniamo una nuova `ScalableOperation` che Lancillotto può richiamare per raccogliere i dati di cui ha bisogno per indovinare l'angolo segreto. La Figura 9.5 mostra la nuova configurazione del gioco di Dagonet.

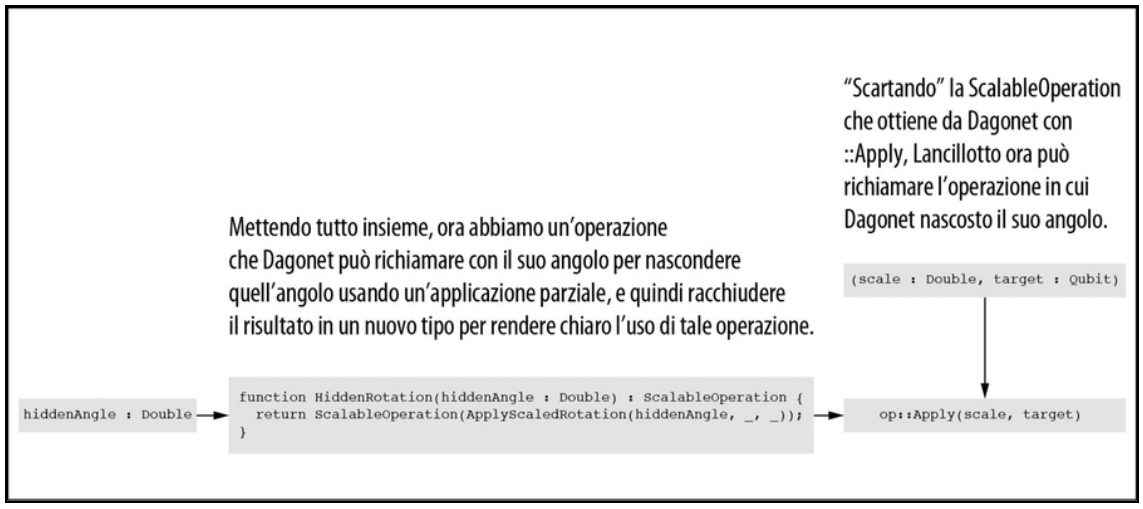


Figura 9.5 Il gioco di Dagonet con l'applicazione parziale e un nuovo tipo. Dagonet utilizza una funzione per costruire il nuovo tipo che passa a Lancillotto, rappresentando la rotazione con il suo angolo segreto.

Con alcuni nuovi tipi e un modo per consentire a Dagonet di nascondere il proprio angolo, continuiamo l'implementazione del resto del gioco. Abbiamo tutto ciò di cui abbiamo bisogno per il passaggio successivo: stimare la probabilità di ogni misurazione che effettuiamo durante la partita fra Lancillotto e Dagonet. La situazione è molto

simile a quella in cui dovremmo stimare le probabilità nel lancio di una moneta (vedi Figura 9.6).

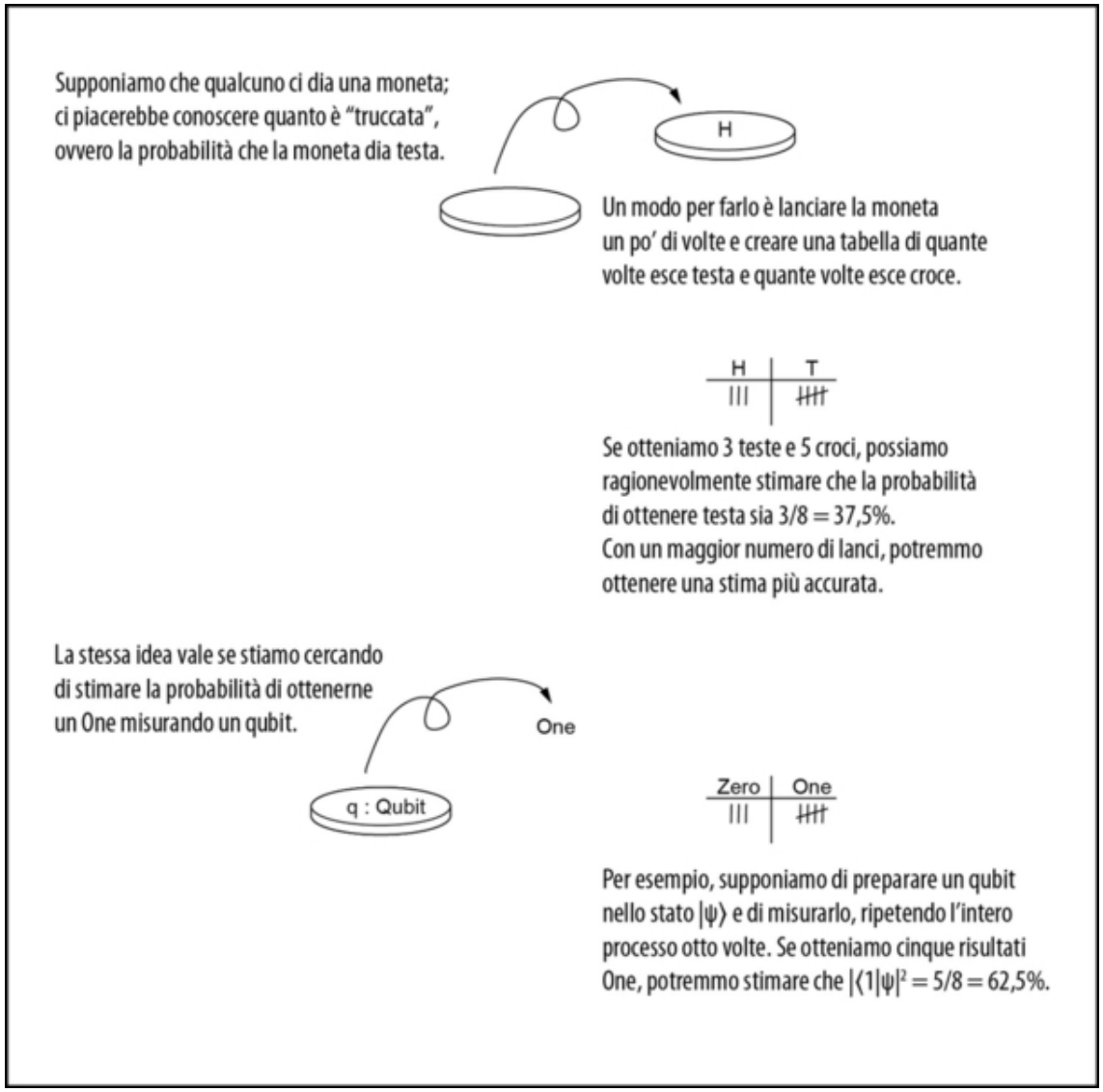


Figura 9.6 La stima di Lancillotto è simile alla stima del risultato dei lanci di una moneta. Può stimare lo "squilibrio" di una moneta lanciandola più volte e contando il numero di "teste". Allo stesso modo, Lancillotto può preparare lo stesso qubit nello stesso stato più volte, misurarlo ogni volta e contare i risultati delle misurazioni.

Vediamo come sarà il codice per questa procedura dei lanci nel gioco di Dagonet. Notate che poiché Lancillotto e Dagonet hanno concordato come asse di rotazione per il loro gioco l'asse *Z*,

Lancillotto può preparare il suo qubit `target` nello stato |+⟩, in modo che la rotazione di Dagonet faccia effettivamente qualcosa.

Listato 9.7 operations.qs: stima della probabilità di misurare |1⟩.

```
operation EstimateProbabilityAtScale(
    scale : Double, (1)
    nMeasurements : Int, (2)
    op : ScalableOperation) (3)
: Double { (4)
    mutable nOnes = 0; (5)
    for idx in 0..nMeasurements - 1 {
        use target = Qubit(); (6)
        within { (7)
            H(target); (8)
        } apply {
            op::Apply(scale, target); (9)
        }
        set nOnes += Meas.MResetZ(target) == One ? 1 | 0; (10)
    }
    return Convert.IntAsDouble(nOnes) /
Convert.IntAsDouble(nMeasurements); (11)

}
```

(1) L'operazione di Lancillotto deve prendere un valore `Double` che rappresenta la scala scelta per eseguire l'operazione datagli da Dagonet.

(2) Lancillotto sceglie un numero di volte per misurare il suo qubit, per ottenere la sua stima della probabilità.

(3) L'ultimo input è di tipo `ScalableOperation`, il tipo dichiarato in precedenza nel capitolo. Questo input rappresenta l'operazione che Dagonet dà a Lancillotto.

(4) Come output, Lancillotto vuole restituire una stima della probabilità, quindi possiamo dichiararlo come un output `Double`.

(5) Per tenere traccia del numero di risultati |1⟩ osservati finora, definiamo una variabile `mutable` con il valore `Int 0`.

(6) Per ogni misurazione, dobbiamo allocare un qubit che è il `target` per l'operazione di Dagonet.

(7) Utilizza le parole riservate `within` e `apply` per il problema delle "calze e scarpe" che abbiamo imparato a conoscere nel Capitolo 8.
(8) Implementa la strategia di Lancillotto utilizzando l'operazione `H` per preparare un qubit nello stato $|+\rangle$.
(9) Una volta che abbiamo preparato l'input per l'operazione di Dagonet, lo richiamiamo usando `::Apply` per "aprire" il nuovo tipo `ScalableOperation`.
(10)> L'operatore ternario `?|` (proprio come l'operatore `if … else` in Python o l'operatore `?:` in C, C++ e C#) fornisce un modo comodo per incrementare `nOnes`.
(11) Per ottenere la nostra stima finale della probabilità di misurare $|1\rangle$, calcoliamo il rapporto tra `One` e il totale. La funzione `Convert.IntAsDouble` ci aiuta a restituire un numero in virgola mobile.

Nel Listato 9.7, il blocco `within`/`apply` si assicura che il qubit di Lancillotto sia sull'asse corretto. Possiamo contare quante volte la misurazione finale restituisce un risultato `One` aggiungendo `1` o `0` a `nOnes`. Qui, l'operatore ternario `?|` (proprio come l'operatore `if … else` in Python o l'operatore `?:` in C, C++ e C#) fornisce un modo comodo per incrementare `nOnes`.

Un'operazione con qualsiasi altro nome
Avrete notato che le *operazioni* tendono a essere denominate usando *verbi*, mentre le *funzioni* tendono a essere denominate usando *nomi*. Questo aiuta a ricordare la distinzione che avete visto nel Capitolo 7: una funzione *è* qualcosa, mentre un'operazione *fa* qualcosa. La coerenza dei nomi può aiutarvi a capire come funziona un programma quantistico; Q# usa convenzioni come questa in tutto il linguaggio e nelle librerie.

Ora possiamo scrivere un'operazione che esegue l'intero gioco e restituisce tutto ciò di cui Lancillotto ha bisogno per indovinare l'angolo segreto di Dagonet.

Listato 9.8 operations.qs: esecuzione completa del gioco.

```
operation RunGame(
    hiddenAngle : Double, scales : Double[], nMeasurementsPerScale : Int
) : Double[] {
    let hiddenRotation = HiddenRotation(hiddenAngle); (1)
    return Arrays.ForEach( (2)
        EstimateProbabilityAtScale( (3)
            _,
            nMeasurementsPerScale,
            hiddenRotation
        ),
        scales (4)
    );

}
```

(1) Crea un nuovo valore `ScalableOperation` che nasconde l'angolo di Dagonet utilizzando la funzione `HiddenRotation` che abbiamo scritto in precedenza.
(2) L'operazione `ForEach` di `Microsoft.Quantum.Arrays` (cui abbiamo dato il nome abbreviato `Arrays`) accetta un'operazione e la applica a ogni elemento dell'array `scales`.
(3) Per ottenere l'operazione che passiamo a `ForEach`, utilizziamo un'applicazione parziale, per bloccare le misurazioni eseguite da Lancillotto a ciascuna scala diversa e l'operazione nascosta che gli è stata data da Dagonet.
(4) Quando passiamo `scale` come secondo input a `ForEach`, nello slot di applicazione parziale _ viene sostituito ogni elemento di `scale`.

> **NOTA**
> Può sembrare strano che `ForEach` agisca come `map` in Python e in altri linguaggi quando Q# dispone anche di `Microsoft.Quantum.Arrays.Mapped`. La differenza fondamentale è che `ForEach` accetta un'operazione, mentre `Mapped` accetta una funzione.

Affinché Lancillotto possa effettivamente dare un senso a tutti i dati che riceve dal suo programma Q#, potrebbe essere utile utilizzare alcune buone vecchie tecniche classiche di scienza dei dati. Dal

momento che Python è bravissimo a farlo, eseguire la nostra nuova operazione `RunGame` da un programma host Python può essere un modo eccellente per aiutare Lancillotto.

Esecuzione di codice Q# da Python

Nei capitoli precedenti, abbiamo eseguito il nostro codice Q# in un notebook Jupyter con un kernel Q#. In questo capitolo, vogliamo esaminare un modo diverso di eseguire il codice Q#: da Python. Richiamare il codice Q# da Python può essere utile in diversi scenari, specialmente se vogliamo pre-elaborare i dati prima di usarli in Q#, o se vogliamo visualizzare l'output dal nostro programma quantistico.

Cominciamo a scrivere i file per implementare il gioco di Dagonet e Lancillotto. Per provare l'interoperabilità fra Q# e Python, utilizzeremo un programma host Python il quale esegue il programma Q#. Ciò significa che avremo due file per il gioco: `operations.qs` e un file `host.py` che lanceremo per eseguire il gioco. Vediamo prima di tutto il file `host.py`, per scoprire come possiamo interagire con Q# da Python (vedi Figura 9.7).

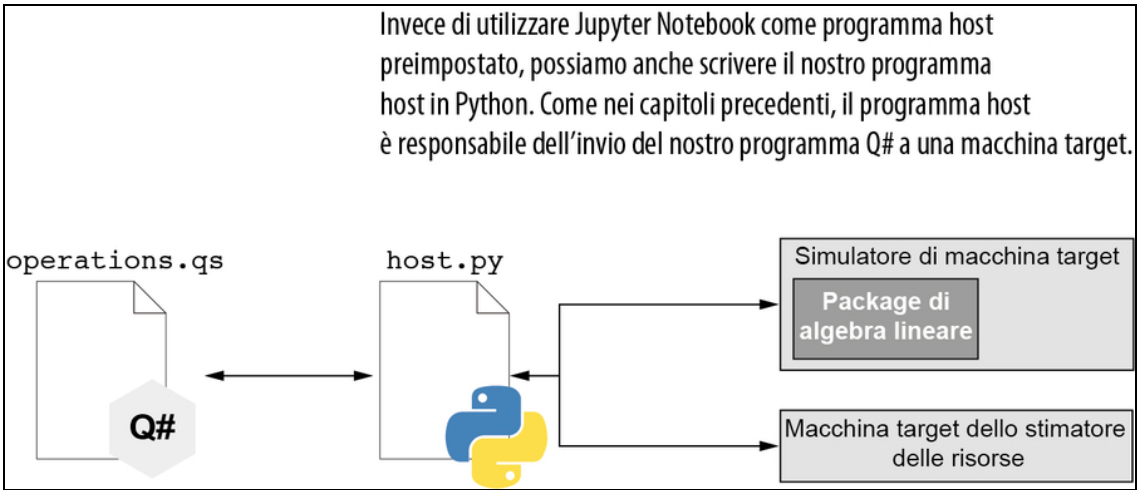


Figura 9.7 Utilizzo di programmi host per Q# scritti in Python. Come l'host Jupyter Notebook, un programma Python può coordinare l'invio del programma Q# a una particolare macchina target e la raccolta dei risultati.

Tutte le funzionalità di interoperabilità di cui abbiamo bisogno tra Python e Q# sono fornite dal package Python `qsharp`.

SUGGERIMENTO

Le istruzioni per l'installazione del package `qsharp` si trovano nell'Appendice A.

Avendo a disposizione il package `qsharp`, possiamo importarlo proprio come qualsiasi altro package Python. Ecco un piccolo file Python di esempio in cui possiamo vederlo in azione.

Listato 9.9 qsharp-interop.py: utilizzo del codice Q# direttamente in Python.

```
import qsharp (1)
prepare_qubit = qsharp.compile("""  (2)
    open Microsoft.Quantum.Diagnostics; (3)
    operation PrepareQubit(): Unit { (4)
        using (qubit = Qubit()) {
            DumpMachine();
        }
    }
    """)

if __name__ == "__main__": prepare_qubit.simulate() (5)
```

(1) Il package `qsharp` Python deve essere importato come qualsiasi altro package Python.

(2) Utilizza la funzione Python `qsharp.compile` per prendere una stringa contenente codice Q# e compilarla per l'uso nel file Python.

(3) Proprio come un normale file Q#, è necessario includere istruzioni `open` per utilizzare le diverse parti della libreria standard Q#.

(4) L'operazione descritta da questa stringa di codice Q# prepara semplicemente un qubit nello stato $|0\rangle$ e utilizza `DumpMachine` per mostrare ciò che il computer di destinazione conosce su tale qubit.

(5) Vogliamo anche usare il codice richiamabile definito come `prepare_qubit`, quindi possiamo usare il metodo `simulate` del package `qsharp`, che esegue frammenti di codice Q# compilati in precedenza.

Proviamo a eseguire lo script `qsharp-interop.py` del Listato 9.9.

Listato 9.10 Esecuzione del Listato 9.9.

```
$ python qsharp-interop.py
Preparing Q# environment...
# wave function for qubits with ids (least to most significant): 0
|0⟩: 1.000000 + 0.000000 i == ******************** [ 1.000000 ] --- [
0.00000 rad ]

|1⟩: 0.000000 + 0.000000 i ==                       [ 0.000000 ]
```

> **SUGGERIMENTO**
>
> Se eseguite il codice da un notebook Q# Jupyter, l'output del frammento di codice Q# avrà un aspetto diverso. La Figura 9.9, più avanti nel capitolo, mostra un esempio.

Dall'output presentato nel Listato 9.10, possiamo vedere che prepara effettivamente uno stato $|0\rangle$, poiché l'unico termine nell'output che ha un coefficiente `1.0` è lo stato $|0\rangle$.

Il package python `qsharp` cerca le operazioni o le funzioni Q# definite nei file `*.qs` nella stessa directory del nostro programma Python. In questo caso, procedendo nel resto di questo capitolo, aggiungeremo istruzioni in un file Q# chiamato `operations.qs`. Questo è un modo abbastanza comodo per avviare il nostro file `host.py` per il gioco. Il package `qsharp` ci consente di importare le operazioni e le funzioni dai namespace nei file Q# contenuti nella stessa directory di `host.py`. Abbiamo già visto `RunGame` in precedenza e presto vedremo `RunGameUsingControlledRotations`.

Listato 9.11 host.py: l'inizio del gioco di stima della fase.

```
import qsharp (1)
from PhaseEstimation import RunGame, RunGameUsingControlledRotations (2)
from typing import Any (3)
```

```
import scipy.optimize as optimization
import numpy as np

BIGGEST_ANGLE = 2 * np.pi
```

(1) Importa il package Q# Python.
(2) Importa le operazioni `RunGame` e `RunGameUsingControlledRotations` da `operations.qs` per creare automaticamente gli oggetti Python che rappresentano ogni operazione Q# che importiamo.
(3) Il resto delle importazioni aiuta con i suggerimenti di tipo di Python, visualizzando i risultati della nostra simulazione Q# e adattando i dati della misurazione per ottenere l'ipotesi finale di Lancillotto.

Ora che abbiamo importato e impostato il nostro file Python, scriviamo `run_game_at_scales`, la funzione che richiama le operazioni Q#.

Listato 9.12 host.py: la funzione Python che richiama le operazioni Q#.

```
def run_game_at_scales(scales: np.ndarray,
        n_measurements_per_scale: int = 100,
        control: bool = False
    ) -> Any: (1)
    hidden_angle = np.random.random() * BIGGEST_ANGLE (2)
    print(f"Pssst the hidden angle is {hidden_angle}, good luck!")
    return ( (3)
        RunGameUsingControlledRotations
        if control else RunGame
    ).simulate( (4)
        hiddenAngle=hidden_angle,
        nMeasurementsPerScale=n_measurements_per_scale,
        scales=list(scales)
    )
```

(1) Imposta il suggerimento del tipo restituito su `Any`, che dice a Python di non preoccuparsi del controllo del tipo sul valore restituito da questa funzione.
(2) Dagonet sceglie l'angolo segreto che Lancillotto deve indovinare.
(3) Il valore restituito da `run_game_at_scales` è condizionato da `control`, che ci permette di scegliere tra due simulazioni che svilupperemo per

questo gioco (per ora usiamo `control = False`).

❹ Quando `qsharp` importa queste operazioni, le loro rappresentazioni Python hanno un metodo `simulate` che accetta gli argomenti richiesti e li passa al simulatore Q#.

Questo file Python dovrebbe essere eseguibile come script, quindi dobbiamo anche definire `__main__`. È qui che possiamo implementare il ragionamento di Lancillotto utilizzando il nostro programma host in Python per prendere le misurazioni e le scale e adattarle a un modello per la rotazione scelta da Dagonet. Il miglior modello del modo in cui l'angolo di rotazione cambia i risultati della misurazione è dato dalla seguente formula, dove θ è l'angolo segreto di Dagonet e `scale` è il fattore di scala di Lancillotto:

$$\Pr(1) = \sin\left(\frac{\theta * \text{scale}}{2}\right)^2$$

ESERCIZIO 9.2 RINATO

Questo modello può essere trovato se usiamo la regola di Born. Di seguito viene mostrata la definizione già presentata nel Capitolo 2. Scoprite se riuscite a tracciare in Python il valore risultante come una funzione della scala di Lancillotto. Il vostro grafico sembra quello di una funzione trigonometrica?

Pr(misurazione|stato) = |<misurazione|stato>|2

Suggerimento: per le misurazioni di Lancillotto, la parte ⟨misurazione| della regola di Born è data da ⟨1|. Immediatamente prima di misurare, il suo qubit è nello stato $HR1(\theta * \text{scale})\,H|0\rangle$. È possibile simulare l'operazione `R1` in QuTiP utilizzando la forma matriciale nel riferimento Q# in https://docs.microsoft.com/qsharp/api/qsharp/microsoft.quantum.intrinsic.r1.

Una volta che abbiamo quel modello e i dati, possiamo usare la funzione `scipy.optimize` del package Python SciPy per adattare i nostri dati al modello. Il valore che trova per il parametro θ è l'angolo segreto

di Dagonet. Il prossimo listato mostra come mettere insieme tutto questo.

Listato 9.13 host.py: eseguire host.py come script.

```
if __name__ == "__main__":
    import matplotlib.pyplot as plt (1)
    scales = np.linspace(0, 2, 101) (2)
    for control in (False, True): (3)
        data = run_game_at_scales(scales, control=control) (4)
        def rotation_model(scale, angle): (5)
            return np.sin(angle * scale / 2) ** 2
        angle_guess, est_error = optimization.curve_fit( (6)
            rotation_model, scales, data, BIGGEST_ANGLE / 2,
            bounds=[0, BIGGEST_ANGLE]
        )
        print(f"The hidden angle you think was {angle_guess}!")
        plt.figure() (7)
        plt.plot(scales, data, 'o')
        plt.title("Probability of Lancelot measuring One at each scale")
        plt.xlabel("Lancelot's input scale value")
        plt.ylabel("Lancelot's probability of measuring a One")
        plt.plot(scales, rotation_model(scales, angle_guess))

    plt.show() (8)
```

(1) Questo script traccia i dati e i risultati adattati, quindi è necessario importare `matplotlib`.

(2) L'elenco degli input di Lancillotto per il gioco (cioè, le sue `scales`) viene generato come una lista sequenziale di numeri regolarmente distanziati da `np.linspace`.

(3) Questo script esegue entrambe le versioni della simulazione del gioco, in modo da poterle confrontare. Non preoccupatevi per il caso `control = True`, per ora; ci torneremo fra poco.

(4) Memorizza il risultato della simulazione del codice Q# eseguita da Python in `run_game_at_scales`.

(5) Rappresenta l'operazione sul qubit. Lancillotto può prendere i dati che ottiene, adattarli al modello ed estrarre un'ipotesi per l'angolo.

(6) La funzione standard di `scipy`, `optimization.curve_fit`, prende un modello di una funzione, gli input, i dati misurati e un'ipotesi iniziale

per cercare di adattare tutti i parametri del modello.
(7) Convalidare l'adattamento trovato da `optimization.curve_fit` è importante per poter tracciare sia i dati sia il modello adattato, per vedere se sembra corretto.
(8) Visualizza i grafici con i dati e lo adatta a una nuova finestra.

Ora che abbiamo un programma host e che possiamo usarlo per eseguire l'intero gioco, possiamo vedere che Lancillotto fa un lavoro abbastanza ragionevole nel capire quale angolo Dagonet ha nascosto nella sua operazione Q#. Prendendo diverse misurazioni e utilizzando le classiche tecniche di scienza dei dati, Lancillotto può stimare la fase che l'operazione di Dagonet applica ai suoi qubit. L'esecuzione di `host.py` dovrebbe generare due finestre pop-up che mostrano i grafici delle probabilità di misurazione come una funzione della scala impostata da Lancillotto per le due strategie che può utilizzare (Figura 9.8). Il primo è l'approccio che abbiamo già delineato. Implementeremo il secondo nell'ultimo paragrafo del capitolo.

SUGGERIMENTO

Poiché i package `SciPy` non sono perfetti, a volte il parametro adattato che trova non è corretto. Eseguitelo un paio di volte, e magari l'algoritmo farà meglio la prossima volta. Se avete domande sul package di plottaggio `matplotlib`, consultate questi altri titoli di Manning: *Data Science Bootcamp* (Capitolo 2; di Leonard Apeltsin; di prossima pubblicazione) e *Data Science with Python and Dask* (Capitoli 7 e 8; di Jesse C. Daniel; 2019).

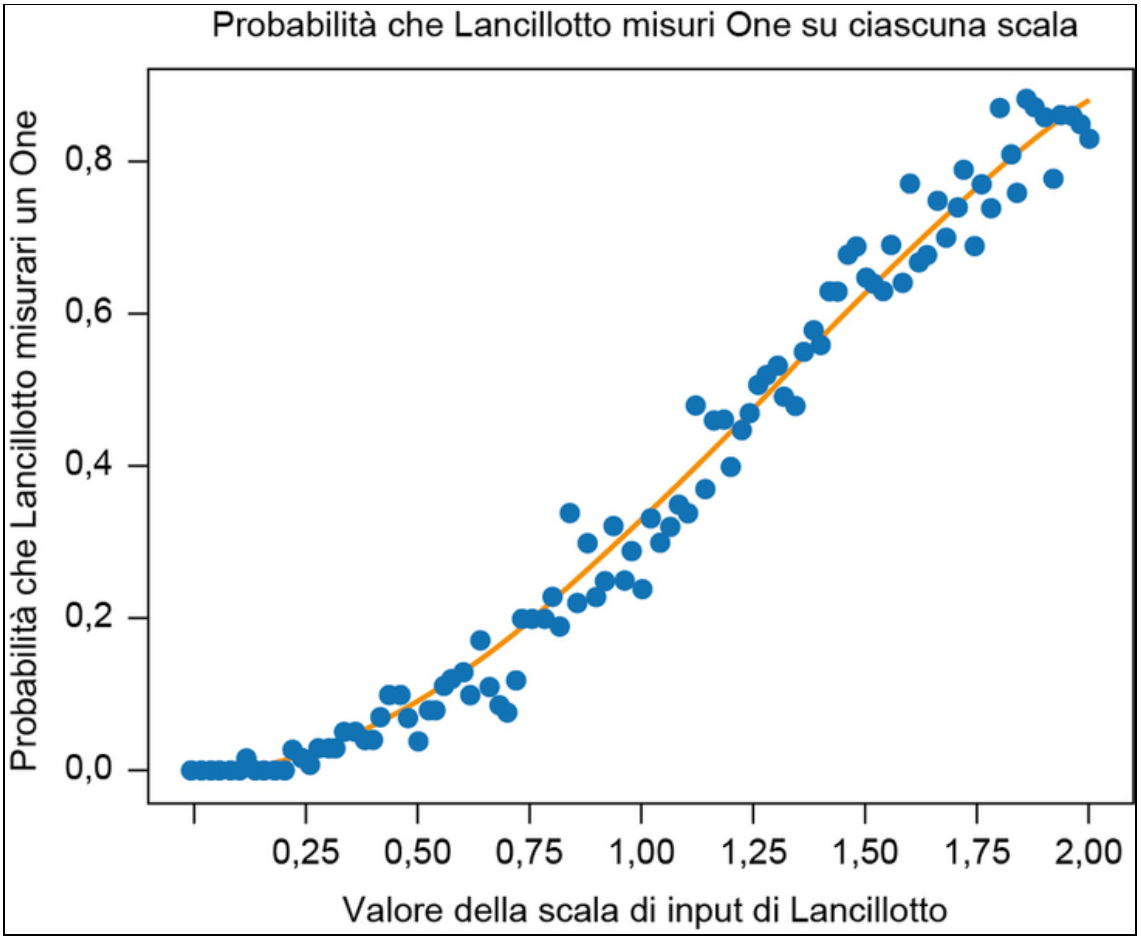


Figura 9.8 Un esempio di uno dei due grafici che dovrebbero essere visualizzati da host.py.

Possiamo vedere da questi grafici che siamo stati in grado di impiegare piuttosto bene i dati di Lancillotto. Ciò significa che il valore adattato che troviamo in `angle_guess` è un'approssimazione piuttosto buona dell'angolo segreto di Dagonet.

Ma c'è ancora un altro fastidioso problema con la strategia di Lancillotto: ogni volta che esegue una misurazione, deve preparare l'input giusto da passare all'operazione di Dagonet. In questo particolare gioco, potrebbe non essere un grosso problema, ma dal momento che nel prossimo capitolo esploriamo applicazioni più grandi di questo gioco, può essere costoso preparare ogni volta il registro di

input nello stato corretto. Fortunatamente, possiamo usare le *operazioni controllate* per riutilizzare gli stessi input più e più volte, come vedremo nel resto di questo capitolo.

Avete già visto altri esempi di operazioni controllate (come CNOT), ma anche molte altre operazioni quantistiche possono essere applicate in modo condizionale, il che può essere molto utile. Le operazioni controllate, insieme all'ultimo nuovo concetto di quantum computing di cui abbiamo bisogno (gli *autostati*), ci aiuteranno a implementare una tecnica cui abbiamo accennato alla fine del Capitolo 8: il *phase kickback*.

SUGGERIMENTO

Nei prossimi paragrafi trovate molte discussioni sulle fasi locali e globali. Ricordiamo che la fase globale è un coefficiente complesso che può essere scomposto in tutti i termini del nostro stato e non può essere osservato. Per un aggiornamento sulla fase, rileggete i Capitoli da 4 a 6.

Autostati e fasi locali

Come abbiamo visto, l'operazione quantistica `x` ci permette di invertire i bit ($|0\rangle \leftrightarrow |1\rangle$) e l'operazione `z` ci permette di invertire le fasi ($|+\rangle \leftrightarrow |-\rangle$). Entrambe queste operazioni, tuttavia, applicano solo fasi globali ad alcuni stati di input. Come abbiamo visto nei capitoli precedenti, non possiamo realmente imparare nulla sulle fasi globali, pertanto è fondamentale capire quali stati non vengono toccati da ciascuna operazione, per capire che cosa possiamo apprendere mediante l'applicazione di tale operazione.

Per esempio, rivediamo l'operazione `z`. Nel seguente listato possiamo vedere che cosa succede quando proviamo a usare `z` non per invertire un qubit tra gli stati $|+\rangle$ e $|-\rangle$, ma un qubit di input nello stato $|0\rangle$.

Listato 9.14 Applicazione di Z per un qubit nello stato |0⟩.

```
use qubit = Qubit(); (1)
Z(qubit); (2)
DumpRegister((), [qubit]); (3)

Reset(qubit); (4)
```

(1) Come al solito in Q#, iniziamo assegnando un `Qubit` con un'istruzione `use`. Ciò fornisce un nuovo qubit nello stato |0⟩.

(2) Applica un'operazione `Z`, tale che lo stato del qubit viene trasformato in $Z|0\rangle = |0\rangle$

(3) Per confermare che l'operazione `Z` non ha fatto nulla, usiamo la funzione `DumpRegister` per chiedere al simulatore di mostrare il vettore di stato completo.

(4) Ripristina il qubit prima di rilasciarlo. Questo non è strettamente necessario, poiché sappiamo in anticipo che il qubit è ancora nello stato |0⟩.

Nel Listato 9.14, possiamo confermare che l'operazione `Z` non ha fatto nulla utilizzando la funzione `DumpRegister` per chiedere al simulatore di mostrare tutte le informazioni diagnostiche, in questo caso il vettore di stato completo. La Figura 9.9 mostra l'aspetto di questo output diagnostico.

SUGGERIMENTO

Se eseguiamo il codice su una macchina target diversa da un simulatore, non otteniamo un vettore di stato ma la diagnostica offerta dalla macchina (per esempio, degli ID hardware).

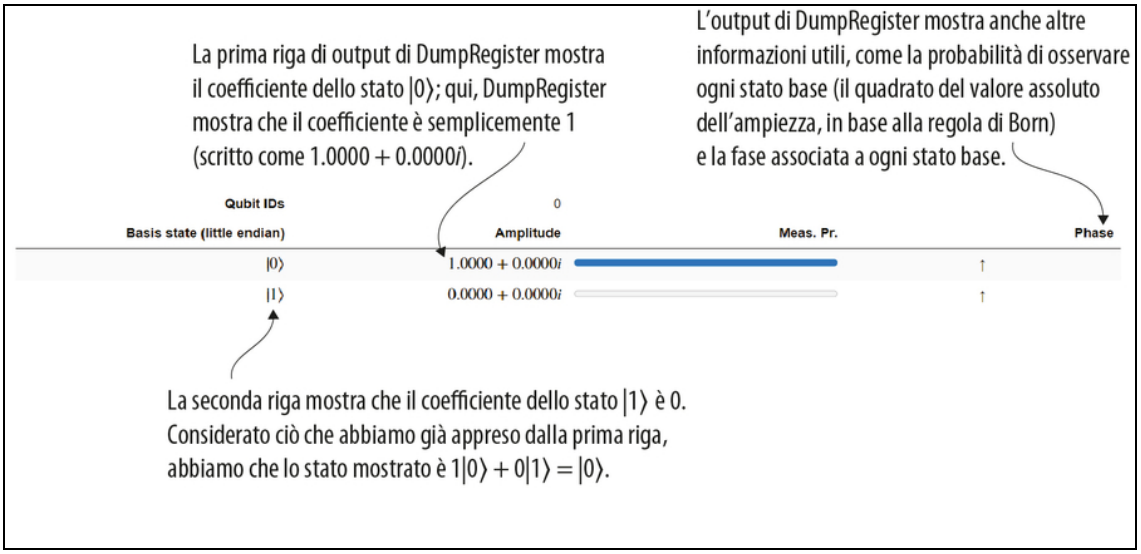


Figura 9.9 Output del Listato 9.14.

Notate che nel Listato 9.14, `z` non fa nulla al qubit, poiché $Z|0\rangle = |0\rangle$. Se modifichiamo il listato per preparare un qubit a $|1$ usando `x` prima di `z`, vediamo qualcosa di molto simile.

Listato 9.15 Applicazione di Z per un qubit nello stato $|1\rangle$.

```
use qubit = Qubit();
X(qubit); (1)
Z(qubit); (2)
DumpRegister("1.txt", [qubit]); (3)

Reset(qubit);
```

(1) Come prima, per preparare uno stato $|1\rangle$, possiamo usare che $|1\rangle = X|0\rangle$.

(2) Ripete il nostro esperimento precedente, ma con un input differente.

(3) Come prima, possiamo scrivere lo stato del qubit in un file di testo, utilizzando il fatto che lo eseguiamo su un simulatore che mantiene internamente il suo stato.

L'output è il seguente:

```
# wave function for qubits with ids (least to most significant): 0
|0⟩: 0.000000 + 0.000000 i ==                          [ 0.000000 ]
```

```
|1⟩: -1.000000 + 0.000000 i == ******************** [ 1.000000 ] --- [
3.14159 rad ] (1)
```

(1) Questo file rappresenta il vettore [[0], [-1]] o -|1⟩ nella notazione di Dirac.

L'effetto di applicare l'operazione `Z` allo stato |1⟩ è quello di invertire il segno dello stato di `qubit`. Questo è un altro esempio di una fase globale, come abbiamo visto nei Capitoli 6 e 8.

Ogni volta che due stati |ψ⟩ e |θ⟩ sono diversi solo per un numero complesso $e^{i}\theta$, |θ⟩ = $e^{i}\theta$|ψ⟩, diciamo che |ψ⟩ e |θ⟩ variano di una *fase globale*. Per esempio, |0⟩ e -|0⟩ differiscono di una fase globale di -1 = e^{i}p.

La fase globale di uno stato non influisce sulle probabilità della misurazione, quindi non potremo mai rilevare se abbiamo applicato un'operazione `Z` quando il suo input è nello stato |0⟩ o |1⟩. Possiamo confermarlo utilizzando l'operazione `AssertQubit`, che verifica la probabilità di un determinato risultato della misurazione.

Listato 9.16 Utilizzo di AssertQubit per controllare il risultato di una misurazione.

```
use qubit = Qubit();
AssertQubit(Zero, qubit); (1)
Z(qubit); (2)
AssertQubit(Zero, qubit); (3)
Message("Passed test!"); (4)

Reset(qubit);
```

(1) Verifica che la misurazione del qubit restituisca il risultato `Zero` e, in caso contrario, interrompi il programma.

(2) Dopo questo, il qubit si trova nello stato -|0⟩ opposto allo stato |0⟩. Cioè, l'operazione `Z` applica una fase globale allo stato del qubit.

(3) Richiama di nuovo `AssertQubit` per verificare che la probabilità di ottenere un risultato `Zero` sia ancora `1`.

(4) Mostra un messaggio per verificare che il programma quantistico superi entrambi gli `Assert`.

L'esecuzione di questo frammento mostra semplicemente `Passed test!`, poiché le chiamate ad `AssertQubit` non fanno nulla nel caso in cui l'asserzione abbia successo. L'utilizzo di asserzioni come questa ci consente di scrivere *unit test* che utilizzano dei simulatori per confermare la nostra comprensione del modo in cui si comportano determinati programmi quantistici. Su un vero hardware quantistico, dal momento che non possiamo eseguire questo tipo di controllo a causa del teorema di non-clonazione, le asserzioni possono essere eliminate in modo sicuro.

IMPORTANTE

Le asserzioni possono essere strumenti davvero utili per scrivere *unit test* e verificare la correttezza dei nostri programmi quantistici. Detto questo, è importante ricordare che verranno eliminate durante l'esecuzione del programma su un vero hardware quantistico, quindi non utilizzate le asserzioni per far funzionare correttamente il programma.Naturalmente, questa è anche una buona pratica di programmazione; le asserzioni nei linguaggi classici, come Python, possono essere disabilitate per motivi prestazionali, pertanto non possiamo contare sul fatto che le asserzioni siano sempre presenti.

Identificare a quali stati quantistici sono assegnate le fasi globali da un'operazione `U` ci dà un modo per comprendere il comportamento di tale operazione quantistica. Chiamiamo tali stati *autostati* dell'operazione `U`. Se due operazioni hanno gli stessi autostati e applicano le stesse fasi globali a ciascuno di quegli autostati, non c'è modo di distinguere queste due operazioni, proprio come se due funzioni classiche avessero la stessa tabella di verità, non possiamo distinguerle, indipendentemente dallo stato in cui si trovano i nostri qubit quando applichiamo l'operazione. Ciò significa che possiamo comprendere le operazioni non solo mediante una loro rappresentazione a matrice, ma anche comprendendo quali sono i loro autostati e quale fase globale si applica a ciascuno di essi. Come abbiamo visto, non possiamo conoscere direttamente la fase globale di

un qubit. Pertanto nel prossimo paragrafo impareremo a usare le versioni controllate di un'operazione, per trasformare tale fase globale in una fase locale, che possiamo misurare. Per ora, però, cerchiamo di riassumere con una definizione più formale il significato di un autostato.

Se, dopo aver applicato un'operazione `U`, lo stato di un registro di qubit, `qs`, viene modificato solo da una fase globale, diciamo che lo stato di quel registro è un *autostato* di `U`. Per esempio, |0⟩ e |1⟩ sono entrambi autostati dell'operazione `Z`. Allo stesso modo, |+⟩ e |-⟩ sono entrambi autostati di `X`.

Il prossimo esercizio vi aiuterà a esercitarvi con gli autostati.

ESERCIZIO 9.3 TECNICHE DIAGNOSTICHE

Provate a scrivere dei programmi Q# che utilizzano `AssertQubit` e `DumpMachine` per verificare che:

- +⟩ e |-⟩ sono entrambi autostati dell'operazione `X`;

- |0⟩ e |1⟩ sono entrambi autostati dell'operazione `Rz`, indipendentemente dall'angolo di rotazione scelto.

Per fare ancora più pratica, provate a immaginare gli autostati delle operazioni `Y` e `CNOT` e scrivete un programma Q# per verificare le vostre ipotesi.

Suggerimento: potete trovare la forma vettoriale degli autostati di un'operazione unitaria impiegando QuTiP. Per esempio, gli autostati dell'operazione `Y` sono dati da `qt.sigmay().eigenstates()`. Da lì, potete usare ciò che avete imparato sulle rotazioni nei Capitoli da 4 a 6 per determinare quali operazioni Q# preparano questi stati.

Non dimenticate: potete sempre verificare se un determinato stato è un autostato di un'operazione scrivendo un test rapido in Q#.

Gli autostati sono un concetto molto utile e possono essere utilizzati in vari algoritmi di quantum computing. Li useremo nel prossimo paragrafo insieme alle operazioni controllate per implementare una tecnica di sviluppo quantistico chiamata *phase kickback*, presentata alla fine del Capitolo 8.

Approfondimento: autostati, autovalori e autovettori

Gli autostati traggono il loro nome da un concetto utilizzato in tutta l'algebra lineare: gli *autovettori*. Proprio come un autostato è uno stato che non viene modificato da un'operazione quantistica (cioè, al massimo viene applicata una fase globale), un autovettore è un vettore che viene preservato fino a un certo fattore di scala quando viene moltiplicato per una matrice. Cioè, se per una matrice A, $A\vec{x} = \lambda\vec{x}$ per un certo numero λ, allora $\vec{x}$ è un autovettore di A. Diciamo anche che λ è l'autovalore corrispondente.

Gli autovettori e gli autovalori ci aiutano a comprendere le proprietà o le caratteristiche delle matrici. In particolare, se una matrice *A* commuta con la sua coniugata trasposta (cioè se $AA\dagger = A\dagger A$), allora può essere *scomposta* in proiettori sugli autovettori, ciascuno scalato dai suoi autovalori:

$$A = \sum_i \lambda_i \vec{x}_i \vec{x}_i^{\dagger}$$

Poiché questa condizione vale sempre per le matrici unitarie, possiamo sempre comprendere le operazioni quantistiche scomponendole nei loro autostati e nelle fasi applicate a ciascun autostato. Per esempio, $Z = |0\rangle\langle 0| - |1\rangle\langle 1|$ e $X = |+\rangle\langle +| - |-\rangle\langle -|$.

Una conseguenza significativa della scomposizione delle matrici in questo modo è che se due matrici *A* e *B* hanno gli stessi autovettori e autovalori, sono la stessa matrice. Allo stesso modo, se due operazioni possono essere rappresentate dagli stessi autostati e autofasi, le operazioni sono indistinguibili l'una dall'altra.

Questo modo di considerare stati e operazioni può aiutarci a trovare il senso di diversi concetti di quantum computing. Un altro modo per pensare a questo gioco di stima della fase è considerarlo un algoritmo per l'apprendimento delle fasi associate a ciascun autostato. Nel Capitolo 10, vedrete che questo si ricollega particolarmente bene ad alcune applicazioni, come l'apprendimento delle proprietà dei sistemi chimici.

Applicazione controllata: trasformazione di fasi globali in fasi

locali

Da quanto abbiamo visto e possiamo verificare, le *fasi globali* degli stati non sono osservabili, mentre le *fasi locali* degli stati possono essere misurate. Per esempio, considerate lo stato 1 / √2 (-*i*|0⟩ -*i*|1⟩) = -*i* / √2 (|0⟩ + |1⟩). Non c'è misura che possiamo fare per distinguere tale stato da (|0⟩ + |1⟩) / √2. Tuttavia, potremmo distinguere entrambi questi stati da (|0⟩ - |1⟩) / √2, in quanto differiscono per una *fase locale*; infatti, uno degli stati |1⟩ è preceduto da un + e l'altro da un -.

> **SUGGERIMENTO**
>
> Per un ripasso sulle fasi e su come immaginarle come rotazioni, consultate i Capitoli 4 e 5. Quando si utilizza come macchina target un simulatore, l'output di `DumpMachine` e `DumpRegister` può aiutarvi a conoscere le fasi degli stati.

Nel paragrafo precedente, abbiamo giocherellato con gli autostati e abbiamo visto che le fasi globali degli autostati possono trasportare informazioni su un'operazione: per esempio l'operazione `U`. Se vogliamo conoscere questa informazione della fase globale sugli autostati, allora sembra che stiamo bloccati. Se Lancillotto dovesse preparare solo gli autostati dell'operazione di Dagonet, non sarebbe mai in grado di scoprire l'angolo segreto di Dagonet.

Gli algoritmi quantistici vengono in nostro soccorso! C'è un trucco molto utile che possiamo applicare per trasformare le *fasi globali* applicate da un'operazione `U` in *fasi locali* applicate da un'operazione strettamente correlata. Per vedere come funziona, torniamo all'operazione `CNOT`. Ricordiamo, dal Capitolo 6, che possiamo simulare `CNOT` usando una matrice unitaria:

$$U_{\text{CNOT}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Quando abbiamo incontrato per la prima volta questa matrice nel Capitolo 6, abbiamo usato l'analogia tra matrici unitarie e tabelle di verità classiche per capire che l'operazione `CNOT` inverte gli stati $|10\rangle$ e $|11\rangle$ ma lascia immutati i qubit negli stati $|00\rangle$ e $|01\rangle$. Cioè, `CNOT` inverte lo stato del secondo qubit, in modo *controllato* dallo stato del primo qubit. Come vediamo nella Figura 9.10, possiamo leggere la matrice unitaria per l'operazione `CNOT` come se descrivesse una sorta di istruzione di "if quantistico": "*Se* il qubit `control` è nello stato $|1\rangle$, *allora* applica l'operazione `X` al qubit `target`".

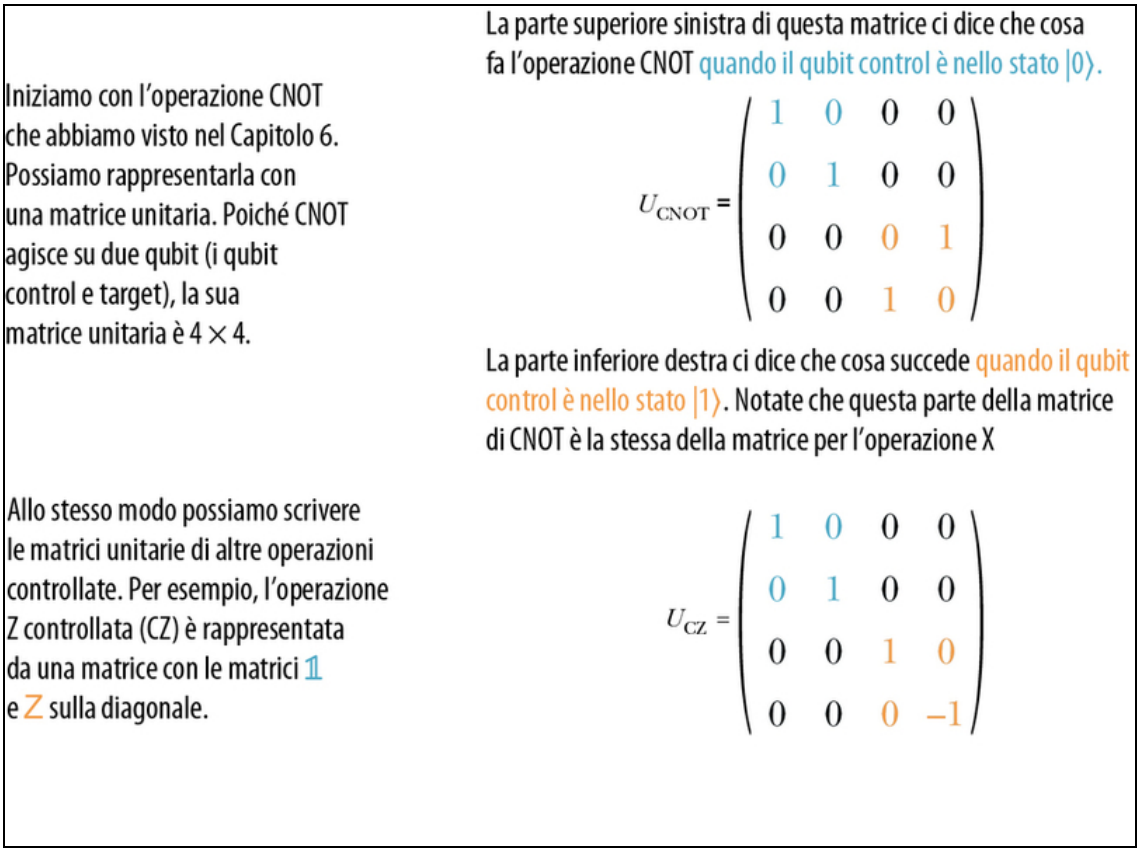


Figura 9.10 Scrittura di matrici unitarie per le operazioni controllate.

Per utilizzare l'operazione `CNOT` in Q#, possiamo provare il codice seguente.

Listato 9.17 Utilizzo dell'operazione CNOT in Q#.

```
use control = Qubit();
use target = Qubit();
H(control); (1)
X(target); (2)
CNOT(control, target); (3)
DumpMachine();
Reset(control);

Reset(target);
```

(1) Prepara il qubit `control` in $|+\rangle$.

(2) Prepara il qubit `target` in $|1\rangle$.

(3) Applica `CNOT` e mostra lo stato del simulatore.

Considerando `CNOT` come un analogo quantistico di un'istruzione condizionale, possiamo scrivere la sua matrice unitaria un po' più direttamente. In particolare, possiamo considerare la matrice unitaria per l'operazione `CNOT` come una sorta di "matrice a blocchi", che possiamo costruire utilizzando il prodotto tensoriale che abbiamo visto nel Capitolo 4:

$$U_{\text{CNOT}} = \begin{pmatrix} \mathbb{1} & 0 \\ 0 & X \end{pmatrix} = |0\rangle\langle 0| \otimes \mathbb{1} + |1\rangle\langle 1| \otimes X$$

ESERCIZIO 9.4 VERIFICA DELLA MATRICE DI CNOT

Verificate che $|0\rangle\langle 0| \otimes + |1\rangle\langle 1| \otimes X$ è la stessa cosa dell'equazione precedente.

Suggerimento: potete verificarlo a mano utilizzando la funzione `np.kron` di NumPy o la funzione `qt.tensor` di QuTiP. Se avete bisogno di un ripasso, controllate come avete simulato il teletrasporto nel Capitolo 6, o riguardate la derivazione dell'algoritmo di Deutsch-Jozsa nel Capitolo 8.

Seguendo questo modello possiamo costruire anche altre operazioni, come l'operazione `CZ` (`Z` controllata):

$$U_{\text{CZ}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix} = |00\rangle\langle 00| + |01\rangle\langle 10| + |10\rangle\langle 10| - |11\rangle\langle 11|$$

Proprio come l'operazione `CNOT` fa la stessa cosa di un'operazione `X` (inverte un bit), ma sotto il controllo dello stato di un altro qubit, quando il qubit `control` è nello stato $|1\rangle$, l'operazione `CZ` inverte una fase come l'operazione *Z*. La Figura 9.10 mostra un esempio di applicazione. Vediamo come funziona in pratica il controllo di `Z` scrivendo alcune righe di Q# per provare `CZ`.

Listato 9.18 Test dell'operazione CZ in Q#.

```
use control = Qubit();
use target = Qubit();
H(control); (1)
X(target); (2)
CZ(control, target); (3)
DumpRegister("cz-output.txt", [control, target]);
Reset(control);

Reset(target);
```

(1) Prepara il qubit `control` in $|+\rangle$.

(2) Prepara il qubit `target` in $|1\rangle$.

(3) Applica `CZ` e salva lo stato risultante.

L'output è il seguente:

```
# wave function for qubits with ids (least to most significant): 0;1
|0⟩:  0.000000 + 0.000000 i ==             [ 0.000000 ]
|1⟩:  0.000000 + 0.000000 i ==             [ 0.000000 ]
|2⟩:  0.707107 + 0.000000 i == ********** [ 0.500000 ] --- [ 0.00000
rad ]

|3⟩: -0.707107 + 0.000000 i == ********** [ 0.500000 ] --- [ 3.14159
rad ]
```

Provando l'esecuzione, il contenuto di `cz-output.txt` mostrerà che lo stato finale del `[control, target]` è $U\text{CZ}|+1\rangle = |-1\rangle$.

ESERCIZIO 9.5 VERIFICA DELL'OUTPUT DI CZ

A mano o impiegando QuTiP, verificate che l'output precedente sia lo stesso di $|-1\rangle = |-\rangle \otimes |1\rangle$.

Se l'ordine dei qubit viene scambiato ma, ciononostante, la risposta è corretta, notate che `DumpMachine` utilizza una rappresentazione *little-endian* per ordinare gli stati. In little endian, $|2\rangle$ è l'abbreviazione di $|01\rangle$, non di $|10\rangle$. Se tutto ciò vi sembra confuso, prendetevela con l'architettura del processore x86.

Cioè, in base allo stato di `target`, è cambiato lo stato di `control`, proprio come abbiamo visto nel Capitolo 8 con l'algoritmo di Deutsch-Jozsa. Questo perché la fase applicata da `Z` nel caso in cui `control` sia nello stato $|0\rangle$ non è la stessa di quando `control` è nello stato $|1\rangle$, un effetto noto come *phase kickback* ("contraccolpo di fase"). Nel

Capitolo 8, abbiamo usato questo effetto con un paio di qubit nello stato |+-⟩ per dire se era stata applicata l'operazione `CNOT`. Qui, abbiamo visto che possiamo usare l'operazione `CZ` per conoscere la fase globale applicata dall'operazione `Z`.

> **IMPORTANTE**
>
> Anche se |1⟩ è un autostato dell'operazione `Z`, |+1⟩ non è un autostato dell'operazione `CZ`. Ciò significa che richiamando `CZ` su un registro nello stato |+1⟩ si ha un effetto osservabile.

Phase kickback è una tecnica di programmazione quantistica molto comune, poiché ci consente di trasformare quelle che altrimenti sarebbero fasi globali in una fase tra i rami |0⟩ e |1⟩ del qubit `control`. Nell'esempio di `CZ`, sia lo stato di input |+⟩|1⟩ sia lo stato di output |-⟩|1⟩ sono stati prodotti, cosa che ci permette di misurare il qubit `control` senza influenzare il qubit `target`.

> **Pensate in modo globale, scoprite le fasi in modo locale**
>
> Notate che una differenza di fase globale tra |1⟩ e Z|1⟩ = -|1⟩ è diventata una differenza di fase locale fra |1⟩ e UCZ|+1⟩ = |-1⟩. Cioè, controllando l'istruzione `Z` su un qubit nello stato |+⟩, siamo stati in grado di apprendere quella che sarebbe stata una fase globale, senza `control`.

Utilizzando l'operazione `CZ`, possiamo implementare la tecnica del *phase kickback* per trasformare una fase globale in una fase locale, che quindi possiamo misurare.

Listato 9.19 Utilizzo di CZ per implementare il phase kickback.

```
use control = Qubit();
use target = Qubit();
H(control); (1)
X(target);
CZ(control, target); (2)
if (M(control) == One) { X(control); } (3)
DumpRegister("cz-target-only.txt", [target]); (4)

Reset(target); (5)
```

(1) Prepara il qubit `control` in |+⟩ e il qubit `target` in |1⟩.

(2) Applica `CZ` e salva lo stato risultante. Ma prima di scaricare lo stato del qubit `target`, cerchiamo di misurare e ripristinare il qubit `control`.

(3) Una cosa divertente: questo è il modo in cui l'operazione di Reset è implementata in Q#.

(4) Ora scarichiamo solo lo stato del target.

(5) Abbiamo già eseguito il reset di `control`, quindi non è necessario eseguirne di nuovo il reset.

Ecco l'output:

```
# wave function for qubits with ids (least to most significant): 1
|0⟩:  0.000000 + 0.000000 i ==                      [ 0.000000 ]

|1⟩: -1.000000 + 0.000000 i == ******************** [ 1.000000 ] --- [
3.14159 rad ] (1)
```

(1) Come previsto, otteniamo che il qubit `target` rimane nello stato |1⟩, pronto per essere inviato in un'altra operazione `CZ`.

Controllo di qualsiasi operazione

Ripensando al gioco di Lancillotto e Dagonet, sarebbe davvero utile se potessimo aiutare Lancillotto a riutilizzare il qubit che egli passa nell'operazione di Dagonet, in modo che non debba prepararlo ogni volta. Fortunatamente, l'utilizzo di operazioni controllate per implementare il *phase kickback* fornisce un modo per farlo. In particolare, quando abbiamo usato il phase kickback nei Capitoli 7 e 8 per implementare l'algoritmo di Deutsch-Jozsa, il qubit `target` era nello stato |-⟩ sia all'inizio sia alla fine dell'algoritmo. Ciò significa che Lancillotto potrebbe riutilizzare lo stesso qubit per ogni round del suo gioco e non ha bisogno di prepararlo nuovamente ogni volta. Questo non ha avuto importanza per Deutsch-Jozsa, dal momento che abbiamo giocato solo un round della partita fra Viviana e Merlino. Ma è esattamente il trucco di cui Lancillotto ha bisogno per vincere la sua

partita con Dagonet, quindi vediamo come possiamo aiutarlo a usare il *phase kickback*.

Il problema è che, mentre il *phase kickback* è uno strumento utile da avere nella nostra cassetta degli attrezzi di sviluppatori quantistici, finora abbiamo visto solo come usarlo con le operazioni `X` e `Z`. Sappiamo che, per il nostro gioco, Dagonet ha detto a Lancillotto che userà l'operazione `R1`; c'è un modo in cui possiamo usare il *phase kickback* per aiutarlo? Il modello che abbiamo usato per implementare il *phase kickback* nel paragrafo precedente ci richiedeva solo di controllare un'operazione, quindi ciò di cui abbiamo bisogno è un modo per controllare l'operazione `op::Apply` che Dagonet dà a Lancillotto. In Q#, questo è semplice come scrivere `Controlled op::Apply` invece di `op::Apply`, grazie al funtore `Controlled`. Proprio come il funtore `Adjoint` del Capitolo 6, `Controlled` è una parola riservata del linguaggio Q# che modifica il comportamento di un'operazione: in questo caso, per trasformarla nella sua versione controllata.

> **SUGGERIMENTO**
>
> Proprio come `Adj` indica che un'operazione può essere usata con `Adjoint`, è `Ctl` nel tipo di operazione indica che può essere usata con il funtore `Controlled`. Per denotare che un'operazione supporta entrambi, possiamo scrivere `is Adj + Ctl`. Per esempio, il tipo dell'operazione `X` è `(Qubit => Unit is Adj + Ctl)`, cosa che ci fa sapere che `X` è sia `Adjoint` sia `Controlled`.

Quindi, per aiutare Lancillotto, possiamo cambiare la riga `op::Apply(scale, target)` in `Controlled op::Apply([control], (scale, target))`, e così abbiamo la versione `Controlled` di `R1`.

Sebbene ciò risolva il problema di Lancillotto, può essere utile scoprire un po' di più che cosa sta accadendo "sotto il cofano". Qualsiasi operazione unitaria (ovvero, un'operazione quantistica che non alloca, dealloca o misura un qubit) può essere controllata, nello

stesso modo in cui abbiamo controllato l'operazione `Z` per ottenere `CZ` e come abbiamo controllato `X` per ottenere `CNOT`. Per esempio, possiamo definire un'operazione *Controlled-Controlled-NOT* (`CCNOT`, detta anche Toffoli) come un'operazione che prende due qubit `control` e inverte il suo `target` se *entrambi* i `control` sono nello stato $|1\rangle$. Matematicamente, scriviamo che l'operazione `CCNOT` trasforma uno stato di input $|x\rangle|y\rangle|z\rangle$ nell'output $|x\rangle|y\rangle|z \text{ XOR } (y \text{ AND } z)\rangle$. Possiamo anche scrivere una matrice che ci permetta di simulare l'operazione `CCNOT`:

$$U_{\text{CCNOT}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Allo stesso modo, l'operazione Controlled-SWAP (nota anche come operazione di Fredkin) trasforma i suoi stati di input da $|1\rangle|y\rangle|z\rangle$ in $|1\rangle|z\rangle|y\rangle$ e lascia immutato il suo input quando il primo qubit è nello stato $|0\rangle$.

SUGGERIMENTO

Possiamo creare una Controlled-SWAP con tre operazioni `CCNOT`: `CCNOT(a, b, c); CCNOT(a, c, b); CCNOT(a, b, c);` è equivalente a `Controlled SWAP([a], (b, c));`. Per vederlo, notate che possiamo anche eseguire l'operazione `SWAP` non controllata con tre operazioni `CNOT`, per lo stesso motivo per il quale possiamo invertire due registri classici in loco usando una sequenza di tre operazioni XOR classiche.

Possiamo generalizzare questo modello per qualsiasi operazione unitaria `U` (ovvero, qualsiasi operazione che non alloca, dealloca o misura i suoi qubit). In Q#, la trasformazione eseguita utilizzando il funtore `Controlled` aggiunge un nuovo input a un'operazione che rappresenta i qubit da utilizzare come controlli.

SUGGERIMENTO

Qui è molto utile il fatto che Q# sia un linguaggio che agisce su tuple e restituisce tuple. Poiché ogni operazione richiede esattamente un input, per qualsiasi operazione `U`, `Controlled U` prende l'input originale di `U` come suo secondo input.

Le operazioni `CNOT` e `CZ` sono semplici scorciatoie per le chiamate appropriate a `Controlled`. La Tabella 9.1 mostra altri esempi di questo modello.

SUGGERIMENTO

Proprio come `Adjoint` funziona su qualsiasi operazione che ha `is Adj` nel suo tipo (come abbiamo visto nel Capitolo 8), il funtore `Controlled` funziona su qualsiasi operazione che ha `is Ctl` nel suo tipo.

Tabella 9.1 Alcuni esempi di operazioni controllate in Q#.

Descrizione	Abbreviazione	Definizione
Controlled-Not	`CNOT(control, target)`	`Controlled X([control], target)`
Controlled-Controlled-NOT (Toffoli)	`CCNOT(control0, control1, target)`	`Controlled X ([control0, control1], target)`
Controlled-SWAP (Fredkin)	Non disponibile	`Controlled SWAP([control], (target1, target2))`
Controlled-y	`CY(control, target)`	`Controlled Y([control], target)`

Controlled-PHASE	`CZ(control, target)`	`Controlled Z([control], target)`

Come abbiamo visto con l'esempio di `CZ`, questo controllo delle operazioni ci consente di trasformare le fasi globali, come quelle applicate agli autostati, in fasi relative, che possiamo scoprire attraverso misurazioni.

Inoltre, utilizzando rotazioni controllate per riportare indietro la fase (*phase kickback*) sul registro `control`, possiamo anche riutilizzare lo stesso qubit `target` più e più volte. Quando abbiamo applicato `CZ` a un registro `target` in un autostato di `Z`, quel registro `target` è rimasto nello stesso stato, anche se il registro `control` è cambiato. Nel resto di questo capitolo, vedremo come usare questo fatto per terminare la strategia di Lancillotto per il suo gioco con Dagonet.

Implementazione della migliore strategia di Lancillotto per il gioco della stima della fase

Ora abbiamo tutto ciò di cui abbiamo bisogno per scrivere una strategia leggermente diversa per Lancillotto, che gli consentirà di utilizzare operazioni controllate per riutilizzare gli stessi qubit. Come abbiamo notato, questo potrebbe non avere un impatto enorme per il gioco di Dagonet, ma certamente lo avrà per altre applicazioni del quantum computing.

Per esempio, nel Capitolo 10, vedremo come i problemi di chimica quantistica possono essere risolti utilizzando un gioco molto simile a quello di Dagonet e Lancillotto. Lì, tuttavia, la preparazione del giusto stato di input può richiedere la chiamata di molte operazioni

quantistiche, e così se possiamo preservare il qubit `target` per un riutilizzo successivo, possiamo anche ottenere un notevole vantaggio prestazionale.

Esaminiamo brevemente i passaggi del gioco.

1. Dagonet sceglie un angolo segreto per un'operazione di rotazione mono-qubit.
2. Dagonet prepara un'operazione che Lancillotto potrà usare, la quale nasconde l'angolo segreto e consente a Lancillotto di specificare un input aggiuntivo di un numero (che chiameremo scala) del quale verrà moltiplicato l'angolo segreto, per ottenere l'angolo di rotazione totale dell'operazione.
3. La migliore strategia di Lancillotto per il gioco è quella di selezionare molti valori di scala e stimare la probabilità di misurare `One` per ogni valore.

   - Prepara lo stato $|+\rangle$ e inserisce il valore della scala nella rotazione di Dagonet. Usa lo stato $|+\rangle$ perché sa che Dagonet esegue rotazioni attorno all'asse Z, e per questo stato, tali rotazioni produrranno un cambiamento di fase locale che egli può misurare.

   - Dopo aver preparato ogni stato $|+\rangle$, Lancillotto può ruotarlo con l'operazione segreta, misurare il qubit e registrare la misurazione.
4. Lancillotto ora dispone dei dati relativi al suo fattore di scala e alla probabilità di aver misurato `One` per quel fattore di scala. Può così ragionare su questi dati e ottenere l'angolo di Dagonet dai parametri adattati (non per nulla è il più grande cavaliere del regno). Possiamo usare Python per aiutarci a fare lo stesso.

Il passo che deve cambiare per utilizzare le nostre nuove abilità con le rotazioni *controllate* è il Passo 3. Per il Passo 3a, cambierà l'*allocazione* dei qubit. Invece di allocare, preparare e misurare un

qubit per ogni misurazione, Lancillotto può allocare un qubit `target` da ruotare con la scatola nera di Dagonet e poi allocare e misurare i qubit `control`. Può anche ripetere le misurazioni, ma non dovrà misurare o preparare nuovamente ogni volta il `target`.

Potremmo riassumere queste modifiche riscrivendo il Passo 3 in questo modo.

3. La migliore strategia di Lancillotto per il gioco è quella di selezionare molti valori di scala e stimare la probabilità di misurare `One` per ogni valore. Per fare ciò, deve eseguire le seguenti operazioni molte volte per ciascuno dei molti valori di scala. Egli prepara un qubit nello stato $|1\rangle$ da utilizzare come `target` per tutte le sue misurazioni, in quanto è un autostato della rotazione nascosta.

    1. Prepara un secondo qubit `control` nello stato $|+\rangle$.
    2. Applica la nuova versione controllata della rotazione segreta con il valore di scala di Lancillotto, de-prepara il qubit `control` e lo misurar, quindi registra la misurazione.

Nel nostro codice, queste modifiche possono essere eseguite modificando la precedente operazione `EstimateProbabilityAtScale`. Poiché l'asse di rotazione potrebbe essere qualsiasi cosa scelga Dagonet (qui è l'asse *Z*, per comodità), Lancillotto deve sapere come controllare una rotazione arbitraria. Possiamo farlo con il funtore `Controlled` prima di chiamare la `ScalableOperation` passata da Dagonet. Il funtore `Controlled` è molto simile al funtore `Adjoint`, in quanto accetta un'operazione e restituisce una nuova operazione. `Controlled U(control, target)` è un esempio della sintassi che ci permette di applicare `U` al nostro qubit `target`, controllato su uno o più qubit `control`. Il listato seguente mostra come modificare `EstimateProbabilityAtScale` per utilizzare il funtore `Controlled`.

Listato 9.20 operations.qs: la nuova strategia di Lancillotto.

```
operation EstimateProbabilityAtScaleUsingControlledRotations(
    target : Qubit,
    scale : Double,
    nMeasurements : Int,
    op : ScalableOperation)
: Double {
    mutable nOnes = 0;
    for idx in 0..nMeasurements - 1 {
        use control = Qubit();
        within {
            H(control); (1)
        } apply {
            Controlled op::Apply( (2)
                 [control],
                 (scale, target)
            );
        }
        set nOnes += Meas.MResetZ(control) == One ? 1 | 0;
    }
    return Convert.IntAsDouble(nOnes) /
Convert.IntAsDouble(nMeasurements);

}
```

(1) L'operazione per indovinare ora prende come input il registro `target` e lo riutilizza. Pertanto, dobbiamo solo allocare e preparare il registro `control`.

(2) L'unico altro cambiamento che dobbiamo fare è richiamare `Controlled op::Apply` invece di `op::Apply`, passando il nuovo qubit `control` insieme agli input originali.

L'altra modifica che dobbiamo apportare (Passo 5) è l'operazione che esegue il gioco. Poiché l'utilizzo dell'operazione controllata consente a Lancillotto di riutilizzare il qubit `target`, è necessario assegnarlo una sola volta all'inizio del gioco. Il prossimo listato mostra come implementarlo.

Listato 9.21 operations.qs: implementazione di RunGameUsingControlledRotations.

```
operation RunGameUsingControlledRotations(
    hiddenAngle : Double,
    scales : Double[],
    nMeasurementsPerScale : Int)
```

```
: Double[] {
    let hiddenRotation = HiddenRotation(hiddenAngle);
    use target = Qubit(); (1)
    X(target); (2)
    let measurements = Arrays.ForEach(
        EstimateProbabilityAtScaleUsingControlledRotations(
            target, _, nMeasurementsPerScale, hiddenRotation
        ),
        scales
    );
    X(target);
    return measurements;

}
```

(1) Usando `EstimateProbabilityAtScaleUsingControlledRotations`, possiamo allocare il qubit `target` una sola volta, poiché lo usiamo più e più volte per ogni tentativo.
(2) Usando l'operazione `X`, possiamo preparare il `target` nello stato |1⟩, un autostato dell'operazione `R1` (non controllata) in cui Dagonet ha nascosto il suo angolo.

Usando l'operazione `X` come nel Listato 9.21, possiamo preparare `target` nello stato |1⟩, un autostato dell'operazione `R1` (non controllata) in cui Dagonet ha nascosto il suo angolo. Poiché ogni misurazione utilizza il *phase kickback* per influenzare solo il registro `control`, questa preparazione può essere eseguita una solo volta, prima del gioco.

Riepilogo

- La stima della fase è un algoritmo quantistico che consente di apprendere la fase applicata a un registro di qubit da una determinata operazione.
- In Q#, possiamo dichiarare nuovi tipi definiti dall'utente per chiarire il modo in cui un dato tipo deve essere utilizzato in un programma quantistico o fornire una scorciatoia per tipi troppo lunghi.

- I programmi quantistici in Q# possono essere eseguiti da soli o da un programma host scritto in Python; ciò consente di utilizzare programmi Q# insieme ad altri strumenti di scienza dei dati, come SciPy.
- Quando un'operazione lascia non modificati gli input in un determinato stato, tranne che per l'applicazione di una fase globale, diciamo che tale stato di input è un *autostato* e la fase corrispondente è un'*autofase*.
- Usando insieme il funtore `Controlled` e il *phase kickback*, possiamo trasformare le autofasi globali in fasi locali che possiamo osservare e valutare.
- Mettendo tutto insieme, possiamo usare le classiche tecniche di adattamento dei dati per apprendere le *autofasi* dalle misurazioni restituite da un programma Q# che svolge la stima della fase.

Parte II: conclusioni

In questa parte del libro, ci siamo divertiti molto a usare Q# e il quantum computing per aiutare i vari abitanti di Camelot. Usando un generatore quantistico di numeri casuali scritto in Q#, siamo stati in grado di aiutare Morgana a giocare un tiro al povero Lancillotto. Allo stesso tempo, abbiamo aiutato Merlino e Viviana a svolgere i rispettivi compiti nel decidere il destino del re, imparando nel contempo l'algoritmo di Deutsch-Jozsa e il *phase kickback*. Ora che il regno è in pace e i camini sono accesi al castello di Camelot, abbiamo visto come usare tutto ciò che abbiamo imparato per aiutare Lancillotto a risolvere un altro gioco, questa volta indovinando un'operazione quantistica nascosta da Dagonet.

In tutte queste nostre visite a Camelot, avete raccolto alcuni nuovi trucchi per aiutarvi nel vostro cammino di sviluppatori quantistici. Ora

conoscete molti concetti.

- Che cos'è un algoritmo quantistico e come implementarlo con il Quantum Development Kit e Q#.
- Come usare Q# da Python e Jupyter Notebook.
- Come progettare oracoli per rappresentare le funzioni classiche nei programmi quantistici.
- I tipi definiti dall'utente.
- Le operazioni controllate.
- La tecnica *phase kickback*.

È tempo di mettere a profitto tutto ciò che avete imparato a Camelot e applicare queste nuove tecniche a qualcosa di un po' più pratico. Nel prossimo capitolo scoprirete come il quantum computing può aiutare a comprendere dei problemi di chimica. Non preoccupatevi se non vi ricordate la tavola periodica; lavorerete con alcuni colleghi che conoscono il lato chimico delle cose e stanno solo cercando il vostro aiuto per utilizzare tutto ciò che avete imparato in questa parte del libro per aggiornare il loro flusso di lavoro con la tecnologia quantistica.

Quantum computing applicato

Giunti a questo punto del libro, abbiamo costruito un'interessante "cassetta degli attrezzi" di tecniche algoritmiche quantistiche; in questa parte vedremo come applicare tali tecniche a diversi problemi pratici. In particolare, implementeremo ed eseguiremo piccoli esempi di tre diversi programmi quantistici, ciascuno dei quali si rivolge a un'area in cui può essere applicato con profitto il quantum computing. Questi esempi sono abbastanza compatti da poter essere simulati con computer classici, ma dimostrano come i dispositivi quantistici possano fornire vantaggi computazionali anche per problemi di interesse pratico.

Nel Capitolo 10, useremo le nostre nuove capacità di programmazione quantistica per implementare un algoritmo che aiuta a risolvere complessi problemi di chimica. Poi lo svilupperemo nel Capitolo 11 per implementare un algoritmo di ricerca su dati non strutturati; impareremo ad applicare le funzionalità integrate del linguaggio Q# e del QDK per stimare le risorse necessarie per eseguire un'applicazione quantistica su larga scala. Infine, nel Capitolo 12, implementeremo l'algoritmo di Shor per la fattorizzazione degli interi, forse uno degli algoritmi quantistici più famosi, grazie alle sue applicazioni nella crittografia classica.

Soluzione di problemi di chimica con computer quantistici

Nel Capitolo 9, abbiamo utilizzato una serie di nuove funzionalità del linguaggio Q#, come i tipi definiti dall'utente e l'esecuzione di programmi da un host Python e abbiamo scritto un programma quantistico in grado di stimare le fasi. Come vedremo in questo capitolo, la stima della fase è comunemente usata negli algoritmi quantistici per costruire programmi più grandi e complessi. In questo capitolo, considereremo la nostra prima area di applicazione pratica: la chimica.

Applicazioni di chimica per il quantum computing

Finora, in questo libro, abbiamo imparato a utilizzare i dispositivi quantistici per fare di tutto, da una chat con la nostra amica Eve fino ad aiutare a decidere il destino del regno. In questo capitolo avremo invece la possibilità di fare qualcosa di un po' più *pratico*.

NOTA

Ora che abbiamo ciò di cui abbiamo bisogno per risolvere problemi più complessi con i computer quantistici, lo scenario di questo capitolo si fa un po' più complicato della maggior parte dei giochi e degli scenari precedenti. Non preoccupatevi se le cose non saranno chiarissime, all'inizio. Prendetevi il

vostro tempo e leggete le frasi più lentamente; vi promettiamo che ne varrà la pena.

A quanto pare, la nostra amica chimica Marie ha raggiunto il limite di ciò che il suo computer classico può fare per aiutarla a modellare diversi sistemi chimici. I problemi che Marie risolve con le tecniche di chimica computazionale possono contribuire a contrastare i cambiamenti climatici, a conoscere le caratteristiche di nuovi materiali e a migliorare l'utilizzo dell'energia in tutti i settori; se possiamo aiutarla usando il linguaggio Q#, ciò potrebbe avere molte e importanti applicazioni pratiche. Fortunatamente, utilizzando quello che abbiamo imparato sulla stima delle fasi nel Capitolo 9, siamo in grado di fare proprio questo.

C'è più gusto con la chimica

Qualsiasi produttore di caramelle può parlarvi della differenza che può fare la temperatura: cuocete lo zucchero fino alla fase di "crack morbido" e otterrete il taffy; ma se aggiungete un altro po' di energia, potete creare molti altri deliziosi composti, dal toffee al caramello. Tutto ciò che riguarda lo zucchero, il gusto, l'aspetto e la consistenza, cambia a seconda dell'energia che gli diamo, sul fuoco. In misura non trascurabile, se comprendiamo come cambia la forma delle molecole di zucchero quando aggiungiamo un po' di energia al "dolce crogiolo", comprendiamo lo zucchero stesso.

Vediamo questo effetto non solo con le caramelle ma un po' in tutti gli aspetti la vita. Acqua, vapore e ghiaccio si differenziano in base alla comprensione delle forme che l'H2O può assumere, ovvero in quali forme può organizzarsi, in funzione dell'energia. In molti casi, vogliamo capire come si dispone la molecola in funzione dell'energia, basandoci su simulazioni e non su esperimenti. In questo capitolo svilupperemo le tecniche apprese nei capitoli precedenti, mostrando come possiamo simulare l'energia dei sistemi chimici, in modo da comprenderli quasi come un produttore di caramelle conosce il proprio mestiere, per utilizzare tali sistemi chimici per rendere migliore, e forse anche un po' più dolce, la nostra vita.

Per avere un'idea di come funzionano le cose, concordiamo con Marie che inizieremo a osservare l'*idrogeno molecolare*, H2, poiché è

un sistema chimico abbastanza semplice, che ci permette di confrontare ciò che apprendiamo dal nostro programma quantistico con ciò che può essere simulato con gli strumenti di modellazione classici. In questo modo, poiché poi utilizzeremo le stesse tecniche per studiare molecole più grandi di quelle che possono essere simulate in modo classico, avremo un ottimo caso di test su cui possiamo contare per assicurarci che tutto proceda nel modo corretto.

Una simulazione all'interno di un'altra simulazione

In questo capitolo, il nostro lavoro con Marie prevede due diversi tipi di simulazioni: utilizziamo un computer classico per simulare un computer quantistico e utilizziamo un computer quantistico per simulare un diverso tipo di sistema quantistico. Spesso vogliamo fare entrambe le cose, nel senso che è utile, per la creazione di applicazioni di chimica quantistica, usare un computer classico per simulare il modo in cui un computer quantistico simula un sistema chimico quantistico. In questo modo, quando eseguiremo la nostra simulazione quantistica su vero hardware quantistico, possiamo essere certi che funzionerà correttamente.

Come vediamo nella Figura 10.1, Marie inizierà usando la sua esperienza in chimica quantistica per descrivere un problema che è interessata a risolvere con un computer quantistico: in questo caso, comprendere la struttura di H2. Per la maggior parte, questi problemi consistono nel conoscere le proprietà di un tipo speciale di matrice detta *hamiltoniana*. Una volta ottenuta da Marie una matrice hamiltoniana, possiamo scrivere un'operazione quantistica molto simile a quella mostrata nel Listato 10.1, per simularla e studiarla, così che Marie possa usarla per capire come si comportano le vere sostanze chimiche. Per tutto il resto di questo capitolo, svilupperemo i concetti e le conoscenze di cui abbiamo bisogno per implementare la procedura descritta nella Figura 10.1.

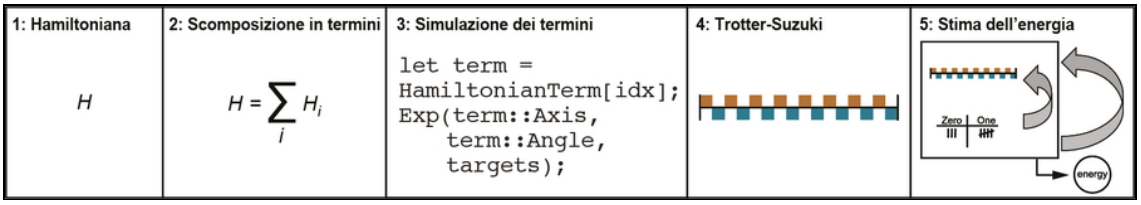


Figura 10.1 Panoramica dei passaggi che svilupperemo in questo capitolo per aiutare Marie a conoscere l'immagine energetica dello stato fondamentale della sua molecola.

In questo capitolo implementeremo le seguenti operazioni per il nostro algoritmo di simulazione hamiltoniana.

1. Collaborare con Marie per capire quale matrice hamiltoniana descrive i livelli di energia nel sistema cui è interessata, e un'approssimazione dello stato fondamentale (a energia più bassa).
2. Preparare l'approssimazione dello stato fondamentale e utilizzare l'operazione `Exp` di Q# per implementare l'evoluzione del sistema quantistico per ogni termine della matrice hamiltoniana.
3. Utilizzando la scomposizione di Trotter-Suzuki implementata nella funzione Q# `DecomposedIntoTimeStepsCA`, simulare l'evoluzione del nostro sistema sotto l'azione di tutti i termini della matrice hamiltoniana, suddividendo l'evoluzione in piccoli passi.
4. Dopo aver simulato l'evoluzione del sistema con la matrice hamiltoniana, utilizzare la stima della fase per conoscere il cambiamento di fase del nostro dispositivo quantistico.
5. Apportare una correzione finale alla fase che stimiamo per il sistema, dopodiché avremo l'energia dello stato fondamentale per H2.

Il listato seguente mostra questi passaggi tradotti in codice.

Listato 10.1 Codice Q# che stima l'energia dello stato fondamentale di H2.

```
operation EstimateH2Energy(idxBondLength : Int) : Double {
    let nQubits = 2;
    let trotterStepSize = 1.0;
    let trotterStep = EvolveUnderHamiltonian(idxBondLength,
        trotterStepSize, _);
    let estPhase = EstimateEnergy(nQubits,
        PrepareInitalState,
        trotterStep,
        RobustPhaseEstimation(6, _, _));
    return estPhase / trotterStepSize + H2IdentityCoeff(idxBondLength);
}
```

Senza altri indugi, affrontiamo il primo concetto quantistico di cui abbiamo bisogno per aiutare Marie: l'energia.

I tanti percorsi che portano alla meccanica quantistica

Finora, abbiamo imparato a conoscere la meccanica quantistica usando il linguaggio della programmazione: bit, qubit, istruzioni, dispositivi, funzioni e operazioni. Tuttavia, il modo in cui Marie ragiona in termini di meccanica quantistica è molto diverso (Figura 10.2). Per lei, la meccanica quantistica è una teoria fisica che le dice come si comportano le particelle subatomiche, come gli elettroni. Ragionando in termini di fisica e chimica, la meccanica quantistica è una teoria sulla *materia* di cui è fatto tutto ciò che ci circonda.

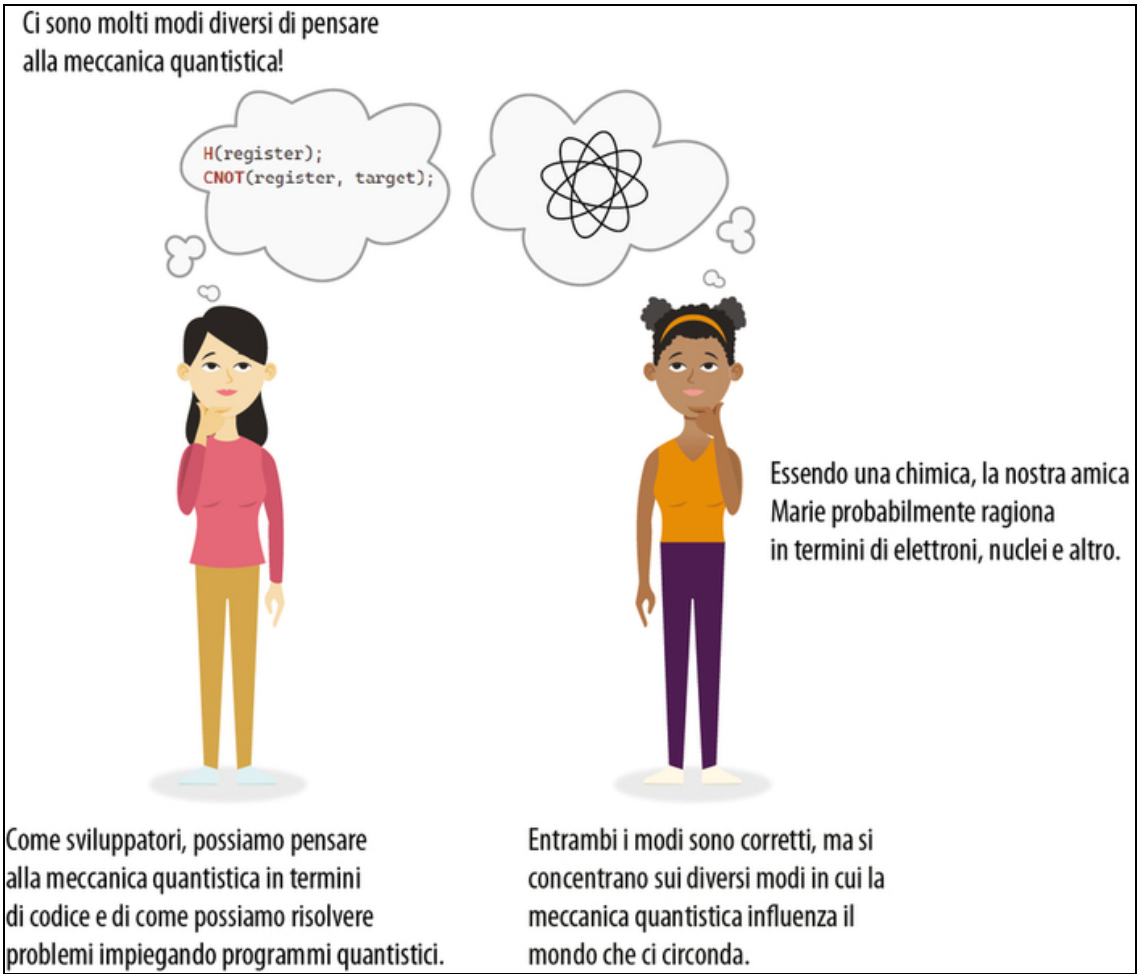


Figura 10.2 La meccanica quantistica vista in due modi molto diversi.

I due modi di pensare si incontrano quando giunge il momento di simulare il comportamento dei sistemi fisici, come le molecole. Possiamo usare computer quantistici per simulare il modo in cui altri sistemi quantistici si evolvono e cambiano nel tempo. Cioè, il quantum computing non riguarda solo la fisica o la chimica; può anche aiutarci a comprendere problemi scientifici come quelli affrontati da Marie.

Le informazioni sono il fulcro del modo in cui pensiamo al quantum computing; ma per un fisico e un chimico, la meccanica quantistica fa molto affidamento sul concetto di *energia*. L'energia ci dice il modo in cui i sistemi fisici, da una sfera a una bussola, sono influenzati dal

mondo che li circonda, dandoci un modo coerente per comprendere ciascuno di questi sistemi. Nella Figura 10.3, possiamo vedere come lo stato di una sfera su una collina e lo stato di una bussola possano essere descritti allo stesso modo, usando il concetto di energia.

Naturalmente, il concetto di energia non riguarda solo i sistemi classici come le palle e le bussole. In effetti, possiamo capire come si comportano i *sistemi quantistici*, come gli elettroni e i nuclei, studiando l'energia delle loro varie configurazioni. Nella meccanica quantistica, l'energia è descritta da un tipo speciale di matrice, detta hamiltoniana. Qualsiasi matrice che è la propria aggiunta può essere usata come hamiltoniana e le matrici hamiltoniane *non* sono operazioni esse stesse.

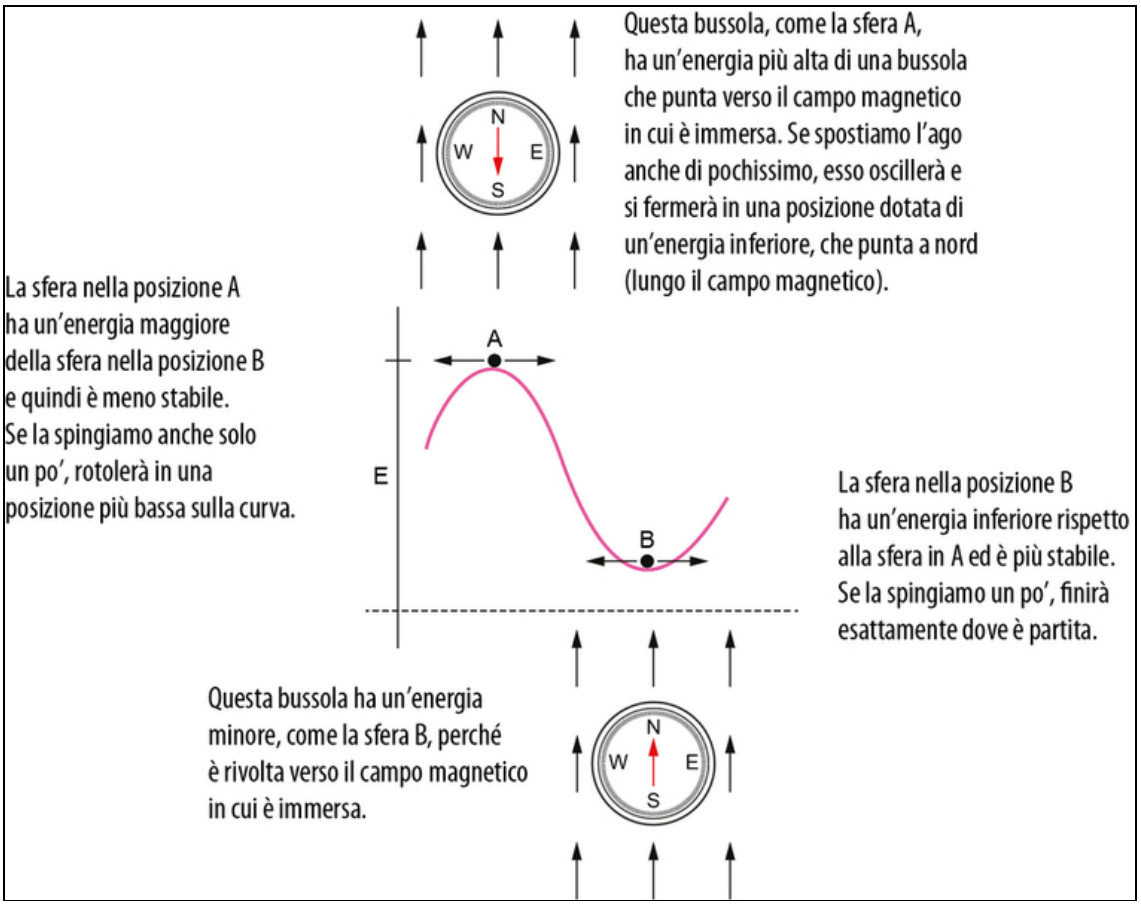


Figura 10.3 Usare l'energia per capire il modo in cui i diversi sistemi fisici sono influenzati dal loro ambiente. Una sfera in cima a una collina e una bussola che punta a Sud sono esempi di sistemi ad alta energia rispetto a una sfera in una valle o una bussola che punta a Nord.

Ricordiamo dai Capitoli 8 e 9 che l'*aggiunta* di una matrice A è la sua coniugata trasposta, $A^\dagger$. Questo concetto è strettamente correlato alla parola riservata `Adjoint` del linguaggio Q#: se un'operazione `op` può essere simulata dalla matrice unitaria U, allora l'operazione `Adjoint op` può essere simulata da $U^\dagger$.

In questo capitolo, scopriremo tutti gli strumenti e le tecniche per capire le energie dei sistemi quantistici per i quali abbiamo una matrice hamiltoniana. Spesso il processo per ottenere le matrici hamiltoniane per i sistemi è frutto di una collaborazione, ma una volta che le abbiamo, insieme a qualche altra informazione, possiamo stimare l'energia di tale sistema. Questo processo è chiamato *simulazione hamiltoniana* ed è fondamentale per molte applicazioni del quantum computing, inclusa la chimica.

SUGGERIMENTO

Abbiamo già visto alcuni esempi di matrici hamiltoniane nei capitoli precedenti: tutte le matrici di Pauli (X, Y e Z) sono esempi di matrici hamiltoniane, oltre a essere matrici unitarie. Tuttavia, non tutte le matrici unitarie possono essere utilizzate come matrici hamiltoniane! La maggior parte degli esempi in questo capitolo richiede ancora un po' di lavoro prima di poterle applicare come operazioni quantistiche.

Marie è interessata a conoscere l'energia dei legami nelle sue sostanze chimiche. Quindi ha senso inventare una matrice hamiltoniana che descriva la sua molecola; in tal modo potremo aiutarla a stimare l'energia cui è interessata. In chimica, questa energia è spesso chiamata *energia dello stato fondamentale* e lo stato corrispondente è noto come *stato fondamentale* (o stato a energia minima).

Una volta che abbiamo la matrice hamiltoniana, il passo successivo consiste nel capire come costruire le operazioni che simuleranno il modo in cui il sistema quantistico cambierà nel tempo, in base alla descrizione data dalla matrice hamiltoniana. Nel prossimo paragrafo, impareremo a descrivere l'evoluzione di un sistema quantistico in base a una matrice hamiltoniana.

Quindi, avendo gli operatori che rappresentano la matrice hamiltoniana, la sfida successiva sarà quella di capire come simulare le matrici hamiltoniane sul nostro dispositivo quantistico. Probabilmente nel dispositivo fisico non ci sarà un'unica operazione che farà esattamente ciò di cui abbiamo bisogno, quindi dobbiamo trovare un modo per scomporre le nostre operazioni per la nostra matrice hamiltoniana in termini di ciò che il nostro dispositivo può fornire. Più avanti vedremo come prendere qualsiasi operazione ed esprimerla in termini di operazioni di Pauli, che sono comunemente disponibili come istruzioni hardware.

Una volta che avremo espresso la nostra matrice hamiltoniana come somma di matrici di Pauli, come possiamo simularle tutte sul nostro sistema? Probabilmente avremo più termini che si sommano tutti insieme per rappresentare l'azione della matrice hamiltoniana, e non necessariamente adatti. Verso la fine del capitolo impareremo a utilizzare il metodo di Trotter-Suzuki per applicare un po' di ogni termine dell'operazione per simulare l'evoluzione dell'intera cosa. A quel punto il nostro sistema quantistico si sarà evoluto in un modo che rappresenta la matrice hamiltoniana di Marie.

Infine, per calcolare l'energia del sistema descritto dalla matrice hamiltoniana che abbiamo trovato, possiamo usare la stima della fase. Nel paragrafo finale del capitolo impiegheremo l'algoritmo che abbiamo imparato nel Capitolo 9 per esplorare la fase applicata ai nostri qubit simulando la matrice hamiltoniana.

Uso di matrici hamiltoniane per descrivere l'evoluzione dei sistemi quantistici nel corso del tempo

La Figura 10.4 mostra un riepilogo dei passaggi per simulare un altro sistema quantistico con il nostro computer quantistico. Per impiegare una matrice hamiltoniana per descrivere l'energia di un sistema fisico o chimico, dobbiamo guardare ai suoi autostati e ai loro autovalori.

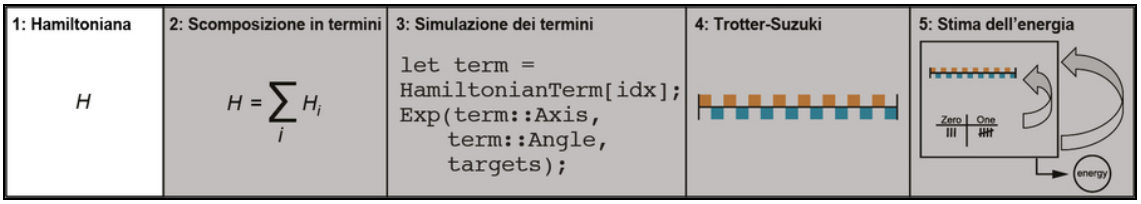


Figura 10.4 Cominciamo imparando a conoscere la molecola di H2 di Marie e quale matrice hamiltoniana descrive la sua evoluzione.

Dal Capitolo 9 sappiamo che se uno stato $|\psi\rangle$ è un autostato di un'operazione `op` allora, applicando `op` a un registro nello stato $|\psi\rangle$, al massimo si applica una fase globale a $|\psi\rangle$. Questa fase è chiamata l'*autovalore* o l'*autofase* di tale autostato. Come tutte le altre fasi globali, questa autofase non può essere osservata direttamente, ma possiamo usare il funtore `Controlled` di cui abbiamo parlato nel Capitolo 9 per trasformare quella fase in una fase locale.

Ogni autostato di una matrice hamiltoniana è uno stato a energia costante; proprio come le operazioni quantistiche non fanno nulla agli autostati, un sistema che si trova in un autostato della sua hamiltoniana rimarrà a quel livello di energia nel tempo. Vale anche l'altra proprietà degli autostati che abbiamo visto nel Capitolo 9: la fase di ogni autostato si evolve nel tempo.

L’osservazione che le fasi degli autostati si evolvono nel tempo è il contenuto dell’equazione di Schrödinger, una delle equazioni più importanti di tutta la fisica quantistica. L’equazione di Schrödinger dice che quando un sistema quantistico si evolve, ogni autostato di una matrice hamiltoniana accumula una fase proporzionale alla sua energia. Tramite operazioni matematiche, possiamo scrivere l’equazione di Schrödinger come illustrato nella Figura 10.5.

I veri esperti sono gli amici che ci siamo fatti strada facendo

Come può essere che siamo nel Capitolo 10 e stiamo vedendo solo ora l’equazione più importante nella fisica quantistica? Lo sviluppo di applicazioni quantistiche può essere strettamente connesso alla fisica quantistica, ma non è la stessa cosa, e non abbiamo bisogno di essere esperti di fisica per scrivere applicazioni quantistiche: possiamo certamente esserlo, se siamo interessati, ma questo non è assolutamente un requisito. L’equazione di Schrödinger entra in gioco solo qui perché ne abbiamo bisogno per capire come usare i computer quantistici per un impatto pratico.

Proprio come la nostra amica Marie è un’esperta di chimica quantistica e non di calcolo quantistico, non abbiamo bisogno di sapere tutto per fare qualcosa di fantastico. Ecco a cosa servono gli amici!

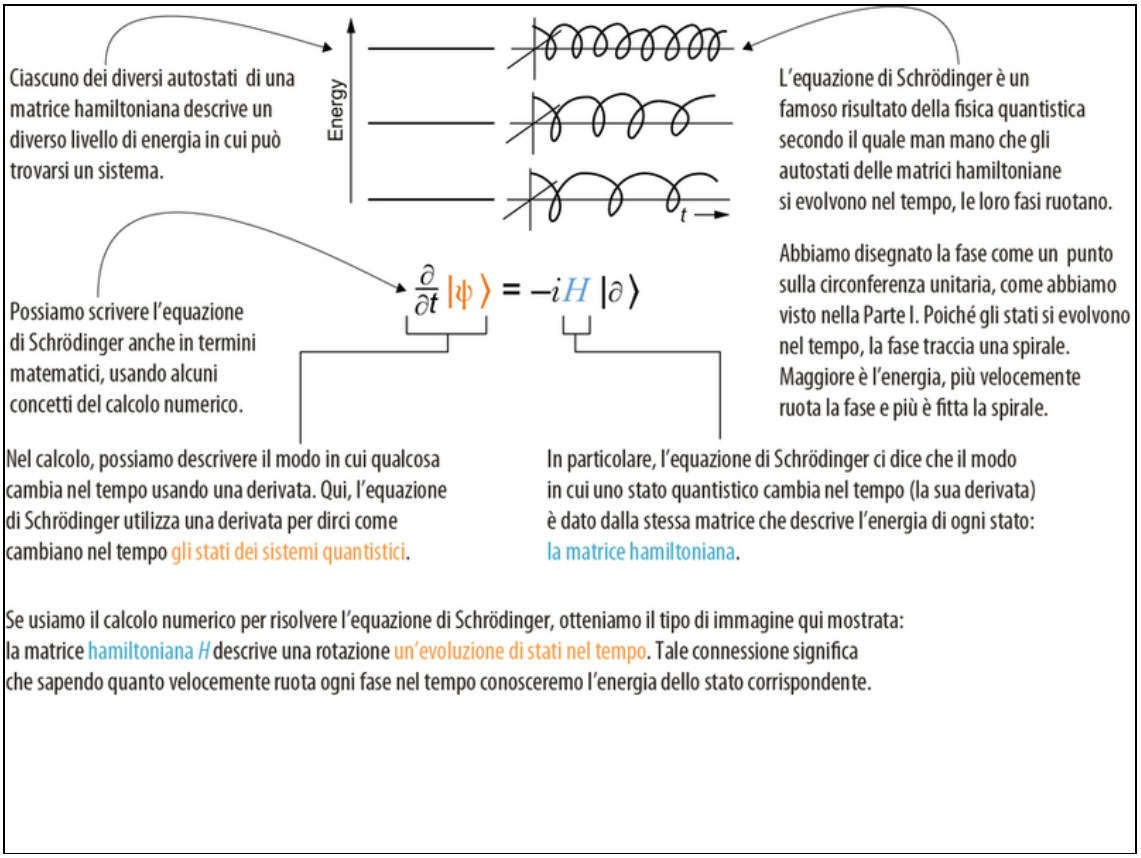


Figura 10.5 L'equazione di Schrödinger, scritta in notazione matematica.

SUGGERIMENTO

L'equazione di Schrödinger mette in relazione il modo in cui le fasi di stati differenti si evolvono nel tempo con l'energia di tali stati. Poiché le fasi globali non sono osservabili e poiché gli autostati delle matrici hamiltoniane acquisiscono fasi globali solo mentre si evolvono, l'equazione di Schrödinger ci dice che gli autostati delle matrici hamiltoniane non si evolvono nel tempo.

L'equazione di Schrödinger è fondamentale per noi in questo capitolo, perché mette in relazione l'energia di un sistema con la fase, una connessione molto utile, dato che nel Capitolo 9 abbiamo imparato proprio a stimare la fase. Ci sono altri modi in cui l'equazione di Schrödinger può essere utile, uno dei quali è una visione alternativa dell'implementazione delle operazioni sui sistemi quantistici. Un modo per implementare le rotazioni che abbiamo visto finora nel libro

consiste nell'impostare la giusta matrice hamiltoniana e poi… aspettare. La derivata temporale ($\partial/\partial t$) nell'equazione di Schrödinger ci dice che il modo in cui ruotano i nostri qubit è interamente descritto dall'energia associata a ciascuno stato. Per esempio, l'equazione di Schrödinger ci dice che se la nostra matrice hamiltoniana è $H = \omega Z$ per qualche numero ω, se vogliamo eseguire una rotazione di un angolo θ attorno all'asse Z, possiamo far evolvere il nostro qubit nel tempo $t = \theta / \omega$.

ESERCIZIO 10.1 ROTAZIONI

Provate a scrivere altre rotazioni mostrate in precedenza nel libro (per esempio, `Rx` e `Ry`) come matrici hamiltoniane.

SOLUZIONI DEGLI ESERCIZI

Tutte le soluzioni degli esercizi di questo libro si trovano nel suo archivio di codice: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Aprite la cartella del capitolo in cui vi trovate e poi il notebook Jupyter con il nome che menziona le soluzioni degli esercizi.

Il Listato 10.2 mostra una semplice operazione Q# che simula l'evoluzione sotto la matrice hamiltoniana $H = \omega Z$.

NOTA

In pratica, gran parte della difficoltà nella costruzione di un computer quantistico è legata al fatto di garantire che i qubit non si evolvano se non come indicato da un programma quantistico. Non sarebbe molto utile se, dopo esserci allontanati per un momento dal nostro dispositivo quantistico trovassimo tutti i nostri qubit in stati completamente diversi. Questo fa parte del motivo per cui, come sviluppatori quantistici, tendiamo a ragionare in termini di istruzioni inviate a un dispositivo, cioè di operazioni quantistiche, e non direttamente in termini di matrici hamiltoniane.

Cambiando registro per un momento e pensando temporaneamente in termini di matrici hamiltoniane ci dà modo di iniziare a fare progressi sul problema che ci è stato sottoposto da Marie. Dopotutto, i problemi con cui lavora Marie sono molto più facili da descrivere in

quel linguaggio. Per esempio, nel Capitolo 9 abbiamo visto come possiamo conoscere la fase applicata da rotazioni come quella che Dagonet ha nascosto a Lancillotto. Ma possiamo esprimere il gioco di Dagonet e Lancillotto anche in termini di matrici hamiltoniane. Supponiamo che Dagonet nasconda l'angolo di rotazione 2.1 π; poiché la sua rotazione era intorno all'asse *Z*, potremmo anche descrivere tale rotazione nascosta come una matrice hamiltoniana nascosta $H = -2.1\ \pi\ Z$.

> **NOTA**
>
> Abbiamo bisogno del segno nell'equazione di Schrödinger. Questo errore è tanto comune nella programmazione quantistica quanto lo sono le classiche sviste in altri linguaggi, quindi non preoccupatevi se dimenticherete una o due volte quel fastidioso segno meno. Va tutto bene.

In base a questo tipo di descrizione, la scala di Lancillotto corrisponde al tempo in cui lascia che i suoi qubit si evolvano sotto la matrice hamiltoniana nascosta di Dagonet. Mentre il fatto di ragionare in termini di gioco rende più facile scrivere programmi quantistici per apprendere la rotazione nascosta di Dagonet, ragionare in termini di matrice hamiltoniana aiuta a mappare meglio i tipi di concetti fisici di cui si occupa Marie, come l'intensità di campo e il tempo. L'unica parte complicata è che dobbiamo moltiplicare l'angolo di R_Z per `2.0`, poiché R_Z moltiplica il suo angolo per -1 / 2, per convenzione in Q#. Dal momento che l'equazione di Schrödinger ci dice che l'angolo richiede un segno meno, `2.0` ci dà l'angolo di cui abbiamo bisogno nella Figura 10.5.

Listato 10.2 Evoluzione sotto la matrice hamiltoniana H = ωZ.

```
operation EvolveUnderZ( (1)
    strength : Double, (2)
    time : Double, (3)
    target : Qubit
) : Unit is Adj + Ctl {
    Rz(2.0 * strength * time, target); (4)
}
```

(1) Operazione che simula l'evoluzione sotto $H = \omega Z$ usando Q#.
(2) ω dice quanto sono grandi le energie descritte dalla matrice hamiltoniana. Questo gioca il ruolo dell'angolo segreto di Dagonet nel Capitolo 9.
(3) Per quanto tempo vogliamo simulare la matrice hamiltoniana. Questo è analogo alla scala di Lancillotto del Capitolo 9.
(4) La simulazione effettiva è costituita da una sola riga, poiché le rotazioni attorno all'asse Z sono standard in Q#.

Poiché l'equazione di Schrödinger ci dice che l'evoluzione delle matrici hamiltoniane ruota i sistemi quantistici in base alla loro energia, se possiamo *simulare* la matrice hamiltoniana che ci ha dato Marie, allora possiamo giocare esattamente allo stesso gioco di stima della fase del Capitolo 9 per scoprire i livelli di energia di tale matrice hamiltoniana.

Approfondimento: le matrici hamiltoniane sono l'unica cosa che possiamo controllare

Quando abbiamo introdotto per la prima volta le operazioni quantistiche, come `H`, `X` e `Z`, vi sarete forse chiesti come le avremmo implementate su un vero dispositivo quantistico. Usando il concetto di matrice hamiltoniana, possiamo rivisitare tale domanda ed esplorare come funzionano le operazioni quantistiche intrinseche sull'hardware.

Quando a un sistema fisico viene applicato un campo magnetico con un dipolo magnetico (per esempio, uno spin degli elettroni), la matrice hamiltoniana di quel sistema include un termine che descrive il modo in cui il sistema interagisce con il campo magnetico. Tipicamente, scriviamo tale termine come $H = \gamma BZ$, dove B è la forza di quel campo magnetico e γ è un numero che descrive la forza con cui tale sistema risponde ai campi magnetici. Pertanto, per applicare una rotazione `Rz` su hardware quantistico che utilizza lo spin degli elettroni per implementare i qubit, possiamo attivare un campo magnetico e attendere il giusto periodo di tempo. Effetti simili possono essere usati per implementare altri termini hamiltoniani o per controllare la matrice hamiltoniana per altri dispositivi quantistici.

Lo stesso principio viene utilizzato anche in altre tecnologie quantistiche, come la risonanza magnetica nucleare (NMR), dove sono stati sviluppati buoni algoritmi

classici per costruire matrici hamiltoniane efficaci facendo *pulsare* i campi magnetici alla giusta frequenza o costruendo impulsi di forma complessa da applicare a un'operazione quantistica. Tradizionalmente, nella NMR e nel quantum computing più in generale, agli algoritmi di definizione dell'impulso vengono dati acronimi stravaganti, come GRAPE, CRAB, D-MORPH e persino ACRONYM. Indipendentemente dalla fantasia degli sviluppatori, tuttavia, questi algoritmi ci consentono di utilizzare computer classici per progettare operazioni quantistiche, date le matrici hamiltoniane di controllo come $H = \gamma BZ$. Se siete interessati a saperne di più, il documento originale di GRAPE presenta gran parte degli aspetti teorici del controllo ottimale che è stata utilizzata (Navin Khaneja et al., *Optimal Control of Coupled Spin Dynamics: Design of NMR Pulse Sequences by Gradient Ascent Algorithms*, in "Journal of Magnetic Resonance" 172, no. 2 (2005): 296, https://www.sciencedirect.com/science/article/abs/pii/S1090780704003696).

In pratica, naturalmente questo non è tutto. Non solo la progettazione degli impulsi di controllo è molto più complessa, ma per i computer quantistici a tolleranza di errore, le operazioni intrinseche con cui lavoriamo come sviluppatori quantistici non si mappano direttamente alle operazioni fisiche allo stesso modo dell'hardware disponibile. Piuttosto, queste operazioni hardware di basso livello vengono utilizzate per creare codici di correzione degli errori, in modo tale che una singola operazione intrinseca possa scomporsi in più impulsi applicati al nostro dispositivo.

Supponiamo che, invece di $H = \omega Z$, Marie chieda se possiamo simulare $H = \omega X$. Fortunatamente, il linguaggio Q# fornisce anche le rotazioni sull'asse X, quindi possiamo modificare la chiamata a Rz nel Listato 10.2 con una chiamata a Rx. Sfortunatamente, non tutte le matrici hamiltoniane a cui Marie è interessata sono semplici come $H = \omega Z$ o $H = \omega X$, quindi vediamo quali tecniche di sviluppo quantistico possiamo usare per simulare le matrici hamiltoniane più difficili.

Queste non sono le matrici hamiltoniane che stiamo cercando

È probabile che quando inizieremo a parlare con Marie, ci descriverà anche le matrici hamiltoniane per il suo sistema nella sua simulazione e nel suo software di modellazione. Tuttavia, probabilmente si tratterà di *hamiltoniane fermioniche*, che sono diverse dal tipo che stiamo usando per descrivere il modo in cui cambiano nel tempo i dispositivi quantistici. Come parte del flusso di lavoro della nostra collaborazione con Marie, probabilmente avremo bisogno di utilizzare alcuni

strumenti come NWChem (https://nwchemgit.github.io) per eseguire conversioni tra una matrice hamiltoniana che descrive come una sostanza chimica cambia nel tempo e il modo in cui cambiano i qubit nel corso del tempo. Questo libro non può esaminare tali metodi in dettaglio, ma sono disponibili ottimi strumenti software che possono utili a risolvere questo problema. Se siete interessati potete consultare la documentazione del Quantum Development Kit: https://docs.microsoft.com/azure/quantum/user-guide/libraries/chemistry.
Questo non è un grosso problema, per ora, solo un consiglio utile per quando occorrerà parlare con i vari collaboratori.

Rotazione attorno ad assi arbitrari con le operazioni di Pauli

Salendo di complessità, forse Marie è interessata a qualcosa che richiede più di una matrice hamiltoniana mono-qubit per essere descritto (Figura 10.6). Se ci dà una matrice hamiltoniana come $H = \omega X \oplus X$, che cosa possiamo fare per simularla? Fortunatamente, ciò che abbiamo imparato sulle rotazioni nella Parte I di questo libro è ancora utile anche per quel tipo di hamiltoniana a due qubit, poiché possiamo considerarla come se descrivesse un altro tipo di rotazione.

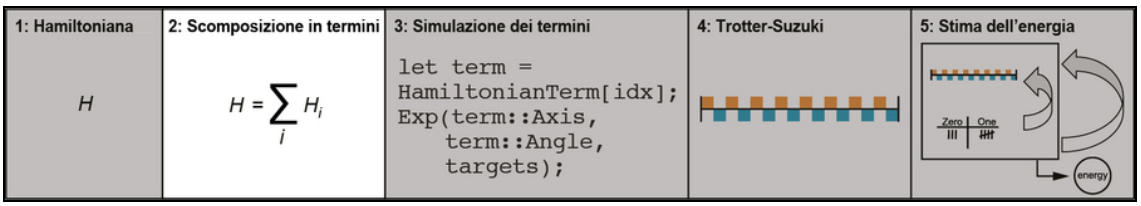


Figura 10.6 Nel Passo 2, vediamo come scomporre la nostra matrice hamiltoniana in rotazioni generalizzate più facili da simulare.

NOTA

Nel paragrafo precedente, abbiamo visto che rotazioni come `Rx`, `Ry` e `Rz` corrispondono, rispettivamente, a matrici hamiltoniane come X, Y e Z. Possiamo pensare che matrici hamiltoniane a due bit come $X \otimes X$ specifichino

un asse più o meno allo stesso modo. Risulta che ci siano 15 possibili assi di rotazione ortogonali per un registro di due qubit, invece delle 3 dimensioni che otteniamo per un registro mono-qubit. Quindi, per disegnare un'immagine, avremmo bisogno un "foglio di carta" di ben 13 dimensioni in più rispetto al normale, cosa che rende un po' difficile ogni illustrazione!

Questa rotazione non somiglia a nessuna delle istruzioni fornite (cioè intrinseche) che abbiamo visto finora, quindi potrebbe sembrare che siamo bloccati. Tuttavia, possiamo ancora simulare questa matrice hamiltoniana usando rotazioni *mono-qubit* come `Rx`, purché usiamo alcune operazioni su due qubit su entrambi i lati. In questo paragrafo, vedremo come funziona l'operazione e come il linguaggio Q# aiuti ad automatizzare l'applicazione di rotazioni multi-qubit.

Per iniziare, vediamo alcuni dei modi in cui possiamo cambiare ciò che fanno le operazioni quantistiche, circondandole con altre operazioni. Possiamo sempre usare la matematica per ragionare sulle cose, come abbiamo visto nel Capitolo 9, ma fortunatamente il linguaggio Q# ci fornisce anche alcune utili funzioni e operazioni di test. Per esempio, nel Capitolo 9, abbiamo visto che circondare un'operazione `CNOT` con operazioni `H` ci dà un `CNOT` che va nella direzione opposta. Vediamo come possiamo verificarlo usando Q#!

SUGGERIMENTO

Il blocco `within`/`apply` del Listato 10.3 applica il principio delle "calze e scarpe" che abbiamo introdotto nel Capitolo 9. La maggior parte degli esempi di codice in questo paragrafo usa dei blocchi `within`/`apply` per gestire il concetto "calze e scarpe".

Listato 10.3 Modifica di control e target di un CNOT.

```
open Microsoft.Quantum.Diagnostics; (1)
operation ApplyCNOT(register : Qubit[]) : Unit is Adj + Ctl { (2)
    CNOT(register[0], register[1]);
}
operation ApplyCNOTTheOtherWay(register : Qubit[]) : Unit is Adj + Ctl {
(3)
    within {
        ApplyToEachCA(H, register);
```

```
    } apply {
        CNOT(register[1], register[0]);
    }
}
operation CheckThatThisWorks() : Unit {
    AssertOperationsEqualReferenced(2, ApplyCNOT, ApplyCNOTTheOtherWay);
(4)
    Message("Woohoo!"); (5)
}
```

(1) Le operazioni e funzioni del namespace `Microsoft.Quantum.Diagnostics` aiutano con il testing e il debugging dei programmi quantistici e possono essere molto utili per assicurarci che i programmi funzionino come previsto.

(2) Per confrontare i due modi di scrivere un'operazione `CNOT`, è necessario che ciascuno sia richiamabile come un'operazione che accetta un array di qubit, che rappresenta un registro quantistico.

(3) Per verificare l'equivalenza che abbiamo visto nel Capitolo 7, possiamo scrivere una seconda operazione che inverte `control` e `target` in un'operazione `CNOT`.

(4) Il primo input specifica le dimensioni di un registro su cui agisce ciascuna operazione e il secondo e il terzo input rappresentano le operazioni da confrontare. Se le operazioni fanno qualcosa di diverso, l'`assert` fallisce e il programma quantistico termina.

(5) Se vediamo il messaggio "Woohoo!", Possiamo concludere con sicurezza che le due operazioni non possono essere distinte l'una dall'altra osservando ciò che fanno agli stati dei registri quantistici.

> **NOTA**
>
> Le `assert` come `AssertOperationsEqualReferenced` hanno senso solo quando vengono eseguite su un simulatore, poiché la loro esecuzione richiede la violazione del teorema di non clonazione. Su un vero hardware quantistico, questo tipo di `assert` verrebbe eliminato, così come l'esecuzione di Python con l'argomento `-O` sulla riga di comando disabilita la parola riservata `assert`. Ciò significa che le `assert` Q# ci danno un modo per "barare" *in modo sicuro*,

poiché i programmi quantistici che utilizzano `assert` faranno la stessa cosa, indipendentemente dal fatto che imbrogliamo.

ESERCIZIO 10.2 VERIFICA DELLE IDENTITÀ DI CNOT

Utilizzate QuTiP per verificare che le due operazioni `ApplyCNOT` e `ApplyCNOTTheOtherWay` possano essere simulate dalla stessa matrice unitaria e quindi fare esattamente la stessa cosa.

ESERCIZIO 10.3 TRE CNOT FANNO UNA SWAP

Proprio come possiamo usare tre istruzioni `XOR` classiche per implementare uno swap classico in loco, possiamo usare tre operazioni `CNOT` per fare la stessa cosa di una singola operazione `SWAP`. Il seguente frammento di codice Q# fa la stessa cosa di `SWAP(left, right)`:

```
CNOT(left, right);
CNOT(right, left);
CNOT(left, right);
```

Verificate che sia la stessa cosa di `SWAP(left, right)` utilizzando `AssertOperationsEqualReferenced` e QuTiP.

Credito extra: `SWAP(left, right)` è uguale a `SWAP(right, left)`, quindi il frammento di codice precedente dovrebbe funzionare anche se iniziamo con `CNOT(right, left)`. Verificatelo!

Approfondimento: l'isomorfismo di Choi-Jamiłkowski

L'operazione `AssertOperationsEqualReferenced` del Listato 10.3 funziona utilizzando un bel principio matematico chiamato *isomorfismo di Choi-Jamiłkowski*, il quale dice che qualsiasi operazione che possa essere simulata usando una matrice unitaria è perfettamente equivalente a un particolare stato che è il suo *stato di Choi*. Ciò significa che un simulatore può effettivamente trovare l'intera tabella di verità per qualsiasi operazione dotata di `adjoint` (ovvero, qualsiasi operazione che ha `Adj` nella sua signature) trovando il suo stato di Choi. L'operazione `AssertOperationsEqualReferenced` utilizza questo principio per preparare un registro di qubit nello stato di Choi per ciascuna delle operazioni passate come input. Su un simulatore, è facile barare e verificare se due stati sono uguali, ma il teorema di non clonazione dice che non possiamo farlo su un dispositivo realmente quantistico.

Quando si scrivono unit test e altre verifiche che i programmi quantistici siano corretti, questa può essere una tecnica molto potente per impiegare simulatori classici pur mantenendo l'impossibilità di barare su un hardware realmente quantistico.

Quando eseguiamo `CheckThatThisWorks` in Jupyter Notebook (come abbiamo visto nel Capitolo 7) o dalla riga di comando, dovremmo vedere il messaggio `"Woohoo!"`, il quale ci dice che il nostro programma Q# ha superato la chiamata a `AssertOperationsEqualReferenced`. Poiché tale `assert` è valida solo se le due operazioni che le diamo in pasto fanno esattamente la stessa cosa per tutti i possibili input, sappiamo che l'equivalenza che abbiamo imparato nel Capitolo 7 funziona.

Possiamo usare la stessa logica per verificare come le operazioni a due qubit, come `CNOT`, trasformano altre operazioni. Per esempio, trasformare una chiamata a `X` con più chiamate a `CNOT` fa la stessa cosa che richiamare `X` più volte, come mostra il listato successivo.

Listato 10.4 Applicare X su ogni qubit di un registro.

```
open Microsoft.Quantum.Diagnostics;
open Microsoft.Quantum.Arrays;
operation ApplyXUsingCNOTs(register : Qubit[]) : Unit is Adj + Ctl { (1)
    within {
        ApplyToEachCA( (2)
            CNOT(register[0], _), (3)
            Rest(register) (4)
        );
    } apply {
        X(register[0]); (5)
    }
}
operation CheckThatThisWorks() : Unit {
    AssertOperationsEqualReferenced(2,
        ApplyXUsingCNOTs,
        ApplyToEachCA(X, _) (6)
    );
    Message("Woohoo!");
}
```

(1) Operazione che rappresenta una singola chiamata a `X` con `CNOT`, utilizzando un blocco `within`/`apply`.

(2) Per la parte "calze" del blocco `within`/`apply`, possiamo scrivere le chiamate a `CNOT` di cui abbiamo bisogno utilizzando `ApplyToEachCA` insieme alla tecnica di applicazione parziale che abbiamo imparato nel Capitolo 7.
(3) Questa parte della nostra chiamata ad `ApplyToEachCA` dice di applicare un'operazione `CNOT` controllata sul primo qubit di un registro a ciascun elemento di un intero array di qubit.
(4) Utilizza `Rest` per selezionare tutto tranne il primo (cioè lo "zeresimo") elemento dell'array di registri.
(5) La parte "scarpe" del nostro blocco `within`/`apply` è un po' più semplice: basta un'operazione `X` sullo stesso qubit che abbiamo utilizzato come `control` per la nostra sequenza di chiamate a `CNOT`.
(6) Questa volta, invece di scrivere la nostra operazione da confrontare, la confrontiamo con un'operazione `X` su ogni qubit di un registro, utilizzando un'applicazione parziale.

ESERCIZIO 10.4 EQUIVALENZA UNITARIA

Impiegando QuTiP, verificate che quando vengono eseguiti su registri a due qubit, i due programmi del Listato 10.4 possono essere simulati dalla stessa matrice unitaria e quindi fanno la stessa cosa sui loro registri di input.

ESERCIZIO 10.5 EQUIVALENZA DEI PROGRAMMI

Provate a modificare il Listato 10.4 per vedere se entrambi i programmi sono equivalenti se applicati a più di due qubit.
Nota: può essere piuttosto costoso utilizzare `AssertOperationsEqualReferenced` per più di pochi qubit.

Possiamo anche costruire altri tipi interessanti di operazioni usando il concetto `within`/`apply`. In particolare, trasformare una rotazione con operazioni `CNOT` nello stesso modo presentato nel Listato 10.4 ci consente di implementare i tipi di rotazioni multi-qubit che Marie ha richiesto all'inizio di questo paragrafo. Usando le funzionalità

`DumpMachine` e `DumpRegister` che abbiamo imparato a usare nel Capitolo 9, possiamo vedere che proprio come `Rx` applica una rotazione sull'asse X tra $|0\rangle$ e $|1\rangle$, possiamo implementare una rotazione sull'asse $(X \oplus X)$ tra $|00\rangle$ e $|11\rangle$.

Listato 10.5 Creazione di un'operazione Rx multi-qubit.

```
open Microsoft.Quantum.Diagnostics;
open Microsoft.Quantum.Math;
operation ApplyRotationAboutXX(angle : Double, register : Qubit[]) :
Unit is Adj + Ctl {
    within {
        CNOT(register[0], register[1]); (1)
    } apply {
        Rx(angle, register[0]); (2)
    }
}
operation DumpXXRotation() : Unit {
    let angle = PI() / 2.0;
    use register = Qubit[2]; (3)
    ApplyRotationAboutXX(angle, register); (4)
    DumpMachine(); (5)
    ResetAll(register); (6)
}
```

(1) Per semplicità, ci siamo specializzati sul caso due-qubit in questo listato, ma possiamo usare una chiamata `ApplyToEachCA` anche per lavorare con registri di più di due qubit.

(2) Invece di applicare un'operazione `X` al qubit `control`, vogliamo applicare al qubit `control` una rotazione `X` di un angolo arbitrario.

(3) Per verificare che cosa fa la nostra nuova operazione `ApplyRotationAboutXX`, iniziamo chiedendo alla nostra macchina target un registro di due qubit con un'istruzione `using`.

(4) Quindi applichiamo la nostra nuova rotazione sull'asse $(X \oplus X)$ al nostro nuovo registro, per vedere che cosa fa.

(5) Quando viene eseguita su un simulatore, `DumpMachine` mostra lo stato completo del simulatore, permettendoci di verificare come la nostra nuova operazione di rotazione ha trasformato lo stato del nostro registro.

❻ Come al solito, prima di rilasciare il nostro registro sulla macchina target, dobbiamo eseguire il reset di tutti i nostri qubit nello stato $|0\rangle$.

ESERCIZIO 10.6 PREVEDERE APPLYROTATIONABOUTXX

Provate a preparare il registro in stati differenti da $|00\rangle$ prima di richiamare `ApplyRotationAboutXX`. L'operazione fa quello che vi aspettavate?

Suggerimento: ricordate, dalla Parte I del libro, che possiamo preparare una copia dello stato $|1\rangle$ applicando un'operazione `X` e che possiamo preparare lo stato $|+\rangle$ applicando un'operazione `H`.

L'output del Listato 10.5 potrebbe apparire leggermente diverso da quello rappresentato nella Figura 10.7, poiché il kernel IQ# per Jupyter Notebook supporta diversi modi per etichettare gli stati dei qubit.

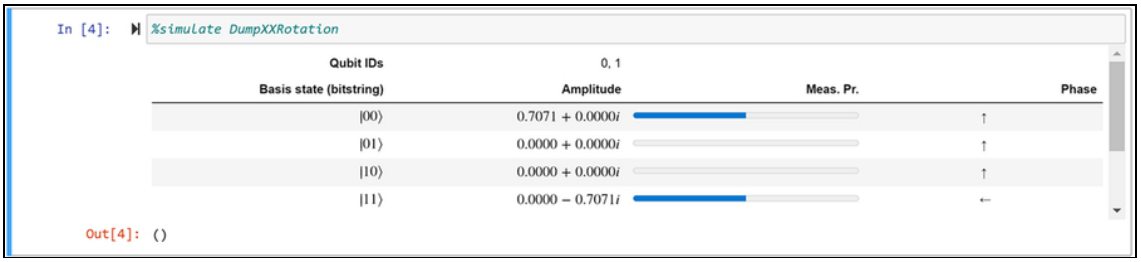


Figura 10.7 Output dell'esecuzione del Listato 10.5 in un notebook Jupyter. Possiamo vedere l'output di DumpMachine in DumpXXRotation, il quale mostra che il nostro stato risultante è una sovrapposizione tra $|00\rangle$ e $|11\rangle$.

SUGGERIMENTO

Per default, IQ# usa la convenzione "little-endian", utile per problemi aritmetici come quelli che vedremo nel Capitolo 12. Per etichettare gli stati dei qubit usando stringhe di bit come quelle che abbiamo visto finora nel libro, eseguite `%config dump.basisStateLabelingConvention = "Bitstring"` da una nuova cella di Jupyter Notebook.

ESERCIZIO 10.7 RX O ROTAZIONE ATTORNO A X ⊗ X

Provate a utilizzare `DumpMachine` per esplorare il funzionamento dell'operazione `Rx` su un singolo qubit e confrontatelo con la rotazione di due qubit sull'asse $(X \otimes X)$ che abbiamo implementato nel Listato 10.5. In che modo le due operazioni di rotazione sono simili e in che cosa differiscono?

Confrontate la rotazione attorno all'asse ($X \otimes X$) con l'applicazione di un'operazione `Rx` a ciascun qubit di un registro di due qubit.

In generale, *qualsiasi* rotazione attorno a un asse data da un prodotto tensoriale di matrici di Pauli (come $X \oplus X$, $Y \oplus Z$ o $Z \oplus Z \oplus Z$) può essere implementata applicando una rotazione mono-qubit trasformata da una sequenza di operazioni come `CNOT` e `H`. Ma trovare qual è la corretta trasformazione può essere un po' noioso, così il linguaggio Q# fornisce una comoda operazione standard: `Exp`.

Listato 10.6 Uso di Exp per capire come trasformare lo stato.

```
open Microsoft.Quantum.Diagnostics;
open Microsoft.Quantum.Math;
operation ApplyRotationAboutXX(angle : Double, register : Qubit[]) :
Unit is Adj + Ctl {
    within {
        CNOT(register[0], register[1]);
    } apply {
        Rx(angle, register[0]);
    }
}
operation CheckThatThisWorks() : Unit {
    let angle = PI() / 3.0;
    AssertOperationsEqualReferenced(2,
        ApplyRotationAboutXX(angle, _),
        Exp([PauliX, PauliX], -angle / 2.0, _)
    );
    Message("Woohoo!");
}
```

ATTENZIONE

Le convenzioni utilizzate da `Exp` e `Rx` per indicare gli angoli, differiscono di un fattore -1 / 2. Quando utilizzate l'operazione `Exp` e le operazioni di rotazione mono-qubit nello stesso programma, assicuratevi di ricontrollare tutti gli angoli!

Usando `Exp`, è facile simulare la matrice hamiltoniana $H = \omega X \oplus X$ o qualsiasi altra hamiltoniana composta da prodotti tensoriali di matrici di Pauli (Figura 10.8). Come mostra il listato seguente, in Q#, possiamo specificare ($X \oplus X$) dal valore `[PauliX, PauliX]` del linguaggio Q#.

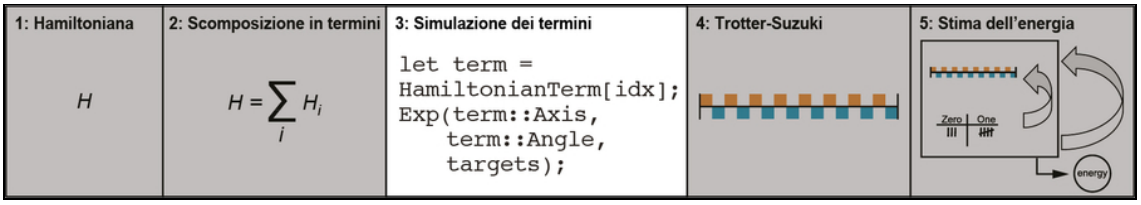


Figura 10.8 Nel Passo 3, vediamo come utilizzare l'operazione Exp per programmare rotazioni generalizzate che rappresentano la matrice hamiltoniana che stiamo cercando di simulare.

Listato 10.7 Utilizzo di Exp per simulare l'evoluzione sotto X ⊗ X.

```
operation EvolveUnderXX( (1)
    strength : Double, (2)
    time : Double, (3)
    target : Qubit
) : Unit is Adj + Ctl {
    Exp([PauliX, PauliX], strength * time, target); (4)
}
```

(1) Utilizzando quello che abbiamo imparato finora, possiamo scrivere un'operazione per simulare l'evoluzione sotto una matrice hamiltoniana proporzionale per ($X \otimes X$), così come l'operazione che abbiamo scritto nel Listato 10.2, simulata con un'evoluzione sotto un'hamiltoniano proporzionale a Z.

(2) Parametro che rappresenta la forza della matrice hamiltoniana: "quanto sono grandi" le energie descritte dalla nostra matrice hamiltoniana.

(3) Parametro che descrive per quanto tempo simulare l'evoluzione (analogo al parametro `scale` di Lancillotto nel Capitolo 9).

(4) Chiede una rotazione attorno all'asse ($X \oplus X$) usando l'operazione `Exp` fornita dal namespace `Microsoft.Quantum.Intrinsic`.

Z ⊗ Z non è solo due rotazioni Z

Si può essere tentati di pensare di poter implementare una rotazione di due qubit su $Z \otimes Z$ ruotando il primo qubit su Z e poi il secondo qubit su Z. Tuttavia, queste risultano essere operazioni molto differenti:

$$R_z(\theta) \otimes R_z(\theta) = \begin{pmatrix} e^{-i\theta} & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\theta} \end{pmatrix} \qquad R_{zz}(\theta) = \begin{pmatrix} e^{-i\theta/2} & 0 & 0 & 0 \\ 0 & e^{i\theta/2} & 0 & 0 \\ 0 & 0 & e^{i\theta/2} & 0 \\ 0 & 0 & 0 & e^{-i\theta/2} \end{pmatrix}$$

Un modo per considerarle è che una rotazione attorno a $Z \otimes Z$ è sensibile solo alla *parità* di ogni stato computazionale di base, quindi $|00\rangle$ e $|11\rangle$ vengono ruotati ciascuno della stessa fase.

Ora che abbiamo a disposizione l'operazione `Exp`, è abbastanza facile usarla per scrivere un'operazione che simuli ogni termine nella matrice hamiltoniana che ci dà Marie.

Listato 10.8 operations.qs: simulare l'evoluzione di un singolo termine.

```
operation EvolveUnderHamiltonianTerm(
    idxBondLength : Int, (1)
    idxTerm : Int, (2)
    stepSize : Double, (3)
    qubits : Qubit[])
: Unit is Adj + Ctl {
    let (pauliString, idxQubits) = H2Terms(idxTerm); (4)
    let coeff = (H2Coeff(idxBondLength))[idxTerm]; (5)
    let op = Exp(pauliString, stepSize * coeff, _); (6)
    (RestrictedToSubregisterCA(op, idxQubits)) (qubits); (7)
}
```

(1) Indice per cercare quale delle matrici hamiltoniane ci ha dato Marie. Ciascuno corrisponde a una diversa lunghezza di legame.
(2) Termine della matrice hamiltoniana di Marie in base al quale si vuole simulare l'evoluzione.
(3) Per quanto tempo simulare l'evoluzione: ovvero quanto tempo deve richiedere una fase della simulazione.
(4) Ottiene il termine dalla matrice hamiltoniana utilizzando `idxTerm` insieme alla funzione `H2Terms` fornita nell'archivio di codice di questo libro.

(5) Ottiene il coefficiente di quel termine utilizzando la funzione `H2Coeff`, fornita anche nell'archivio di esempi di questo libro

(6) Simula l'evoluzione in quel termine utilizzando `Exp` per eseguire una rotazione scalata dalla dimensione del passo della simulazione, proprio come l'operazione `EvolveUnderXX` del Listato 10.7.

(7) Poiché non tutti i termini influiscono su tutti i qubit, è possibile utilizzare l'operazione Q# `RestrictedToSubregisterCA` per applicare la nostra chiamata a `Exp` solo a un sottoinsieme dell'input.

Nel prossimo paragrafo, vedremo come simulare l'evoluzione sull'intera matrice hamiltoniana di Marie.

Apportare i cambiamenti che vogliamo vedere nel sistema

Ora che abbiamo imparato a descrivere il modo in cui un dispositivo quantistico può cambiare nel tempo usando il concetto di matrice hamiltoniana, una domanda molto naturale è: come implementiamo la specifica hamiltoniana che vogliamo simulare? La maggior parte dei dispositivi quantistici offre alcune comode operazioni per farlo. Per esempio, abbiamo visto nel paragrafo precedente che è semplice simulare l'evoluzione sotto qualsiasi matrice hamiltoniana data da un prodotto tensoriale di matrici di Pauli. Detto questo, la matrice hamiltoniana a cui noi (e Marie) siamo interessati probabilmente *non* è un'operazione standard del linguaggio Q#, ma piuttosto qualcosa che non è direttamente disponibile sul nostro computer quantistico.

> **SUGGERIMENTO**
>
> Di solito è facile che i dispositivi implementino alcuni degli operatori di Pauli e magari alcune altre operazioni. Il gioco diventa quindi cercare di capire come trasformare l'operazione che vogliamo in operazioni che il dispositivo può svolgere facilmente.

Se non esiste un modo semplice per simulare l'evoluzione sotto la nostra matrice hamiltoniana, come possiamo implementare una simulazione di una determinata hamiltoniana che possiamo applicare ai qubit del nostro dispositivo?

Vediamo come fare. Abbiamo imparato fin dal Capitolo 2 che possiamo descrivere un vettore come una combinazione lineare di vettori *di base* o direzioni. Possiamo fare la stessa cosa con le matrici, e una base davvero comoda per farlo sono gli operatori di Pauli.

Ripasso sulle matrici di Pauli

Se avete bisogno di un ripasso su che cosa sono le matrici Pauli, non preoccupatevi eccole qui:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Proprio come possiamo descrivere qualsiasi direzione su una mappa (Nord, Ovest…), possiamo descrivere qualsiasi matrice come una combinazione lineare di matrici di Pauli. Per esempio:

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} = \frac{1}{2}\mathbb{1} + \frac{1}{2}Z.$$

Allo stesso modo:

$$\begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix} = \frac{1}{2}\left(7\mathbb{1} + 7X - iY - 3Z\right).$$

Lo stesso vale per le matrici che operano su più qubit:

$$U_{\text{SWAP}} = \frac{1}{2}(\mathbb{1} \otimes \mathbb{1} + X \otimes X + Y \otimes Y + Z \otimes Z)$$

ESERCIZIO 10.8 VERIFICA DELLE IDENTITÀ

Usate QuTiP per verificare le equazioni precedenti.

Suggerimento: potete usare `qt.qeye(2)` per ottenere una copia di $\mathbb{1}$, `qt.sigmax()` per ottenere una copia di X e così via. Per calcolare i prodotti tensoriali, come $X \otimes X$, potete usare `qt.tensor`.

Questa è una buona notizia, perché così possiamo scrivere la matrice hamiltoniana che vogliamo simulare come una combinazione lineare di matrici di Pauli. Nel paragrafo precedente, abbiamo visto che possiamo usare `Exp` per simulare facilmente le matrici hamiltoniane costituite solo da prodotti tensoriali di matrici di Pauli. Ciò rende la base di Pauli molto comoda, poiché è probabile che il flusso di lavoro degli strumenti di chimica di Marie produca già la matrice hamiltoniana per il nostro dispositivo quantistico nella base di Pauli. Vediamo la rappresentazione della matrice hamiltoniana che Marie vuole che simuliamo, utilizzando la base di Pauli per espanderla. Usando le sue competenze di modellazione chimica, Marie può utilmente dirci che la matrice hamiltoniana che dobbiamo simulare con i nostri qubit è data dall'equazione seguente, dove a, $b0$, … $b4$ sono tutti numeri reali che dipendono dalla lunghezza di legame in cui vuole simulare H2:

$$H = a\mathbb{1} \otimes \mathbb{1} + b_0 Z \otimes \mathbb{1} + b_1 \mathbb{1} \otimes Z + b_2 Z \otimes Z + b_3 Y \otimes Y + b_4 X \otimes X$$

SUGGERIMENTO

Tutti i termini e i coefficienti utilizzati da Marie vengono da *Scalable Quantum Simulation of Molecular Energies (PJJ O'Malley et al., Scalable Quantum Simulation of Molecular Energies (2015), https://arxiv.org/abs/1512.06860)*. I coefficienti esatti dipendono dalla

lunghezza di legame tra gli atomi di idrogeno, ma tutte queste costanti sono opportunamente riportate nell'archivio di codice del libro: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp.

Avendo a disposizione questa rappresentazione della matrice hamiltoniana di Marie, è tempo di capire come usarla in pratica. Questa matrice hamiltoniana ha sei termini, quindi quale termine dovremmo applicare per primo? L'ordine è importante? Sfortunatamente, l'ordine in cui i termini sono usati spesso è importante quando si simula l'evoluzione di un sistema sotto una matrice hamiltoniana. Nel prossimo paragrafo, scopriremo un metodo che ci permette di suddividere l'evoluzione del sistema in piccoli passi, per simulare l'evoluzione sotto tutti i termini in una sola volta.

Svolgere alcuni (piccolissimi) cambiamenti

A questo punto, è utile fare un passo indietro e valutare a che punto siamo nell'aiutare Marie. Abbiamo visto come suddividere matrici hamiltoniane arbitrarie in somme di matrici di Pauli e come utilizzare l'operazione `Exp` per simulare l'evoluzione sotto ogni termine di tale somma. Per simulare matrici hamiltoniane arbitrarie, tutto ciò che resta da fare è combinare queste simulazioni per simulare l'intera hamiltoniana (Figura 10.9). Per farlo, possiamo usare un altro trucco del quantum computing: *la scomposizione di Trotter-Suzuki*.

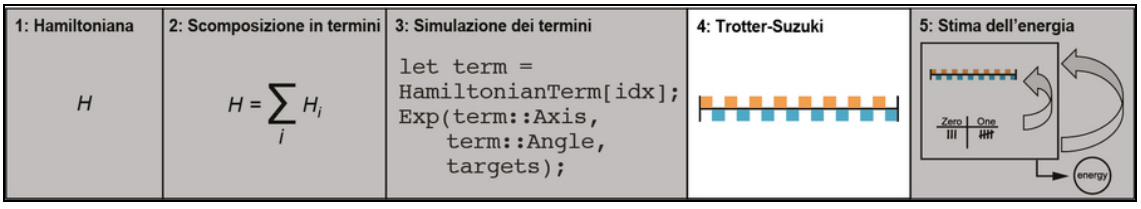


Figura 10.9 In questo paragrafo, vediamo come utilizzare la scomposizione di Trotter-Suzuki per simulare l'azione dell'intera matrice hamiltoniana, suddividendola

in evoluzioni molto più piccole di ciascun termine del Passo 3.

Prima di entrare nei dettagli della scomposizione di Trotter-Suzuki, però, torniamo all'analogia con la mappa che abbiamo usato in tutto il libro per trattare i concetti di algebra lineare (vedi Appendice C).

Supponiamo di voler esplorare il centro di Phoenix e di decidere di voler vedere come sarebbe andare a Nordest per tutta la città. Se iniziamo andando verso Nord per alcuni isolati e poi verso Est per alcuni isolati, il percorso che tracceremo sulla mappa non somiglierà molto a una linea diagonale. Al contrario, se alterneremo fra Nord ed Est a ciascun isolato, tracceremo qualcosa di molto più simile al percorso indicato nella mappa di Minneapolis nell'Appendice C. Cioè, siamo in grado di simulare il modo in cui potremmo camminare per Minneapolis anche se siamo a Phoenix, cambiando rapidamente la nostra direzione (vedi Figura 10.10).

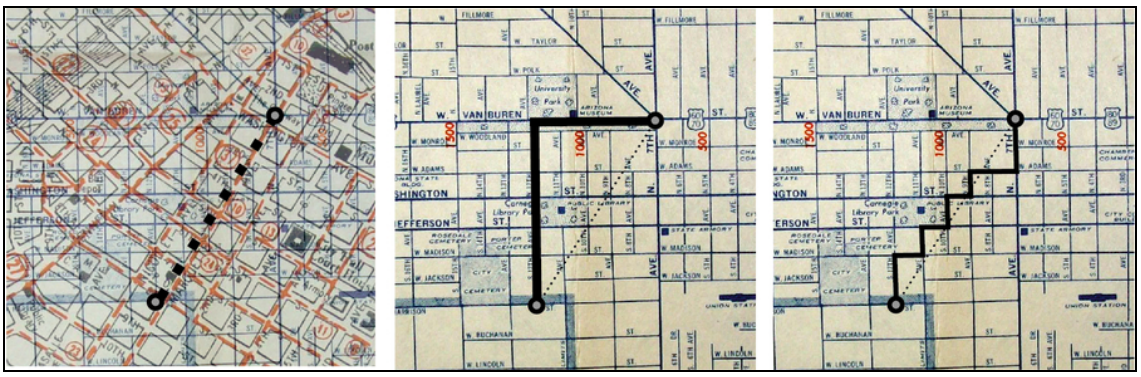


Figura 10.10 Se siamo nel centro di Phoenix, possiamo ancora simulare come potremmo attraversare il centro di Minneapolis alternando rapidamente le direzioni. Idealmente, dovremmo semplicemente viaggiare in diagonale verso la nostra destinazione, ma data la configurazione delle vie possiamo solo approssimare la diagonale, facendo brevi zigzag. Foto di davecito.

Nel paragrafo precedente, abbiamo visto che, proprio come gli stati, i diversi termini di una matrice hamiltoniana possono essere immaginati come direzioni su una mappa multidimensionale. I prodotti

tensoriali delle matrici di Pauli, come $Z \oplus \mathbb{1}$ e $X \oplus Z$, giocano un ruolo simile ai punti cardinali o agli assi di una mappa. Quando proviamo a simulare la matrice hamiltoniana di Marie, però, vediamo che non punta lungo un singolo asse, ma lungo una sorta di diagonale in quello spazio multidimensionale. È qui che entra in gioco la scomposizione di Trotter-Suzuki.

Proprio come il nostro percorso somiglia di più a una diagonale se cambiamo rapidamente la direzione in cui camminiamo, possiamo commutare rapidamente la simulazione di termini hamiltoniani. Come illustra la Figura 10.11, la scomposizione di Trotter-Suzuki ci dice che quando cambiamo rapidamente in questo modo, seguiamo approssimativamente un'evoluzione sotto la somma dei diversi termini che stiamo simulando.

In linea di principio potremmo scriverlo in formato Q# come un ciclo `for`. In pseudocodice, potremmo avere qualcosa di simile al seguente listato.

Listato 10.9 Simulare una matrice hamiltoniana usando Trotter-Suzuki.

```
operation EvolveUnderHamiltonian(time, hamiltonian, register) { (1)
    for idx in 0..nTimeSteps - 1 { (2)
        for term in hamiltonian { (3)
            evolve under term for time / nTimeSteps
        }
    }
}
```

(1) Dal momento che questo è pseudocodice, non preoccupiamoci troppo dei tipi. Questa operazione, infatti, non verrà compilata senza i tipi.

(2) Per ogni passo in cui vogliamo dividere la nostra simulazione (pensate agli isolati di una città o ai pixel di uno schermo), dobbiamo procedere per "un po'" in ogni termine hamiltoniano.

(3) All'interno di ogni intervallo temporale, possiamo creare un ciclo su ogni termine che abbiamo bisogno di simulare, e simularli tutti per

ciascun passo.

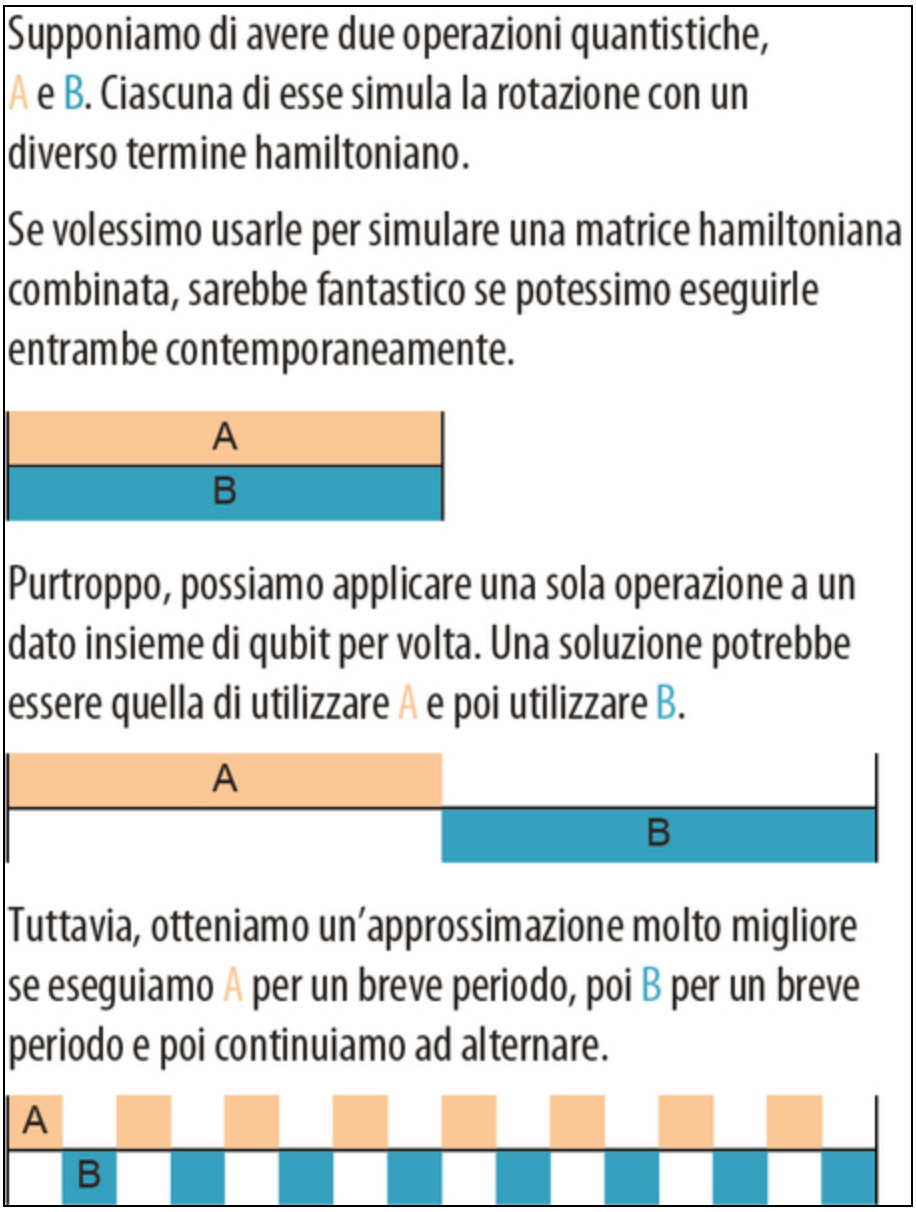


Figura 10.11 Utilizzo della scomposizione di Trotter-Suzuki per approssimare l'evoluzione sotto due termini hamiltoniani contemporaneamente. Proprio come l'analogia della mappa, se vogliamo applicare l'effetto di due matrici hamiltoniane il più rapidamente possibile, dovremmo alternarle, evolvendo un po' sotto ciascuna di esse fino a raggiungere la piena evoluzione.

Fortunatamente, Q# fornisce una funzione di libreria standard che fa proprio questo: `DecomposedIntoTimeStepsCA`. Nel Listato 10.10, mostriamo

come `DecomposedIntoTimeStepsCA` aiuti a usare la scomposizione di Trotter-Suzuki per simulare l'evoluzione sotto la matrice hamiltoniana di Marie. La funzione `DecomposedIntoTimeStepsCA` supporta le scomposizioni Trotter-Suzuki di ordine superiore rispetto all'approssimazione del primo ordine che abbiamo esplorato finora in questo capitolo (infatti, qui `trotterOrder` è uguale a `1`). In alcuni casi, questa funzione può essere utile per aumentare la precisione della nostra simulazione, ma `trotterOrder = 1` è più che adatto ai nostri scopi.

Listato 10.10 operations.qs: uso di DecomposedIntoTimeStepsCA.

```
operation EvolveUnderHamiltonian(
    idxBondLength : Int, (1)
    trotterStepSize : Double, (2)
    qubits : Qubit[])
: Unit is Adj + Ctl {
    let trotterOrder = 1; (3)
    let op = EvolveUnderHamiltonianTerm(idxBondLength, _, _, _); (4)
    (DecomposedIntoTimeStepsCA ((5, op), trotterOrder))
(trotterStepSize, qubits); (5)
}
```

(1) `EvolveUnderHamiltonian` applica la matrice hamiltoniana appropriata in base ai coefficienti per la lunghezza di legame desiderata per la molecola di H2 sulla quale Marie ci ha chiesto di aiutarla.
(2) Dimensione del passo che rappresenta per quanto tempo vogliamo simulare l'evoluzione hamiltoniana.
(3) In alcuni casi, un `trotterOrder` maggiore di `1` può essere utile per aumentare la precisione della simulazione, ma `trotterOrder = 1` funziona bene per i nostri scopi.
(4) L'applicazione parziale può correggere l'input `idxBondLength` in `EvolveUnderHamiltonianTerm`, lasciando vuoti gli argomenti `idxTerm`, `stepSize` e `qubits`.
(5) Questa funzione produce in output un'operazione che può essere utilizzata per simulare automaticamente l'evoluzione sotto l'intera

matrice hamiltoniana, usando l'operazione che simula ciascun termine, uno per uno, in modo da poter procedere e applicarlo.

Conclusione

Ora che abbiamo un'idea migliore di che cosa siano le matrici hamiltoniane e del modo in cui possiamo simulare l'evoluzione sotto di esse per capire il modo in cui i sistemi quantistici cambiano nel tempo, siamo pronti a mettere insieme un programma che aiuterà Marie a rispondere alla sua domanda (Figura 10.12). Come promemoria, Marie è una chimica che studia le energie dello stato fondamentale (ovvero le energie più basse possibili) di diverse sostanze chimiche. Ci ha chiesto di aiutarla a capire le energie dello stato fondamentale per la molecola H2 con il nostro dispositivo quantistico. Poiché anche gli atomi di idrogeno che compongono le molecole di H2 sono sistemi quantistici, è molto più facile simulare il loro comportamento con i qubit che con un computer classico.

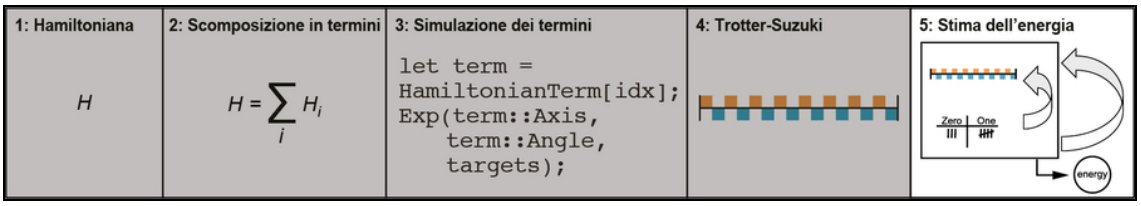


Figura 10.12 L'ultimo passo per aiutare Marie a simulare la sua molecola H2 consiste nell'usare la stima della fase per conoscere l'energia dello stato fondamentale.

SUGGERIMENTO

I computer quantistici sono talmente adatti a simulare il comportamento di altri sistemi quantistici che questa è stata probabilmente la prima applicazione mai proposta per il quantum computing.

La Figura 10.13 è un promemoria di tutti i passaggi e tutte le tecniche che abbiamo appreso in questo capitolo per simulare

l’evoluzione della molecola H2 di Marie nel nostro dispositivo quantistico.

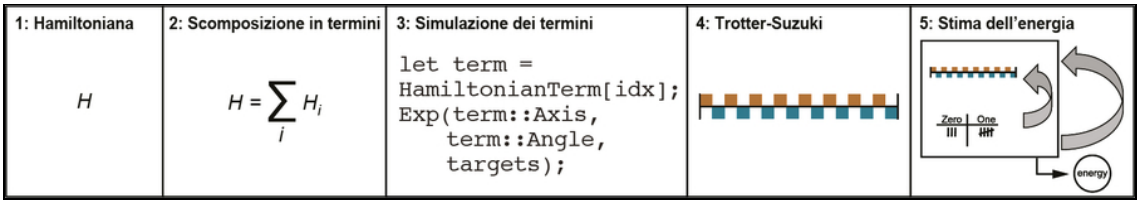


Figura 10.13 Panoramica dei passi sviluppati in questo capitolo per aiutare Marie a studiare l’energia dello stato fondamentale della sua molecola.

Pertanto, da sviluppatori quantistici possiamo collaborare con Marie per simulare l’evoluzione della molecola di H2 nel tempo e calcolare l’energia dello stato fondamentale, grazie all’equazione di Schrödinger. Il punto chiave da ricordare è che i possibili livelli di energia della molecola di H2 corrispondono ai diversi *autostati* della matrice hamiltoniana.

Supponiamo che i nostri qubit siano in un autostato della matrice hamiltoniana. Allora, la simulazione dell’evoluzione sotto tale matrice hamiltoniana non cambierà lo stato del nostro registro di qubit, *tranne* se applichiamo una fase globale proporzionale all’energia di quello stato. Tale energia ci dice esattamente di che cosa abbiamo bisogno per risolvere il problema di Marie, ma le fasi globali non sono osservabili. Fortunatamente, nel Capitolo 9 abbiamo imparato dal gioco di Lancillotto e Dagonet a trasformare le fasi globali in qualcosa che possiamo conoscere con la stima della fase: ecco un ottimo ambito in cui applicarlo! Riassumendo, i passaggi per collaborare con Marie e risolvere il suo problema sono i seguenti.

1. Preparare lo stato iniziale fornitoci da Marie. In questo caso, ci dice utilmente di preparare lo stato $|10\rangle$.
2. Suddividere la matrice hamiltoniana che rappresenta il sistema in piccoli passi che possano essere simulati sequenzialmente per

rappresentare l'intera operazione.

3. Applicare al nostro stato iniziale ogni passo che rappresenta la matrice hamiltoniana.
4. Utilizzare un algoritmo di stima della fase per conoscere la fase globale accumulata nel nostro stato quantistico, che sarà proporzionale all'energia.

Abbiamo le competenze e il codice delle sezioni precedenti di questo capitolo per mettere insieme tutto questo, quindi proviamo a farlo.

A partire dal file Q# (qui chiamato `operations.qs` per corrispondere a ciò che abbiamo visto nei capitoli precedenti), possiamo aprire alcuni namespace per utilizzare le funzioni e operazioni predefinite che essi offrono.

Listato 10.11 operations.qs: namespace necessari del QDK.

```
namespace HamiltonianSimulation {
    open Microsoft.Quantum.Intrinsic; (1)
    open Microsoft.Quantum.Canon; (1)
    open Microsoft.Quantum.Simulation; (2)
    open Microsoft.Quantum.Characterization; (3)
```

(1) Abbiamo già visto `Microsoft.Quantum.Intrinsic` e `Microsoft.Quantum.Canon`; hanno tutte le funzioni e le operazioni di utilità/di supporto di cui abbiamo bisogno.

(2) `Microsoft.Quantum.Simulation` è un namespace per il QDK che offre utility per la simulazione dei sistemi.

(3) `Microsoft.Quantum.Characterization` offre implementazioni di facile uso degli algoritmi di stima della fase che abbiamo sviluppato nel Capitolo 9.

Poi, dobbiamo aggiungere i dati che Marie ci ha dato sulla sua molecola. Tutto ciò si trova pronto all'uso nel file di esempio contenuto nell'archivio GitHub di questo libro:

https://github.com/crazy4pi314/learn-qc-with-python-and-

qsharp/blob/master/ch10/operations.qs. Ciò di cui abbiamo bisogno sono le funzioni `H2BondLengths`, `H2Coeff` e `H2IdentityCoeff`, un po' lunghe da riprodurre qui nel testo.

Una volta che abbiamo tutti i dati dei coefficienti di Marie in un file, abbiamo bisogno dei termini e della struttura per la matrice hamiltoniana che useremo con quei coefficienti. Il listato seguente mostra lo schema di una funzione che restituisce i termini della matrice hamiltoniana di Marie espressi come operatori di Pauli, nonché un'operazione che preparerà il nostro registro di due qubit nello stato corretto per l'algoritmo.

Listato 10.12 operations.qs: funzione che restituisce i termini dalla matrice hamiltoniana.

```
function H2Terms(idxHamiltonian : Int) : (Pauli[], Int[]) { (1)
    return [
        ([PauliZ], [0]), (2)
        ([PauliZ], [1]),
        ([PauliZ, PauliZ], [0, 1]), (3)
        ([PauliY, PauliY], [0, 1]),
        ([PauliX, PauliX], [0, 1])
    ][idxHamiltonian];
}
operation PrepareInitalState(q : Qubit[]) : Unit { (4)
    X(q[0]);
}
```

(1) La funzione `H2Terms` semplifica la costruzione dei termini della matrice hamiltoniana di Marie.

(2) Questa funzione è in realtà solo una lista di tuple che descrivono i termini della matrice hamiltoniana. Questa prima tupla dice che il primo termine della matrice hamiltoniana è l'operazione `PauliZ` sullo "zeresimo" qubit.

(3) Applica `PauliZ` sia allo zeresimo sia al primo qubit.

(4) Abbiamo anche bisogno di un modo per preparare i nostri qubit per l'algoritmo. Seguendo il consiglio di Marie, mettiamo il primo qubit nello stato $|1\rangle$, lasciando il resto dei qubit di input nello stato $|0\rangle$.

Per occuparci dei Passi 2 e 3 del nostro algoritmo quantistico, abbiamo bisogno delle operazioni che abbiamo definito in precedenza: `EvolveUnderHamiltonianTerm` ed `EvolveUnderHamiltonian`.

Fasi globali e funtore Controlled

Proprio come abbiamo visto nel Capitolo 9, l'applicazione di `EvolveUnderHamiltonian` non fa nulla ai qubit preparati in un autostato della matrice hamiltoniana di Marie (questo è esattamente il punto!). Nel Capitolo 9, siamo stati in grado di risolvere questo problema utilizzando il funtore `Controlled` per trasformare la fase globale, risultante dall'esecuzione dell'operazione di Dagonet su un qubit preparato in un autostato, in una fase locale che potesse essere osservata, e poi abbiamo utilizzato il *phase kickback* per applicare tale fase a un qubit `control`. L'operazione `EstimateEnergy` fornita dal Quantum Development Kit utilizza lo stesso identico trucco per conoscere quella che altrimenti sarebbe una fase globale della nostra operazione `EvolveUnderHamiltonian`. Questo significa che è *fondamentale* che la nostra operazione supporti il funtore `Controlled` con l'aggiunta di `is Ctl` alla signature di ogni operazione che passiamo a `EstimateEnergy`.

Infine, possiamo utilizzare l'operazione di stima dell'energia fornita con il Quantum Development Kit per automatizzare l'applicazione dei Passi Trotter-Suzuki e di stima della fase. Possiamo anche utilizzare un'operazione di stima della fase standard, che implementa una versione migliore dell'algoritmo che abbiamo introdotto nel Capitolo 9. Per esempio, la libreria `Microsoft.Quantum.Simulation` offre un'operazione chiamata `EstimateEnergy` che utilizza la stima della fase per stimare l'energia di un autostato. Accetta in input una specifica del numero di qubit (`nQubits`), un'operazione per preparare il nostro stato iniziale desiderato(`PrepareInitialState`), il modo in cui applicare la nostra matrice hamiltoniana (`trotterStep`) e l'algoritmo che vogliamo utilizzare per stimare la fase risultante dall'applicazione della nostra matrice hamiltoniana. Vediamola in azione.

Listato 10.13 operations.qs: operazione Q# che stima l'energia dello stato fondamentale di H2.

```
operation EstimateH2Energy(idxBondLength : Int) : Double { (1)
    let nQubits = 2; (2)
    let trotterStepSize = 1.0; (3)
    let trotterStep = EvolveUnderHamiltonian(idxBondLength,
trotterStepSize, _); (4)
    let estPhase = EstimateEnergy(nQubits, (5)
        PrepareInitalState, trotterStep,
        RobustPhaseEstimation(6, _, _));
    return estPhase / trotterStepSize + H2IdentityCoeff(idxBondLength);
(6)
}
```

(1) L'operazione `EstimateH2Energy` prende l'indice della lunghezza di legame della molecola e restituisce l'energia del suo stato fondamentale o stato energetico più basso.
(2) Definisce che abbiamo bisogno di due qubit per simulare questo sistema.
(3) Imposta un parametro di scala per i passi Trotter-Suzuki che applicano i termini della nostra matrice hamiltoniana ai nostri qubit.
(4) Si basa su `ApplyHamiltonian` e dà un nome comodo per l'operazione che applica i termini della nostra matrice hamiltoniana con i nostri parametri.
(5) Abbiamo inserito nella libreria `Microsoft.Quantum.Simulation` un'operazione che stima la fase risultante dall'applicazione della nostra matrice hamiltoniana, che sappiamo rappresenta l'energia del sistema.
(6) Per assicurarsi che le unità siano corrette per l'energia restituita, dobbiamo dividere per la dimensione del passo Trotter e sommare l'energia dal termine identità nell'hamiltoniana.

Non resta che eseguire effettivamente l'algoritmo. Poiché l'energia dello stato fondamentale è una funzione della lunghezza di legame molecolare, possiamo utilizzare un host Python per eseguire l'algoritmo Q# e quindi tracciare i risultati in funzione della lunghezza di legame.

Listato 10.14 host.py: predisposizione della simulazione in Python.

```
import qsharp (1)
import HamiltonianSimulation as H2Simulation
bond_lengths = H2Simulation.H2BondLengths.simulate() (2)
def estimate_energy(bond_index: float, (3)
                    n_measurements_per_scale: int = 3) -> float:
    print(f"Estimating energy for bond length of
{bond_lengths[bond_index]} Å.")
    return
min([H2Simulation.EstimateH2Energy.simulate(idxBondLength=bond_index)
                for _ in range(n_measurements_per_scale)])
```

(1) Importa il package Python per Q# e poi importa il namespace Q# `HamiltonianSimulation` dal nostro file `operations.qs`. Il package `qsharp` Python rende disponibili i namespace Q# come normali istruzioni `import`.

(2) Per semplificare le cose, estrae l'elenco delle lunghezze dei legami che possiamo simulare per H2 dalla funzione Q# `H2BondLengths`.

(3) La funzione `estim_energy` è un wrapper Python per l'operazione Q# `EstimateH2Energy`, ma la esegue alcune volte per assicurarsi che la stima dell'energia sia ridotta al minimo.

Perché abbiamo bisogno di eseguire più volte EstimateH2Energy?

Lo stato $|01\rangle$ che Marie ci ha fornito non è in realtà un autostato di alcuna matrice hamiltoniana di H2, ma qualcosa che Marie ha calcolato usando un'approssimazione della chimica quantistica nota come *teoria di Hartree-Fock*. Poiché la chimica quantistica è la sua area di competenza, può aiutarci fornendoci approssimazioni di questo tipo.

In pratica, questo significa che, quando eseguiamo la stima della fase utilizzando gli strumenti forniti dal namespace `Microsoft.Quantum.Characterization`, non stiamo studiando l'energia di un particolare autostato, ma la stiamo proiettando in modo casuale su un autostato e studiando la sua energia. Poiché il nostro stato iniziale è un'approssimazione piuttosto buona, la maggior parte delle volte eseguiremo la proiezione sullo stato di energia più bassa della matrice hamiltoniana di Marie (cioè lo stato fondamentale), ma possiamo anche essere sfortunati e studiare correttamente l'energia dell'autostato sbagliato. Dal momento che stiamo

cercando l'energia più piccola, ripetere più volte l'esecuzione e prendere il valore minimo rende molto più probabile che studieremo l'energia corretta.

Ora che tutto è impostato nell'host Python, non ci resta che scrivere ed eseguire la funzione principale.

Listato 10.15 host.py: il programma principale per la nostra simulazione.

```
if __name__ == "__main__":
    import matplotlib.pyplot as plt (1)
    print(f"Number of bond lengths: {len(bond_lengths)}.\n")
    energies = [estimate_energy(i) for i in range(len(bond_lengths))]
(2)
    plt.figure()   @3
    plt.plot(bond_lengths, energies, 'o') (3)
    plt.title('Energy levels of H2 as a function of bond length') (3)
    plt.xlabel('Bond length (Å)') (3)
    plt.ylabel('Ground state energy (Hartree)') (3)
    plt.show() (4)
```

(1) L'esecuzione di `host.py` come script traccia le energie stimate dello stato di base dall'algoritmo quantistico scritto in Q#.
(2) Genera direttamente la lista delle energie stimate per ogni lunghezza di legame della molecola H2.
(3) Impostazione dei dati e dello stile per il grafico.
(4) La chiamata a `plt.show()` mostra il grafico.

La Figura 10.14 mostra un esempio di ciò che dovrebbe restituire l'esecuzione di `python host.py`. Questo grafico mostra i risultati delle nostre simulazioni per varie matrici hamiltoniane per diverse lunghezze di legame della molecola H2. Possiamo vedere che l'energia dello stato più basso è molto più alta quando la lunghezza di legame è breve e si livella quando i legami si allungano. L'energia più bassa possibile dovrebbe verificarsi intorno a una lunghezza di legame di circa 0,75 Å. In effetti, la lunghezza di legame stabile (cioè l'equilibrio) per l'idrogeno è di 0,74 Å! Quindi sì, in effetti il comportamento della molecola di Marie è abbastanza noto, ma come possiamo vedere, potremmo seguire questo stesso processo per

simulare non solo altre sostanze chimiche ma anche altri sistemi quantistici.

Abbiamo così implementato la nostra prima applicazione pratica per un computer quantistico! Naturalmente, la sostanza chimica che abbiamo usato qui è abbastanza semplice, ma questo processo vale per la maggior parte dei sistemi quantistici che potremmo voler simulare. Nei prossimi due capitoli, esploreremo altre due applicazioni per i computer quantistici: la ricerca non strutturata con l'algoritmo di Grover e la fattorizzazione di numeri con l'algoritmo di Shor.

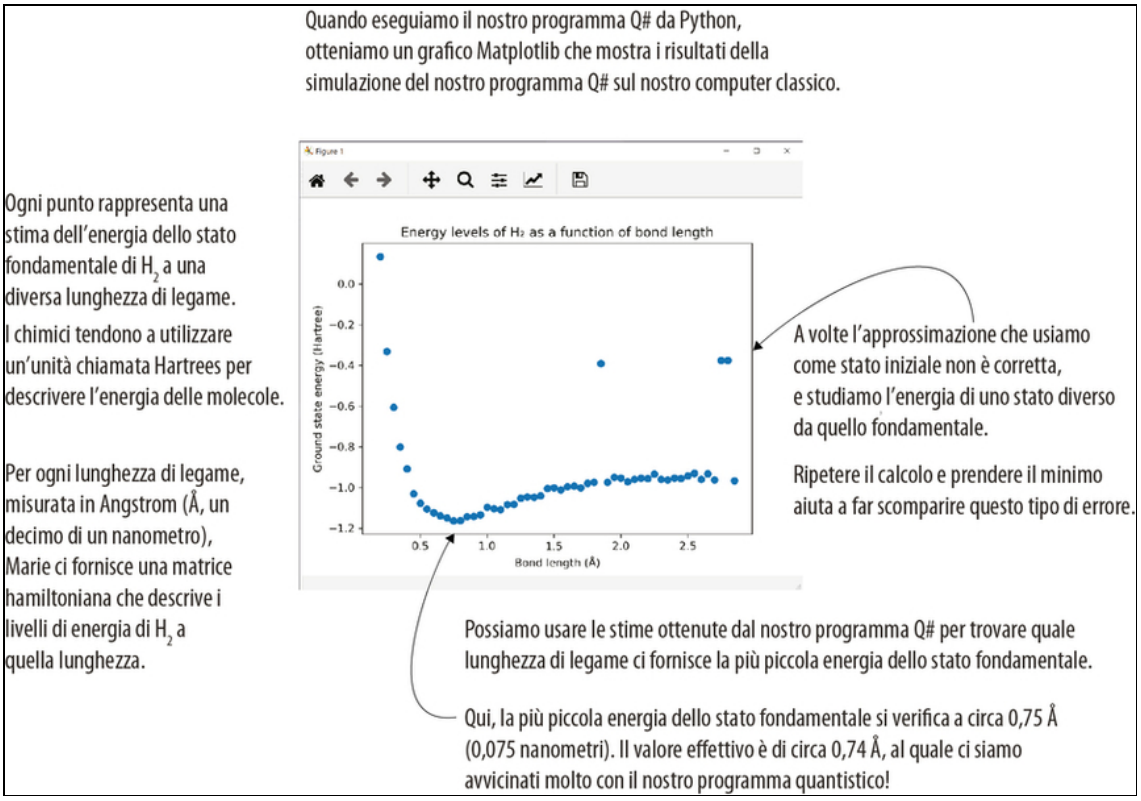


Figura 10.14 Un esempio del grafico prodotto dall'esecuzione di host.py. I dati esatti probabilmente varieranno, ma in generale dovremmo vedere che l'energia minima dello stato fondamentale si ha a circa 0,75 Å sull'asse orizzontale, che è quindi la lunghezza di legame con l'energia più bassa.

Riepilogo

- Una delle applicazioni più interessanti del quantum computing è quella di aiutarci a comprendere le proprietà dei sistemi meccanici quantistici, come le interazioni chimiche.
- Possiamo considerare la meccanica quantistica in molti modi diversi. Usando Python e Q#, abbiamo ragionato in termini di calcolo, ma i chimici e i fisici possono ragionare in termini di un insieme di regole che descrivono il modo in cui i sistemi fisici interagiscono e si comportano.
- Fisici e chimici usano un tipo speciale di matrice, detta hamiltoniana, per prevedere il modo in cui i sistemi meccanici quantistici cambiano nel tempo. Se possiamo simulare una matrice hamiltoniana su un computer quantistico, possiamo simulare i sistemi fisici descritti da quella matrice.
- Un importante caso speciale di una matrice hamiltoniana è un prodotto tensoriale di matrici di Pauli. Queste hamiltoniane descrivono una sorta di generalizzazione delle rotazioni che abbiamo visto in tutto il libro e possono essere simulate usando l'operazione Q# `Exp`.
- Le matrici hamiltoniane più complicate possono essere suddivise in somme di hamiltoniane più semplici, permettendoci di simulare una parte di una matrice hamiltoniana per volta.
- Se alterniamo rapidamente le varie parti della matrice hamiltoniana che simuliamo, possiamo ottenere una migliore approssimazione della simulazione dell'intera hamiltoniana. Allo stesso modo, alternando rapidamente tra le direzioni Nord e Ovest sarà un po' come procedere diagonalmente verso Nordovest.
- Usando modelli chimici (che possiamo ottenere dai nostri amici chimici), possiamo scrivere e simulare le matrici hamiltoniane per

la chimica quantistica. La combinazione di questa simulazione con la stima della fase ci consente di studiare la struttura energetica di diverse sostanze chimiche, aiutandoci a prevederne il comportamento.

Ricerca con computer quantistici

Nel Capitolo 10, abbiamo approfondito la nostra prima applicazione del quantum computing aiutando con la nostra amica Marie a calcolare l'energia dello stato fondamentale di una molecola di idrogeno. Per fare ciò, abbiamo implementato un algoritmo di simulazione hamiltoniano che utilizzava alcune delle tecniche di stima della fase che abbiamo sviluppato nel Capitolo 9.

In questo capitolo vedremo un'altra applicazione del quantum computing: la ricerca dei dati. Quest'area applicativa è sempre un tema caldo nel campo dell'elaborazione ad alte prestazioni e mostra un altro modo in cui possiamo utilizzare le tecniche apprese per costruire un programma quantistico, in questo caso basato sul *phase kickback*. Esaminiamo anche lo strumento di *stima delle risorse* integrato nel Quantum Development Kit (QDK), per vedere come può aiutarci a valutare il dimensionamento dei nostri programmi quantistici, anche quando diventano troppo grandi per essere eseguiti localmente.

Ricerca di dati non strutturati

Supponiamo di voler eseguire ricerche su alcuni dati, per trovare il numero di telefono di un contatto. Se l'elenco dei contatti è ordinato per nome, è abbastanza facile trovare il numero di telefono associato a un nome utilizzando una ricerca binaria.

ALGORITMO 11.1 PSEUDOCODICE PER LA RICERCA BINARIA

1. Scegli una coppia nome/numero di telefono in mezzo all'elenco. Questa coppia è il nostro pivot.
2. Se il nome dell'elemento pivot è quello che stiamo cercando, restituisci il relativo numero di telefono.
3. Se il nome che stiamo cercando precede il nome del pivot, ripeti la ricerca nella prima metà dell'elenco.
4. Altrimenti, se il nome che stiamo cercando segue il nome del pivot, ripeti la ricerca nella seconda metà dell'elenco.

Dati di ogni tipo

In questo capitolo, parleremo molto di ricerca su dati. Questi dati possono presentarsi in molte forme:

- numeri di telefono;
- nomi di cani;
- misurazioni meteorologiche;
- tipi di campanelli per porte.

Ciò che tutti questi elementi hanno in comune è che possiamo rappresentarli sui computer classici come stringhe di bit, utilizzando varie convenzioni relative al modo in cui dovrebbe funzionare quella rappresentazione.

Una tecnica di ricerca di questo tipo può essere eseguita abbastanza rapidamente, ed è perfetta per eseguire ricerche in database pieni di informazioni. Il problema è che nell'Algoritmo 11.1 dipendiamo in modo critico dal fatto che l'elenco di nomi e numeri di telefono è ordinato. Se l'elenco non è ordinato, la ricerca binaria, semplicemente, non funziona.

NOTA

Ora che abbiamo ciò di cui abbiamo bisogno per risolvere problemi più difficili con i computer quantistici, lo scenario di questo capitolo si fa un po' più complicato della maggior parte dei giochi e scenari precedenti. Non preoccupatevi se le cose non saranno subito immediatamente comprensibili:

prendetevi il vostro tempo e leggete più lentamente. Vi promettiamo che ne varrà la pena!

In altre parole, per cercare rapidamente nei dati, dobbiamo applicare loro un qualche tipo di struttura: ordinarli o fare qualche altro tipo di ipotesi che ci consenta di evitare di dover controllare ogni singolo elemento. Se i dati non hanno alcuna struttura, la cosa migliore che possiamo fare è controllarli in modo casuale fino a trovare quello che vogliamo. I passaggi elencati nell'Algoritmo 11.2 mostrano lo pseudocodice del modo in cui potremmo eseguire ricerche in un elenco non ordinato. Potremmo anche essere fortunati, ma in media la ricerca casuale sarà solo due volte più veloce rispetto a controllare ogni singolo elemento.

ALGORITMO 11.2 PSEUDOCODICE PER LA RICERCA DI ELENCHI NON STRUTTURATI

1. Scegli un elemento a caso dall'elenco.
2. Se è quello giusto, restituiscilo, altrimenti, scegli un nuovo elemento e ripeti.

Il fatto che la ricerca in elenchi non strutturati sia difficile è anche alla base di gran parte della crittografia. In tal caso, invece di scrivere l'elenco in modo esplicito, il nostro compito nel cercare di violare un algoritmo di crittografia è provare più chiavi fino a trovarne una funzionante. Possiamo pensare all'operazione di decrittografia come alla definizione *implicita* di un elenco, in cui è presente uno speciale "elemento contrassegnato" che corrisponde alla chiave segreta corretta.

Lo pseudocodice nell'Algoritmo 11.3 potrebbe rappresentare questo compito di decrittografia. L'input casuale che scegliamo è la chiave, che usiamo con la "funzione" o algoritmo di decrittografia per vedere se decifra il messaggio.

ALGORITMO 11.3 PSEUDOCODICE PER LA RICERCA DI INPUT NON STRUTTURATI IN UNA FUNZIONE

1. Scegli un input casuale.
2. Richiama la funzione con quell'input. Se funziona, restituisci l'input.
3. Altrimenti, scegli un nuovo input casuale e ripeti.

Se potessimo eseguire più rapidamente le ricerche in elenchi non strutturati, ciò ci consentirebbe di ordinare database, risolvere problemi matematici e perfino violare alcuni metodi classici di crittografia.

Potrà sorprendere il fatto che se la funzione che definisce il nostro elenco può essere scritta come un'operazione quantistica (usando ciò che abbiamo imparato sugli oracoli nel Capitolo 8), allora possiamo usare un algoritmo quantistico noto come *algoritmo di Grover* per trovare un input molto più velocemente rispetto all'Algoritmo 11.3.

SUGGERIMENTO

Ci stiamo avvicinando alla fine del libro, il che significa che abbiamo l'opportunità di mettere insieme ciò che abbiamo imparato. In particolare, in questo capitolo, useremo ciò che già sappiamo sugli *oracoli*, dal gioco di Viviana e Merlino nel Capitolo 8, per rappresentare l'input all'algoritmo di Grover. Se avete bisogno di un ripasso sugli oracoli, non preoccupatevi; il Capitolo 8 è lì per aiutarvi.

Quando eseguiamo l'algoritmo di Grover, stiamo cercando uno o più valori di una funzione fra tutti i suoi possibili input. Se vogliamo eseguire ricerche su un elenco di dati non strutturato, possiamo immaginare di definire una funzione responsabile della ricerca di una particolare voce nell'elenco. Possiamo quindi cercare tra gli input di questa funzione i valori di output che desideriamo.

Considerate uno scenario in cui dobbiamo decifrare un messaggio in un minuto. Ci sono 2,5 milioni di chiavi possibili, ma solo una di esse decifra il messaggio, e il tempo non è sufficiente per applicarle tutte. Possiamo usare l'algoritmo di Grover e una funzione che rappresenta il problema, come "Questa chiave decifra un determinato messaggio?" per trovare la chiave giusta molto più rapidamente e senza dover provare ogni chiave singolarmente. Questa situazione è molto simile

all'esempio del lucchetto illustrato nella Figura 11.1, dove le chiavi rappresentano i possibili input.

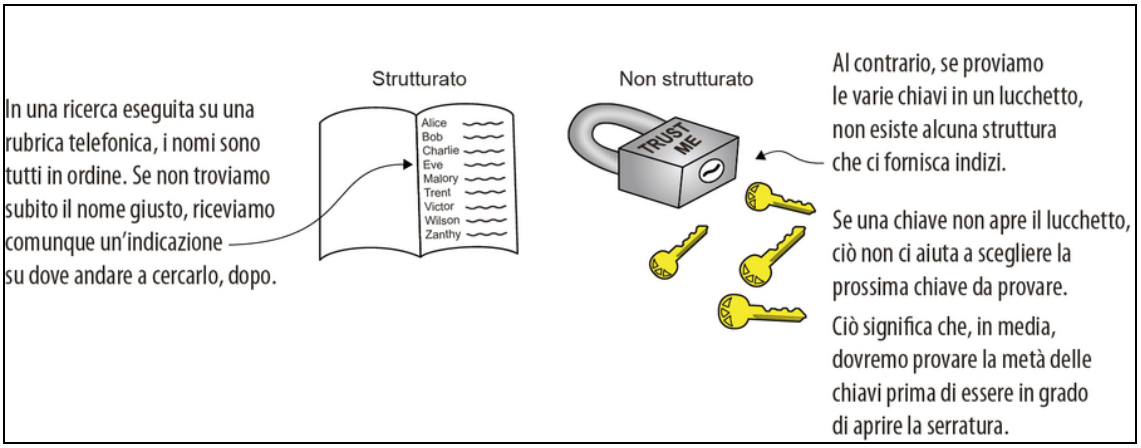


Figura 11.1 Ricerche strutturate e non strutturate. In una tipica rubrica alfabetica, possiamo contare su una struttura dei dati. Nel caso di una scatola di chiavi, possiamo solo continuare a provare "a campione", fino a trovare quella che apre le serratura.

Creeremo la funzione di cui abbiamo bisogno per far sì che l'algoritmo di Grover rappresenti il problema nel modo più preciso quando riparleremo degli oracoli, più avanti nel capitolo; ma è utile ricordare che quando eseguiamo ricerche con l'algoritmo di Grover, stiamo cercando gli *input di una funzione*, non elementi in un elenco di dati. Detto questo, ecco lo pseudocodice dell'algoritmo di Grover.

ALGORITMO 11.4 PSEUDOCODICE PER UNA RICERCA NON STRUTTURATA (ALGORITMO DI GROVER)

1. Alloca un registro di qubit abbastanza grande da rappresentare tutti gli input della funzione che usiamo per la ricerca.
2. Prepara il registro in uno stato di sovrapposizione uniforme: cioè, tutti gli stati possibili hanno la stessa ampiezza. Questo perché, a causa del tipo di problema, non abbiamo alcuna informazione aggiuntiva su quale input sia quello "corretto", quindi questo assetto rappresenta una distribuzione di probabilità uniforme sui dati.
3. Rifletti il registro sullo stato contrassegnato o sullo stato che stiamo cercando. Qui, *riflettere* significa scegliere un determinato stato e invertire il suo segno; vedremo più in dettaglio nel prossimo paragrafo, come

implementare le riflessioni in Q#.

4. Rifletti il registro sullo stato iniziale (sovrapposizione uniforme).

5. Ripeti i Passi 3 e 4 fino a quando la probabilità di misurare l'elemento che stiamo cercando è sufficientemente elevata. Quindi misura il registro. Siamo in grado di calcolare matematicamente il numero ottimale di ripetizioni per massimizzare la risposta corretta.

La Figura 11.2 illustra questi passi.

NOTA

Mentre procediamo in questo capitolo, vedremo che un modo per pensare all'algoritmo di Grover è come a una sorta di rotazione tra stati, che rappresentano il fatto di aver trovato l'elemento corretto: la chiave di decrittografia per il nostro scenario. Se applichiamo i Punti 3 e 4 dell'algoritmo di Grover troppe volte, la rotazione supererà lo stato che stiamo cercando, quindi per l'algoritmo è importante scegliere il numero corretto di iterazioni.

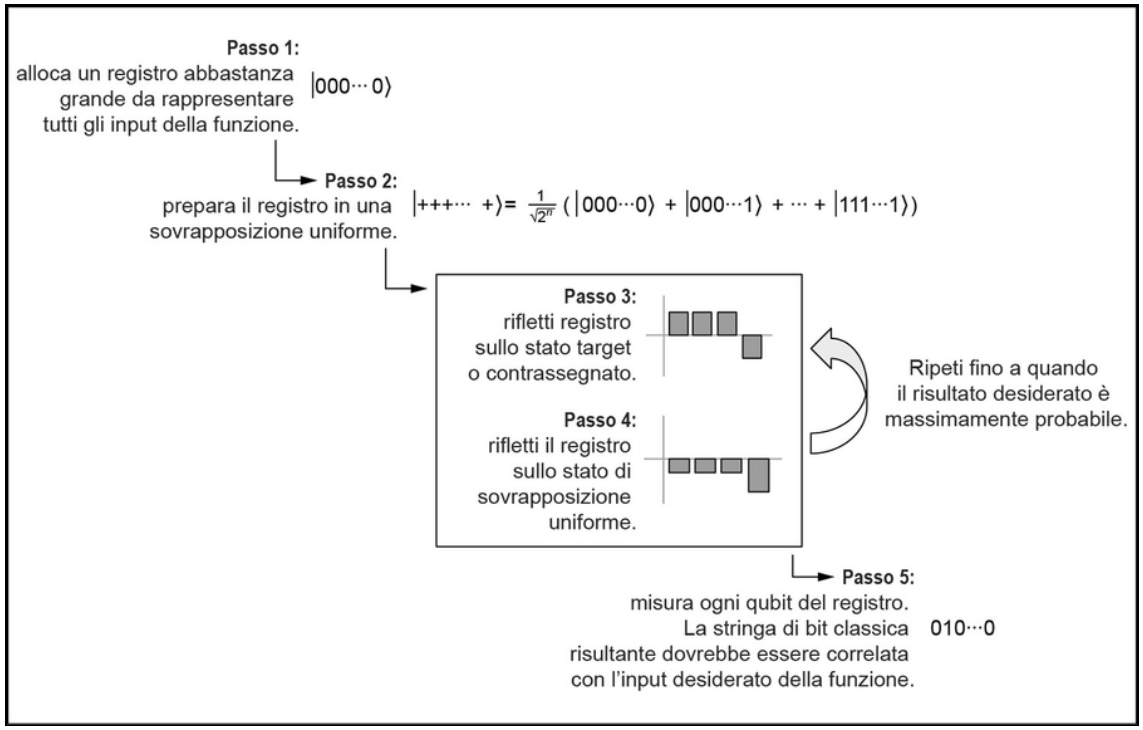


Figura 11.2 I passi dell'algoritmo di Grover, che esamina gli input di una funzione, alla ricerca di un determinato output. Iniziamo allocando un registro di qubit abbastanza grande da rappresentare tutti gli input su cui vogliamo eseguire la ricerca e

poi lo mettiamo in uno stato di sovrapposizione uniforme. Infine, possiamo riflettere lo stato del registro il numero di volte necessario per massimizzare la probabilità di misurare la risposta che stiamo cercando.

La Figura 11.3 mostra un confronto fra il costo della ricerca classica in un elenco non strutturato rispetto all'uso dell'algoritmo di Grover.

SUGGERIMENTO

Un modo per descrivere ciò che è illustrato nella Figura 11.3 è usare un concetto chiamato *complessità asintotica*. In particolare, diciamo che una ricerca classica non strutturata richiede $O(N)$ chiamate di funzione per cercare in N input, mentre l'algoritmo di Grover richiede $O(\sqrt{N})$ chiamate. Non preoccupatevi se questa notazione non vi è familiare; ma se siete curiosi di sapere come valutare gli algoritmi in questo modo, consultate il Capitolo 1 di *Grokking Algorithms* di Aditya Y. Bhargava (Manning, 2016).

Come al solito, cerchiamo di vedere quale potrebbe essere l'aspetto del codice. Il Listato 11.1 è un esempio di un'operazione Q# che utilizza l'algoritmo di Grover per cercare un elemento contrassegnato in un elenco non strutturato. Qui, invece di 2,5 milioni di chiavi, ridurremo l'ambito a 8 chiavi, dove l'elemento o la chiave corretta è indicato da un numero intero compreso tra 0 e 7. Sì, è vero: potremmo risolvere questo problema altrettanto velocemente con un computer classico. Tuttavia, alla fine del capitolo, vedremo come, all'aumentare del numero di chiavi che dobbiamo cercare, il numero di passaggi o calcoli necessari per trovare la chiave corretta si riduca molto quando si utilizza l'algoritmo di Grover.

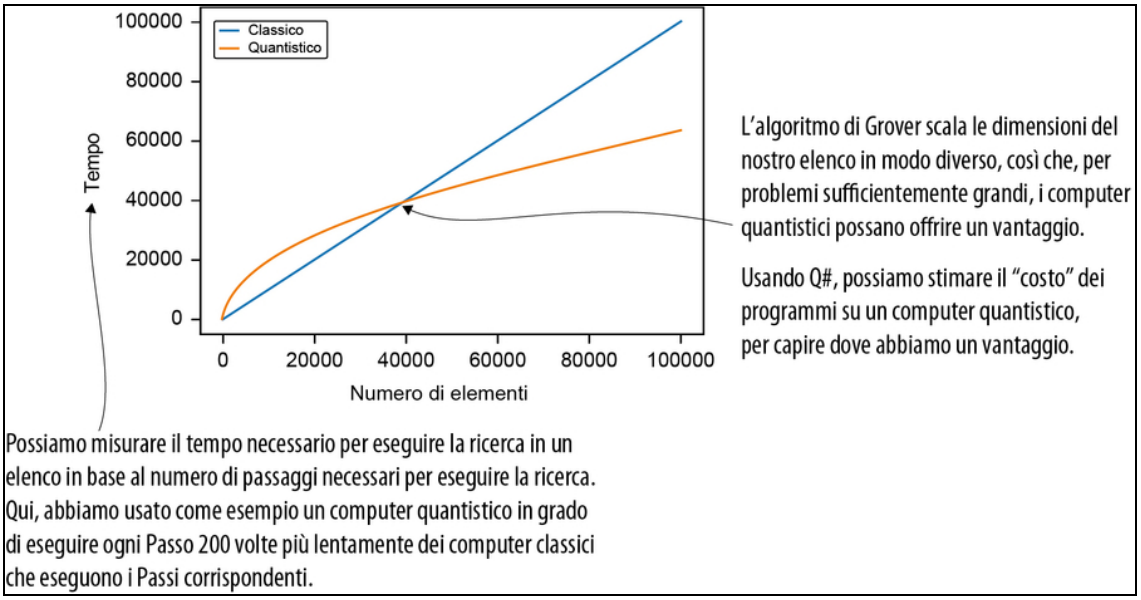


Figura 11.3 Un esempio del tempo impiegato per la ricerca di elenchi non strutturati in computer classici e quantistici. Possiamo vedere che per un numero inferiore di elementi, l'approccio quantistico richiede più tempo; ma man mano che aumentiamo il numero di elementi sui quali eseguire la ricerca, l'approccio quantistico richiede meno tempo.

Inoltre, per il nostro codice di esempio, la funzione che rappresenta l'algoritmo di decrittografia in realtà non esegue alcuna decrittografia; si comporta come se stesse giocando a indovinare e restituisce un valore booleano se riceve la chiave giusta. L'implementazione di un algoritmo di decrittografia per questo capitolo non rientra negli scopi del libro e probabilmente richiederebbe un po' di ricerca. L'obiettivo qui è solo quello di mostrare come l'utilizzo dell'algoritmo di Grover per eseguire ricerche tra gli input di una funzione possa accelerare specifici problemi.

Listato 11.1 operations.qs: codice Q# per la ricerca di Grover.

```
operation RunGroverSearch(nItems : Int, idxMarkedItem : Int) : Unit {
    let markItem = ApplyOracle(idxMarkedItem, _, _); (1)
    let foundItem = SearchForMarkedItem(nItems, markItem); (2)
    Message($"Marked {idxMarkedItem} and found {foundItem}."); (3)
}
```

(1) Possiamo usare l'applicazione parziale per includere l'indice dell'elemento contrassegnato dell'oracolo che forniamo all'algoritmo di ricerca.

(2) Esegue algoritmo di ricerca di Grover su un registro dei tre qubit e fornisce all'oracolo il `markItem` che abbiamo definito in precedenza.

(3) Emette un messaggio per verificare di aver trovato l'elemento corretto.

Eseguendo l'esempio del Listato 11.1, dovremmo ottenere il seguente output:

```
In [1]: %simulate RunGroverSearch nItems=7 idxMarkedItem=6
Out[1]: Marked 6 and found 6
```

Possiamo vedere dall'esecuzione dell'algoritmo di Grover che la chiave di decrittografia che stavamo cercando era quella contrassegnata dal numero 6. Ora, poiché l'operazione `SearchForMarkedItem` è l'elemento centrale di questo esempio, vediamone l'implementazione.

Listato 11.2 operations.qs: scrivere Grover come un'operazione Q#.

```
operation SearchForMarkedItem( (1)
    nItems : Int, (2)
    markItem : ((Qubit[], Qubit) => Unit is Adj) (3)
) : Int { (4)
    use qubits = Qubit[BitSizeI(nItems)]; (5)
    PrepareInitialState(qubits); (6)
    for idxIteration in 0..NIterations(BitSizeI(nItems)) - 1 { (7)
        ReflectAboutMarkedState(markItem, qubits);
        ReflectAboutInitialState(PrepareInitialState, qubits);
    }
    return MeasureInteger(LittleEndian(qubits)); (8)
}
```

(1) Come di consueto, iniziamo definendo una nuova operazione, utilizzando la parola riservata `operation`.

(2) Il primo input di cui necessita la nostra operazione è il numero di elementi nel nostro elenco, `nItems`.

(3) L'input successivo è una rappresentazione del nostro problema di ricerca. Possiamo definire implicitamente il nostro problema di ricerca

da un oracolo che indica se un elemento nel nostro elenco è quello giusto.

(4) Quando avremo terminato la nostra ricerca, avremo un indice della posizione dell'elemento contrassegnato. Definire il nostro output come `Int` ci consente di restituire quell'indice.

(5) Per avviare la ricerca, dobbiamo allocare un registro abbastanza grande da memorizzare un indice del nostro elenco.

(6) Poiché stiamo iniziando una ricerca su un elenco non strutturato, tutti gli elementi sono ugualmente possibili risultati corretti. Lo rappresentiamo preparando una sovrapposizione uniforme su tutti gli indici del nostro elenco.

(7) Il cuore dell'algoritmo di Grover si riduce a riflettere ripetutamente sul nostro stato iniziale e sull'indice dell'elemento che stiamo cercando.

(8) Al termine, la misurazione del nostro registro di qubit ci dice l'indice dell'elemento trovato dall'algoritmo di Grover.

La libreria standard Q# fornisce un'utile operazione, `MeasureInteger`, che interpreta i risultati della misurazione come un numero intero classico. Per utilizzare `MeasureInteger` come nel Listato 11.2, possiamo contrassegnare il nostro registro così che codifichi un numero intero in un registro *little-endian*, utilizzando il tipo `Microsoft.Quantum.Arithmetic.LittleEndian`.

> **NOTA**
>
> Se avete bisogno di un promemoria su che cosa sono i tipi definiti dall'utente e su come usarli, date un'occhiata al Capitolo 9, dove li abbiamo usati per aiutare Lancillotto e Dagonet a giocare il loro gioco e a indovinare l'angolo.

A questo punto del libro, abbiamo quasi tutti i concetti quantistici di cui abbiamo bisogno per comprendere il Listato 11.2. Nel resto di questo capitolo, vedremo come utilizzare quello che abbiamo imparato per implementare un oracolo che possa definire un semplice problema

di ricerca e implementare le due riflessioni che compongono l'algoritmo di Grover per risolvere quel problema.

Amico mio, ho un dubbio sulla tua domanda

Può sembrare un po' artificioso utilizzare un'operazione come `ReflectAboutMarkedState` come oracolo per definire la richiesta per l'algoritmo di Grover. Dopotutto, dal momento che l'indice dell'elemento viene fornito come input di `ReflectAboutMarkedState`, può sembrare che stiamo barando con questo esempio. Detto questo, `SearchForMarkedItem` vede il nostro oracolo come una "scatola nera", non vede i suoi input, pertanto questo modo di gestire l'input non ci permette affatto di imbrogliare.

Usare un semplice oracolo come questo ci aiuta a concentrarci sul funzionamento dell'algoritmo di Grover senza avere a che fare con un oracolo troppo complicato. All'atto pratico, tuttavia, vogliamo utilizzare un oracolo più sofisticato che rappresenta un problema di ricerca più difficile. Per eseguire vere ricerche su un lungo elenco di elementi, per esempio, potremmo usare una tecnica chiamata *quantum RAM* (qRAM) per trasformare il nostro elenco in un oracolo. I dettagli di qRAM non rientrano negli scopi di questo libro, ma esistono alcune ottime risorse online su qRAM e su quanto possa essere costoso utilizzarlo in un'applicazione. In `https://github.com/qsharp-community/qram` trovate un'eccellente introduzione a qRAM e una libreria Q# con cui potete iniziare.

Un'altra area in cui è molto utilizzato l'algoritmo di Grover è la crittografia a chiave simmetrica (quindi non a chiave pubblica, argomento che tratteremo nel Capitolo 12). Per esempio, `https://github.com/microsoft/grover-blocks` fornisce implementazioni di oracoli che rappresentano gli elementi chiave della cifratura AES e LowMC, in modo da valutare questi sistemi utilizzando l'algoritmo di Grover.

Riflessioni sugli stati

Nell'Algoritmo 11.4 e nel Listato 11.2, abbiamo usato ripetutamente due operazioni nel ciclo `for`: `ReflectAboutInitialState` e `ReflectAboutMarkedState`. Vediamo come queste operazioni aiutano a

cercare quell'input della funzione che rappresenta il nostro scenario di decrittografia.

Ciascuna di queste operazioni è un esempio di una riflessione su un determinato stato. Questo è un esempio di un nuovo tipo di operazione quantistica, ma possiamo ancora simularla con una matrice unitaria. Il termine *riflessione su uno stato* significa che abbiamo un registro di qubit, scegliamo un determinato stato in cui potrebbe trovarsi, e se il qubit è proprio in quello stato, ne capovolgiamo il segno (cambiamo la fase di tale stato). Se pensate che questo somigli alle operazioni controllate che abbiamo esaminato in precedenza, avete ragione; in effetti impiegheremo operazioni controllate per implementare queste riflessioni.

Riflessione sullo stato "tutti-1"

Cominciamo con un esempio particolarmente utile: una riflessione sullo stato tutti-1, $|11\cdots1\rangle$. Possiamo implementare questa riflessione usando l'operazione `CZ` (`Z` controllata) che abbiamo già visto nel Capitolo 9.

> **SUGGERIMENTO**
>
> Ricordatevi che il funtore `Controlled` non è solo un blocco `if`, ma può essere usato in sovrapposizione. Per un ripasso del funzionamento del funtore `Controlled`, consultate il Capitolo 9, dove l'abbiamo usato per aiutare Lancillotto e Dagonet nel loro gioco.

Listato 11.3 operations.qs: riflessione sullo stato $|11\cdots1\rangle$.

```
operation ReflectAboutAllOnes(register : Qubit[]) : Unit is Adj + Ctl {
    Controlled Z(Most(register),
    Tail(register)); (1)
}
```

(1) Il funtore `Controlled` ci consente di utilizzare l'operazione `Z` in modo controllato.

Come altre operazioni controllate, anche `Controlled Z` richiede due input: il registro che dovrebbe essere usato per i qubit `control` e il qubit cui verrà applicata l'operazione `Z` se tutti i qubit del registro `control` sono nello stato $|1\rangle$.Nel Listato 11.3, possiamo usare la funzione `Most` di `Microsoft.Quantum.Arrays` per ottenere tutto tranne l'ultimo qubit e `Tail` per ottenere solo l'ultimo qubit.

SUGGERIMENTO

Utilizzando `CZ` insieme a `Most` e a `Tail`, la nostra implementazione funziona indipendentemente dal numero di qubit presenti nel nostro registro. Ciò sarà utile in seguito, poiché potremmo aver bisogno di un numero diverso di qubit per rappresentare i dati nel nostro elenco.

Ricordiamo dal Capitolo 9 che l'operazione `CZ` applica una fase -1 allo stato $|11\cdots1\rangle$ e non fa nulla a ogni altro stato computazionale di base. Ripensando al Capitolo 2, dove abbiamo detto che ogni stato computazionale di base è una sorta di direzione, ciò significa che una sola direzione viene capovolta da `CZ`, mentre tutti gli altri stati di input vengono lasciati come sono. Questo tipo di rappresentazione è il motivo per cui chiamiamo *riflessioni* le operazioni che si comportano in questo modo, sebbene una rappresentazione grafica effettiva sia complicata, a causa del numero di dimensioni che possono essere coinvolte.

Rappresentazione a matrice di CZ

Un modo per rappresentare il fatto che l'operazione `CZ` inverte il segno di un singolo stato di input, come descritto in precedenza, consiste nello scrivere una matrice unitaria che simuli `CZ`. Ecco un esempio con un singolo qubit `control`:

$$U_{\mathrm{CZ}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

Ed ecco un esempio per due qubit `control`:

$$U_{\mathrm{CCZ}} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix}$$

Usando ciò che abbiamo imparato finora sulle matrici unitarie, queste matrici chiariscono che gli stati di input |11⟩ e |111⟩, rispettivamente, vengono invertiti con -1, mentre tutti gli altri stati di input vengono lasciati inalterati (ottengono una fase di +1). Lo stesso modello si ripresenta indipendentemente dal numero di qubit `control` che usiamo con `CZ`.

ESERCIZIO 11.1 DIAGNOSTICA SU CZ

Usate `DumpMachine` per vedere come agisce `CZ` sullo stato di sovrapposizione uniforme |+···⟩.

Suggerimento: Ricordiamo che |+⟩ = *H*|0⟩, pertanto possiamo utilizzare il programma `ApplyToEachCA(H, register)` per preparare |+···+⟩ su un registro che inizia nello stato |00···0⟩.

SOLUZIONI DEGLI ESERCIZI

Tutte le soluzioni degli esercizi di questo libro si trovano nel suo archivio di codice: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Basta andare nella cartella del capitolo in cui ci troviamo e aprire il notebook Jupyter con il nome che menziona le soluzioni degli esercizi.

Riflessione su uno stato arbitrario

Ora che abbiamo la riflessione su $|11\cdots1\rangle$, possiamo utilizzarla per le riflessioni su altri stati. Questo è importante, perché probabilmente non possiamo definire la nostra funzione oracolo che rappresenta l'algoritmo di decrittografia in modo che l'input o la chiave che vogliamo sia rappresentata da input tutti-1. Ricordate, dal nostro codice di esempio, che l'oracolo implementa solo una sorta di decrittografia a tentativi, non un vero algoritmo di decrittografia.

Approfondimento: le riflessioni sono rotazioni?

Data la nostra interpretazione geometrica delle riflessioni, sarebbe naturale considerarle rotazioni di 180° gradi. Ma questo è vero solo perché gli stati quantistici sono vettori di numeri complessi; se avessimo solo numeri reali, non saremmo in grado di ottenere una riflessione utilizzando rotazioni. Possiamo immaginarlo ripensando a come abbiamo descritto gli stati e le rotazioni nei Capitoli 2 e 3: come rotazioni su un cerchio bidimensionale.

Se prendiamo un oggetto bidimensionale, lo capovolgiamo e poi lo poggiamo, non c'è modo di riportarlo com'era senza riprenderlo. D'altra parte, lo spazio tridimensionale ci offre abbastanza spazio per combinare più rotazioni fino a creare una riflessione. Poiché i numeri complessi ci danno un terzo asse per descrivere gli stati dei qubit (vale a dire, l'asse Y), questo ci consente di eseguire le riflessioni di cui abbiamo bisogno nell'algoritmo di Grover.

Il trucco per eseguire riflessioni su stati differenti da quello "tutti-1" consiste nel trasformare qualunque stato su cui vogliamo eseguire la riflessione nello stato "tutti-1", richiamare `ReflectAboutAllOnes`, e poi annullare l'operazione che abbiamo usato per mappare la nostra riflessione sullo stato "tutti-1". Possiamo descrivere qualsiasi stato

partendo dallo tutti-0, quindi abbiamo bisogno di un modo per passare dallo stato tutti-0 allo stato tutti-1 in cui possiamo usare la riflessione che abbiamo appena appreso. Il listato seguente mostra un esempio di preparazione di un registro nello stato tutti-1 dallo stato tutti-0.

Listato 11.4 operations.qs: preparazione dello stato tutti-1.

```
operation PrepareAllOnes(register : Qubit[]) : Unit is Adj + Ctl {
    ApplyToEachCA(X, register); (1)
}
```

(1) L'operazione `ApplyToEachCA` ci permette di applicare il primo input (un'operazione) a ciascun qubit di un registro (il secondo input).

In Q#, tutti i nuovi registri partono nello stato $|00\cdots0\rangle$. Così, nel Listato 11.4, quando applichiamo `X` a ogni qubit appena allocato, ciò prepara il nostro nuovo registro nello stato $|11\cdots1\rangle$.

Per il passo successivo, dobbiamo considerare l'operazione che prepara lo stato su cui vogliamo eseguire la riflessione. Se abbiamo un'operazione `adjoint` (`is Adj`) che prepara un determinato stato su cui vogliamo eseguire la riflessione, allora tutto ciò che dobbiamo fare è *de-preparare* lo stato, preparare lo stato tutti-1, eseguire la riflessione sullo stato tutti-1 ($|11\cdots1\rangle$), de-preparare lo stato tutti-1 e poi *ri-preparare* lo stato sul quale stiamo cercando di eseguire la riflessione.

Perché amiamo così tanto la notazione di Dirac

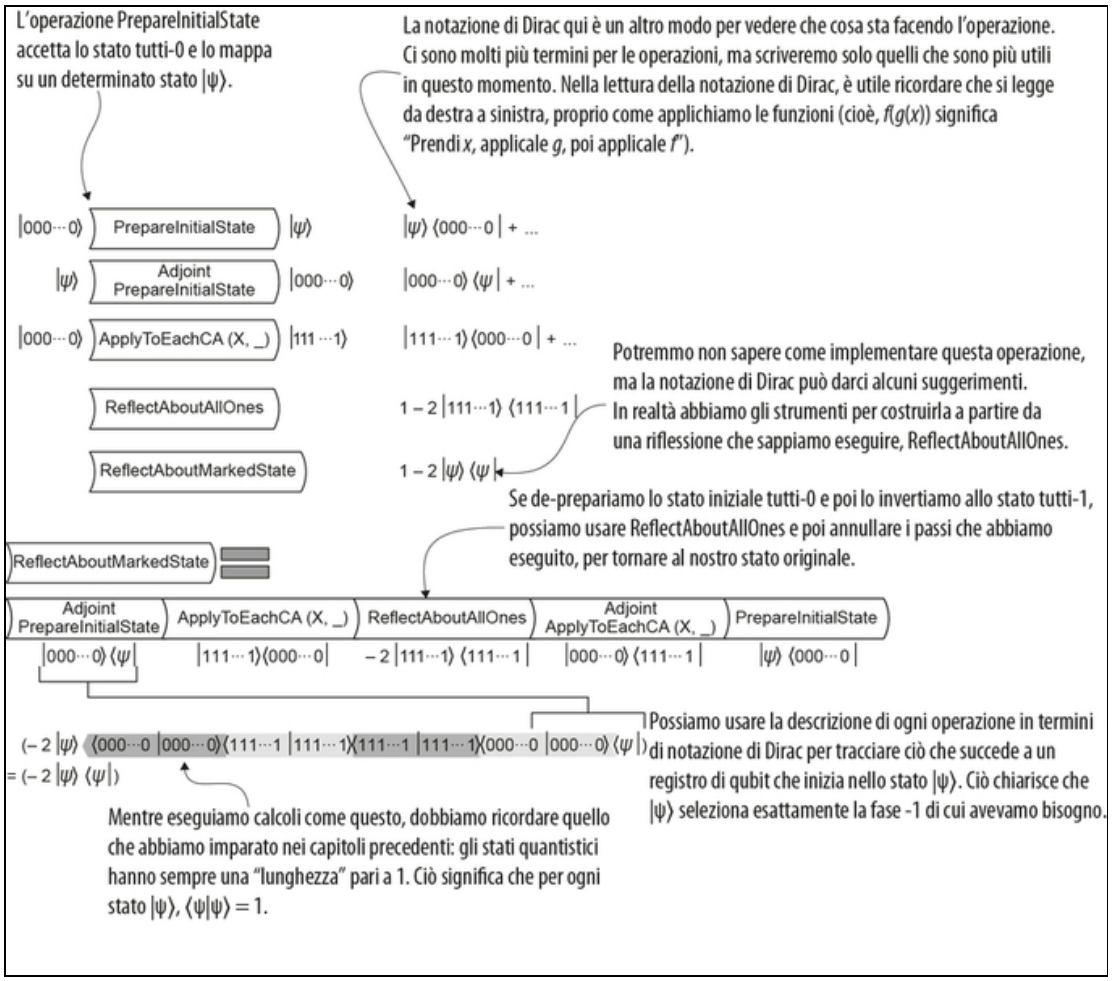
Per comprendere i passi dell'Algoritmo 11.5, è utile considerare che cosa fa al suo stato di input ogni operazione di quella sequenza di passi. Fortunatamente, la notazione di Dirac (che abbiamo incontrato nel Capitolo 2) può aiutare a scrivere come le matrici unitarie trasformano i vari stati, in modo che possiamo capire e prevedere che cosa faranno ai nostri qubit le corrispondenti operazioni Q#.

Per esempio, considerate l'operazione di Hadamard, `H`. Come abbiamo visto in tutto il libro, `H` può essere simulata dalla seguente matrice unitaria:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

Questa matrice unitaria si comporta come una sorta di tabella di verità, dicendoci che *H* trasforma lo stato $|0\rangle$ nello $1 / \sqrt{2} (|0\rangle + |1\rangle)$. Usando la notazione di Dirac, possiamo renderlo più chiaro scrivendo $H = |+\rangle\langle 0| + |-\rangle\langle 1|$. Pensando ai ket ($|\cdot\rangle$) come input e ai bra ($\langle\cdot|$) come output, possiamo leggere la formula come: "L'operazione `H` trasforma $|0\rangle$ in $|+\rangle$ e $|1\rangle$".

La seguente figura mostra come la notazione di Dirac si comporti come una sorta di linguaggio visivo che ci parla degli input e degli output per varie matrici unitarie, rendendo più facile capire il modo in cui funzionano le sequenze di operazioni Q#.



Descrizione delle riflessioni sugli stati con la notazione di Dirac (o bra-ket). Viene descritto ogni passo di ReflectAboutMarkedState e ne viene mostrata la notazione di Dirac. Quando tutti i singoli passi sono collegati, lo stato risultante è -2|ψ⟩⟨>ψ|.

ALGORITMO 11.5 RIFLESSIONE SU UNO STATO ARBITRARIO

1. Usando il funtore `Adjoint`, de-prepara il nostro stato arbitrario, mappandolo sullo stato tutti-0 |00···0⟩.
2. Prepara lo stato tutti-1 |11···1⟩ dallo stato tutti-0.
3. Utilizza `Cz` per eseguire la riflessione su |11···1⟩.
4. De-prepara lo stato tutti-1, mappandolo di nuovo sullo stato tutti-0.
5. Prepara di nuovo il nostro stato, mappando lo stato tutti-0 sul nostro stato arbitrario.

SUGGERIMENTO

Nell'Algoritmo 11.5, i Passi 1 e 5 si annullano vicendevolmente, così come i Passi 2 e 4. Utilizzando ciò che abbiamo imparato sul ragionamento "calze e scarpe", questo rende la procedura dell'Algoritmo 11.5 ideale per implementare la funzione `within`/`apply` di Q#. Per un promemoria del suo funzionamento, consultate il Capitolo 8, dove abbiamo dei blocchi `within`/`apply` per implementare l'algoritmo di Deutsch-Jozsa per Viviana e Merlino.

Poiché non abbiamo alcuna idea di quale sia l'input giusto per il nostro oracolo quando eseguiamo l'algoritmo di Grover, vogliamo iniziare la nostra ricerca dalla sovrapposizione uniforme |+···⟩ per rappresentare il fatto che qualsiasi input potrebbe essere quello giusto. Questo ci dà la possibilità di usare ciò che abbiamo appreso dall'Algoritmo 11.5 per esercitarci nell'implementazione delle riflessioni in Q#. Seguendo i passaggi per eseguire riflessioni sullo stato iniziale, possiamo implementare l'operazione `ReflectAboutInitialState` utilizzata nel Listato 11.2. Il seguente listato mostra come possiamo applicare l'Algoritmo 11.5 in un'operazione Q#.

Listato 11.5 operations.qs: riflessioni su uno stato arbitrario.

```
operation PrepareInitialState(register : Qubit[]) : Unit is Adj + Ctl {
    ApplyToEachCA(H, register); (1)
}
operation ReflectAboutInitialState(
    prepareInitialState : (Qubit[] => Unit is Adj), (2)
    register : Qubit[]) (3)
: Unit {
    within { (4)
        Adjoint prepareInitialState(register); (5)
        PrepareAllOnes(register);
    } apply {
        ReflectAboutAllOnes(register);
    }
}
```

(1) Lo stato di sovrapposizione uniforme rappresenta il fatto che non disponiamo di informazioni preliminari per la nostra ricerca (dopotutto, è un problema di ricerca non strutturata).
(2) Seguendo l'Algoritmo 11.5, per eseguire la riflessione sullo stato iniziale, abbiamo bisogno di fornire un'operazione che lo prepari.
(3) Naturalmente, abbiamo anche bisogno di un registro di qubit cui applicare la nostra riflessione.
(4) Esegue i Passi 1 e 2 dell'Algoritmo 11.5.
(5) Il funtore `Adjoint` indica che vogliamo eseguire l'inversa dell'operazione che prepara lo stato iniziale. In altre parole, se abbiamo iniziato con lo stato iniziale, applicando l'operazione `Adjoint prepareInitialState` ci ritroveremo nello stato $|00\cdots0\rangle$.

Ora che abbiamo il codice per eseguire riflessioni sullo stato iniziale, come possiamo verificare se fa quello che ci aspettiamo? Quando eseguiamo il simulatore della macchina target, possiamo usare comandi come `DumpRegister` per mostrare tutte le informazioni utilizzate per simulare il nostro registro di qubit. La Figura 11.4 mostra l'output di `Dump Register` dopo aver preparato una sovrapposizione uniforme.

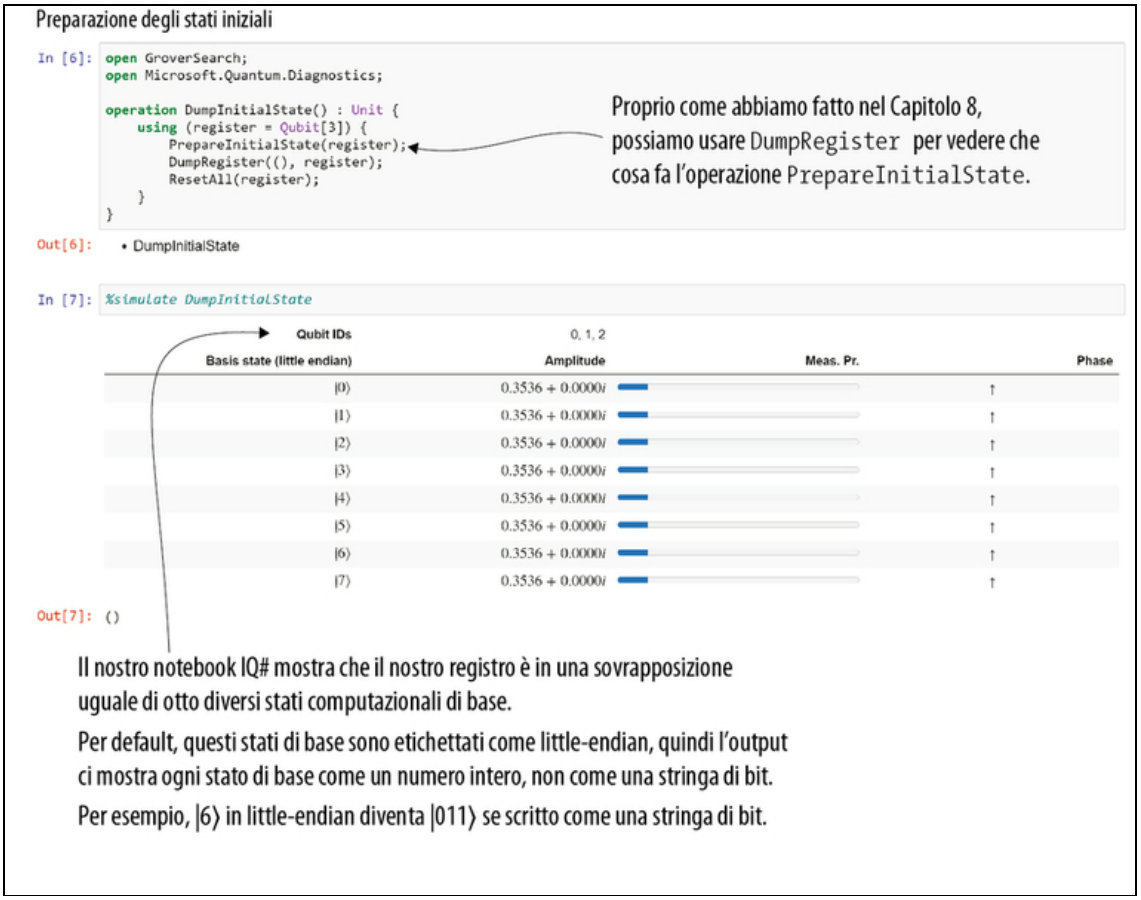


Figura 11.4 Utilizzo di DumpRegister per visualizzare lo stato iniziale preparato dalla nostra operazione PrepareInitialState. Ogni possibile stato base ha la stessa ampiezza (e quindi la stessa probabilità di essere misurato): si parla di sovrapposizione uniforme.

Per eseguire riflessioni sullo stato contrassegnato, l'altra riflessione di cui abbiamo bisogno per l'algoritmo di Grover, dobbiamo usare un approccio leggermente diverso. Dopotutto, non sappiamo come preparare lo stato contrassegnato: questo è esattamente il problema che dobbiamo risolvere con l'algoritmo di Grover! Fortunatamente, ripensando al Capitolo 8, possiamo usare ciò che abbiamo imparato dal gioco di Viviana e Merlino per implementare una riflessione su uno stato anche quando non conosciamo tale stato.

NOTA

Questo è il punto cruciale dell'algoritmo di Grover: possiamo usare il nostro oracolo per eseguire riflessioni sullo stato contrassegnato richiamandolo una volta con la giusta sovrapposizione di input. Come vedremo nel prossimo paragrafo, ciascuna di queste riflessioni ci fornisce alcune informazioni sull'elemento contrassegnato. Al contrario, ogni chiamata di funzione classica può eliminare al massimo un input.

Per vedere come funziona in questo caso, facciamo prima un passo indietro e guardiamo qual è il nostro stato contrassegnato. Poiché il nostro elenco è definito da un oracolo, possiamo scrivere un operatore unitario *O* che ci permette di simulare tale oracolo. Ecco la notazione di Dirac:

$$O|x\rangle \otimes |y\rangle = |x\rangle \otimes \begin{cases} |\neg y\rangle & x \text{ è l'elemento contrassegnato} \\ |y\rangle & \text{altrimenti} \end{cases}$$

Nel Capitolo 8, abbiamo visto che l'applicazione di un'operazione `x` a un qubit in |-⟩ applicava una fase di -1, poiché |-⟩ è un autostato dell'operazione `x`. Usando lo stesso trucco, possiamo scrivere che cosa fa il nostro oracolo quando il qubit flag (il registro |*y*⟩) è nello stato |-⟩:

$$O|x\rangle \otimes |-\rangle = \begin{cases} -|x\rangle|-\rangle & x \text{ è l'elemento contrassegnato} \\ |x\rangle|-\rangle & \text{altrimenti} \end{cases}$$

Questa è esattamente l'operazione di cui abbiamo bisogno per attuare la riflessione! Quindi, seguendo quello che abbiamo imparato nel Capitolo 8, possiamo implementarla nello stesso modo: applichiamo semplicemente il nostro oracolo a un qubit che inizia nello stato |-⟩. Questo oracolo rappresenta l'algoritmo di decrittografia dello scenario, qui semplificato in una funzione che prende come input le chiavi possibili e restituisce solo un booleano che indica se la chiave è corretta. Il prossimo listato mostra un'operazione Q# di esempio che

utilizza questo approccio; ricordate che in Q#, possiamo passare un'operazione come input a un'altra funzione o operazione.

Listato 11.6 operations.qs: riflessione su uno stato contrassegnato.

```
operation ReflectAboutMarkedState(
    markedItemOracle : ((Qubit[], Qubit) => Unit is Adj), (1)
    inputQubits : Qubit[]) (2)
: Unit is Adj {
    use flag = Qubit(); (3)
    within {
        X(flag); (4)
        H(flag);
    } apply{
        markedItemOracle(inputQubits, flag); (5)
    }
}
```

(1) L'operazione per il nostro oracolo di marcatura degli oggetti ha il tipo `((Qubit[], Qubit) => Unit is Adj)`: prende un registro di qubit più un qubit aggiuntivo ed è `Adjoint`-abile.

(2) Il secondo input per eseguire riflessioni sullo stato contrassegnato è il registro cui vogliamo applicare la nostra riflessione.

(3) Abbiamo bisogno di allocare un qubit aggiuntivo (chiamato `flag`) per applicare il nostro oracolo, che corrisponde al registro *y* nelle equazioni precedenti.

(4) Nello stesso modo in cui abbiamo usato le operazioni `H` e `X` nel Capitolo 8 per preparare il qubit di Viviana nello stato $|-\rangle$, usiamo il fatto che $|-\rangle = HX|0\rangle$ per preparare il nostro qubit `flag`.

(5) Applichiamo il nostro oracolo per usare il trucco Deutsch-Jozsa e applichiamo una fase di -1 allo stato contrassegnato dal nostro oracolo.

Nel Listato 11.6, dal momento che questa preparazione è in un blocco `within`/`apply`, Q# mette automaticamente il nostro qubit nello stato $|0\rangle$, annullando le operazioni `H` e `X`. Dopotutto, come abbiamo visto nel Capitolo 8, l'applicazione del nostro oracolo lascia il suo target nello stato $|-\rangle$.

ESERCIZIO 11.2 PREPARAZIONE DEL FLAG

Nel Listato 11.6, possiamo anche scrivere `H(flag); Z(flag);`. L'utilizzo di QuTiP o `Assert OperationsEqualReferenced` o di entrambi dimostra che questi due modi di preparare il nostro qubit `flag` producono la stessa riflessione.

La cosa sorprendente è che abbiamo usato il trucco Deutsch-Jozsa per eseguire riflessioni su uno stato che era stato implicitamente definito dal nostro oracolo! Non abbiamo bisogno di sapere esplicitamente qual era lo stato contrassegnato, per applicare la riflessione: una soluzione perfetta per una ricerca non strutturata.

Nel prossimo paragrafo, vedremo come combinare le riflessioni sullo stato iniziale e sullo stato contrassegnato per riunire il tutto, implementare l'algoritmo di Grover e trovare la chiave.

Implementazione dell'algoritmo di ricerca di Grover

Ora che abbiamo parlato di rotazioni sugli stati e di oracoli, è tempo di mettere tutto insieme per implementare, finalmente, la ricerca non strutturata. Cominciamo esaminando tutti i passi necessari per implementare l'algoritmo di Grover (Figura 11.5).

1. Allocare un registro di qubit abbastanza grande per indicizzare il set di dati su cui stiamo eseguendo la ricerca.
2. Preparare il registro in uno stato di sovrapposizione uniforme: cioè, tutti gli stati possibili con la stessa ampiezza. Questo perché a causa del tipo di problema, non abbiamo informazioni aggiuntive sul set di dati, quindi questo stato rappresenta una distribuzione di probabilità uniforme sui dati.
3. Eseguire una riflessione sullo stato contrassegnato o sullo stato che stiamo cercando.
4. Eseguire una riflessione del registro sullo stato iniziale (sovrapposizione uniforme).

5. Ripetere i Passi 3 e 4 fino a quando la probabilità di misurare l'elemento che stiamo cercando è sufficientemente elevata. Poi misurare il registro. Possiamo calcolare matematicamente il numero ottimale di ripetizioni di cui abbiamo bisogno per massimizzare la probabilità di riottenere l'oggetto contrassegnato.

Il più vicino possibile, ma senza andare oltre

Se applichiamo troppe iterazioni dell'algoritmo di Grover, l'ampiezza dello stato che vogliamo misurare si riduce. Questo perché ogni iterazione, in effetti, è una rotazione; il trucco consiste nel fermare la rotazione al punto giusto. Per elaborare i calcoli trigonometrici per i criteri di arresto, scriviamo lo stato del registro utilizzato nell'algoritmo di Grover come una sovrapposizione degli stati non contrassegnati e contrassegnati. Non tratteremo i dettagli di questa derivazione, ma se siete interessati a saperne di più sulla matematica da impiegare, consultate https://docs.microsoft.com/quantum/libraries/standard/algorithms o il paragrafo 6.1.3 di *Quantum Computation and Quantum Information* di Michael A. Nielsen e Isaac L. Chuang (Cambridge University Press, 2010). Il processo si trova già implementato e pronto all'uso nell'archivio di codice del libro, ma la formula è qui sotto, se volete provare a programmarla in Q#:

$$N_{\text{iterations}} = \text{round}\left(\frac{\pi}{4\arcsin(\frac{1}{\sqrt{2^n}})} - \frac{1}{2}\right)$$

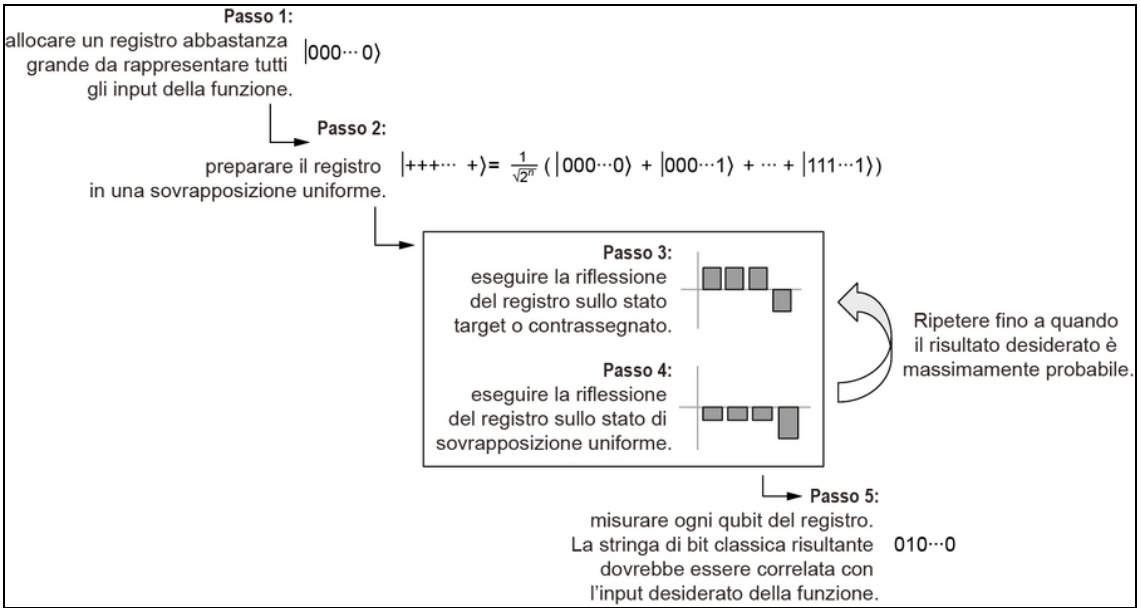


Figura 11.5 I passi dell'algoritmo di Grover che eseguono la ricerca sugli input di una funzione, cercando un determinato output. Iniziamo assegnando un registro di qubit abbastanza grande da rappresentare tutti gli input su cui vogliamo eseguire la ricerca e poi lo mettiamo in uno stato di sovrapposizione uniforme. Infine, possiamo eseguire la riflessione dello stato del registro il numero di volte migliore per massimizzare la probabilità di misurare la risposta che stiamo cercando.

Nel paragrafo precedente, abbiamo sviluppato alcune delle operazioni necessarie per la piena implementazione. Per esempio, abbiamo implementato il Passo 2 con l'operazione `PrepareInitialState` e le riflessioni nei Passi 3 e 4 rispettivamente con le operazioni `ReflectAboutMarkedState` e `ReflectAboutInitialState`. Abbiamo ancora bisogno di una funzione per capire quante volte ripetere i Passi 3 e 4 e di un'implementazione dell'oracolo che identifichi l'elemento che stiamo cercando. Cominciamo con la funzione che aiuta a definire i criteri di arresto per l'algoritmo di Grover.

Listato 11.7 operations.qs: i criteri di arresto per Grover.

```
function NIterations(nQubits : Int) : Int {
    let nItems = 1 <<< nQubits; (1)
    let angle = ArcSin(1. / Sqrt(IntAsDouble(nItems))); (2)
```

```
    let nIterations = Round(0.25 * PI() / angle - 0.5); (3)
    return nIterations;
}
```

(1) `<<<` è l'operatore di scorrimento dei bit a sinistra, utilizzato per calcolare `2nQubits`, che rappresenta il numero massimo di elementi che possono essere indicizzati da un registro quantistico di `nQubits`.
(2) Calcola l'angolo di rotazione effettivo applicato da ciascuna iterazione dell'algoritmo di Grover.
(3) Utilizzando l'effettivo angolo di rotazione e un po' di trigonometria, possiamo calcolare quante iterazioni potranno massimizzare la probabilità che misureremo l'elemento contrassegnato.

Ora che possiamo calcolare quando fermare il ciclo nella nostra implementazione dell'algoritmo di Grover, abbiamo bisogno di un oracolo che possa, dato l'elemento che stiamo cercando e un potenziale elemento dal set di dati, invertire la fase su una parte del nostro registro se l'elemento potenziale è l'oggetto che stiamo cercando. Ai fini di questo esempio, immaginiamo l'oracolo come una sorta di gioco casuale. Se qualcuno sta pensando al numero 4 e ci chiede di indovinare il suo numero, questo è un esempio di una sorta di funzione classica:

$$f(x) = \begin{cases} 1 & x = 4 \\ 0 & \text{altrimenti} \end{cases}$$

Classicamente, non avremmo una strategia migliore che provare diversi input per *f*, fino a provare $x = 4$. Se invece vogliamo provare l'algoritmo di Grover, usando quanto appreso nel Capitolo 8, sappiamo che abbiamo bisogno di un'operazione che rappresenti *f*:

$$U_f\left(|x\rangle \otimes |y\rangle\right) = |x\rangle \otimes \begin{cases} X|y\rangle & x = 4 \\ |y\rangle & \text{altrimenti} \end{cases}$$

È abbastanza facile implementare un'operazione che può essere simulata da *Uf* utilizzando la funzione Q# `ControlledOnInt`, fornita dalle librerie standard Q#. Come il funtore `Controlled`, la funzione `ControlledOnInt` ci consente di controllare un'operazione in base allo stato di un altro registro. La differenza è che mentre `Controlled` esegue il controllo sempre sullo stato tutti-1 $|11\cdots1\rangle$, la funzione `ControlledOnInt` permette di eseguire il controllo su un altro stato, indicato da un numero intero. Per esempio, se `Length(register)` è `3`, allora `(ControlledOnInt(4, X))(register, flag)` inverte lo stato di `flag` ogni volta che `register` è nello stato $|100\rangle$, poiché `4` si scrive `100` in notazione little-endian.

ESERCIZIO 11.3 IL COMPORTAMENTO DI CONTROLLEDONINT

Provate a scrivere che cosa fa `(ControlledOnInt(4, X))(register, flag)` allo stato `register + [flag]` usando la notazione di Dirac (consultate i Capitoli 2 e 4 se avete bisogno di un ripasso) o scrivendo una matrice unitaria che possa essere utilizzata per simulare `(ControlledOnInt(4, X))` agendo su un registro di tre qubit e un qubit `flag`.

Suggerimento: poiché in questo esempio `(ControlledOnInt(4, X))` opera su quattro qubit (tre qubit `control` e un qubit `target`), la matrice unitaria dovrebbe essere 16×16.

Provate a fare la stessa cosa, ma per `(ControlledOnInt(4, Z))`.

Usando la funzione `ControlledOnInt`, possiamo scrivere rapidamente un oracolo che inverte lo stato di un qubit `flag` in base a un input di tale oracolo, come mostrato nel prossimo listato. Qui, il nostro oracolo

dovrebbe invertire il suo qubit `flag` ogni volta che l'input dell'oracolo è nello stato contrassegnato.

Listato 11.8 operations.qs: un oracolo che contrassegna lo stato che vogliamo.

```
operation ApplyOracle(
    idxMarkedItem : Int, (1)
    register : Qubit[],
    flag : Qubit
) : Unit is Adj + Ctl {
    (ControlledOnInt(idxMarkedItem, X))(register, flag); (2)
}
```

(1) Indica l'indice dell'elemento che stiamo cercando come un numero intero (l'esempio qui utilizza tre qubit, quindi possiamo inserire qualsiasi numero intero da 0 a 23-1, ovvero 7).

(2) La funzione `ControlledOnInt` di cui abbiamo appena parlato può applicare una `X` al `flag`, sotto il controllo sul registro di input quando è nell'elemento contrassegnato corretto.

Con questi due nuovi frammenti di codice, possiamo tornare al codice precedente.

Listato 11.9 operations.qs: l'algoritmo di Grover in un'operazione Q#.

```
operation SearchForMarkedItem( (1)
    nItems : Int, (2)
    markItem : ((Qubit[], Qubit) => Unit is Adj) (3)
) : Int { (4)
    use qubits = Qubit[BitSizeI(nItems)]; (5)
    PrepareInitialState(qubits); (6)
    for idxIteration in 0..NIterations(BitSizeI(nItems)) - 1 { (7)
        ReflectAboutMarkedState(markItem, qubits);
        ReflectAboutInitialState(PrepareInitialState, qubits);
    }
    return MeasureInteger(LittleEndian(qubits)); (8)
}
```

(1) Come di consueto, iniziamo definendo una nuova operazione, utilizzando la parola riservata `operation`.

(2) Il primo input è il numero di elementi del nostro elenco.

(3) Proprio come nell'esempio di crittografia, possiamo definire implicitamente il nostro elenco con un oracolo che contrassegna se un

elemento nel nostro elenco è quello giusto.
(4) Quando abbiamo finito la nostra ricerca, abbiamo un indice che punta all'elemento contrassegnato. Il fatto di definire il nostro output come `Int` ci consente di restituire quell'indice.
(5) Per avviare la ricerca, dobbiamo allocare un registro abbastanza grande da memorizzare un indice nel nostro elenco.
(6) Poiché stiamo eseguendo la ricerca su un elenco non strutturato, quando iniziamo la ricerca per la prima volta, tutti gli elementi sono ugualmente probabili.
(7) Il cuore dell'algoritmo di Grover si riduce a eseguire ripetutamente riflessioni sul nostro stato iniziale e sull'indice per l'elemento che stiamo cercando.
(8) Al termine, la misurazione del nostro registro di qubit ci dice l'indice dell'elemento trovato dall'algoritmo di Grover.

La libreria standard Q# fornisce un'utile operazione, `MeasureInteger`, che interpreta i risultati della misurazione come un numero intero classico. Per utilizzare `MeasureInteger`, come nel Listato 11.10, possiamo contrassegnare il nostro registro come un numero intero little-endian, utilizzando il tipo `Microsoft.Quantum.Arithmetic.LittleEndian`.

Ora abbiamo tutto il codice di cui abbiamo bisogno, quindi procediamo con un esempio.

Listato 11.10 operations.qs: un esempio specifico di Grover.

```
operation RunGroverSearch(nItems : Int, idxMarkedItem : Int) : Unit {
    let markItem = ApplyOracle(idxMarkedItem, _, _); (1)
    let foundItem = SearchForMarkedItem(nItems, markItem); (2)
    Message($"Marked {idxMarkedItem} and found {foundItem}."); (3)
}
```

(1) Possiamo usare un'applicazione parziale per includere l'indice dell'elemento contrassegnato nell'oracolo che forniamo all'algoritmo di ricerca.

(2) Esegue l'algoritmo di Grover su un registro di tre qubit e fornisce l'oracolo `markItem` che abbiamo definito in precedenza.

(3) Produce un messaggio per verificare di aver trovato l'elemento giusto.

L'esempio, dovrebbe fornire il seguente output:

```
In [1]: %simulate RunGroverSearch nItems=7 idxMarkedItem=2
Out[1]: Marked 2 and found 2.
```

ESERCIZIO 11.4 CAMBIARE ORACOLI

Provate a cambiare la definizione dell'oracolo per controllare un numero intero diverso. Questo cambia l'output dell'algoritmo di Grover?

Bene: ora possiamo utilizzare un programma quantistico per eseguire ricerche non strutturate. Ma che cosa sta accadendo, in pratica? L'intuizione chiave dalla teoria geometrica che fa funzionare l'algoritmo di Grover è che quando eseguiamo riflessioni su due assi diversi, otteniamo una rotazione. La Figura 11.6 mostra un esempio applicato alle mappe.

La stessa idea funziona per gli stati quantistici. Nell'algoritmo di Grover, le riflessioni dello stato iniziale e contrassegnato si combinano in un'unica rotazione dagli stati non contrassegnati allo stato contrassegnato. Per capire come funziona, possiamo usare le tecniche che abbiamo imparato in tutto il libro per osservare che cosa sta succedendo alle ampiezze di ogni stato del registro mentre svolgiamo i passi dell'algoritmo.

Possiamo vedere dalla Figura 11.7 che ogni ciclo di riflessioni sembra *amplificare* l'ampiezza dello stato che corrisponde all'indice che stiamo cercando. Eseguendo riflessioni tra gli stati non contrassegnati e contrassegnati, possiamo allineare lo stato dei nostri qubit con lo stato contrassegnato che vogliamo trovare.

Possiamo usare queste idee anche in altre applicazioni. L'algoritmo di Grover è un esempio di una classe più ampia di algoritmi quantistici che eseguono quella che viene chiamata *amplificazione di ampiezza*.

Ciò significa che abbiamo notevolmente amplificato le possibilità che quando misuriamo il nostro registro di qubit, la stringa di bit classica dei risultati della misurazione sarà l'elemento che stiamo cercando.

Prima di chiudere questo capitolo, sarà utile parlare di come eseguire ricerche come quella che abbiamo appena implementato su hardware quantistico e non classico.

Algoritmi quantistici deterministici o probabilistici

L'algoritmo di Grover funziona aumentando la probabilità di ottenere la risposta giusta a ogni iterazione. In generale, tuttavia, l'algoritmo di Grover potrebbe non essere in grado di aumentare tale probabilità fino al 100%. Pertanto, diciamo che l'algoritmo di Grover è un esempio di *algoritmo quantistico probabilistico*, il che significa che non abbiamo la garanzia di ottenere ogni volta la risposta che stiamo cercando. Questo non è un problema in pratica, poiché possiamo sempre eseguirlo un piccolo numero di volte per ottenere una probabilità di successo ancora maggiore.

Si può essere tentati di concludere che tutti gli algoritmi quantistici siano probabilistici, ma non è così. Come abbiamo visto nel Capitolo 8, l'algoritmo di Deutsch-Jozsa è un esempio di *algoritmo quantistico deterministico*, in quanto fornisce sempre gli stessi risultati a ogni esecuzione.

ESERCIZIO 11.5 ESPLORAZIONE DELL'ALGORITMO DI GROVER CON DUMPMACHINE

Finora abbiamo imparato molto sulle rotazioni, cosa che può aiutare a comprendere la rotazione applicata da ogni iterazione dell'algoritmo di Grover. Provate a modificare l'implementazione dell'algoritmo di Grover per applicare il doppio delle iterazioni e a usare `DumpMachine` per esaminare lo stato risultante. È quello che vi aspettereste applicando una rotazione due volte?

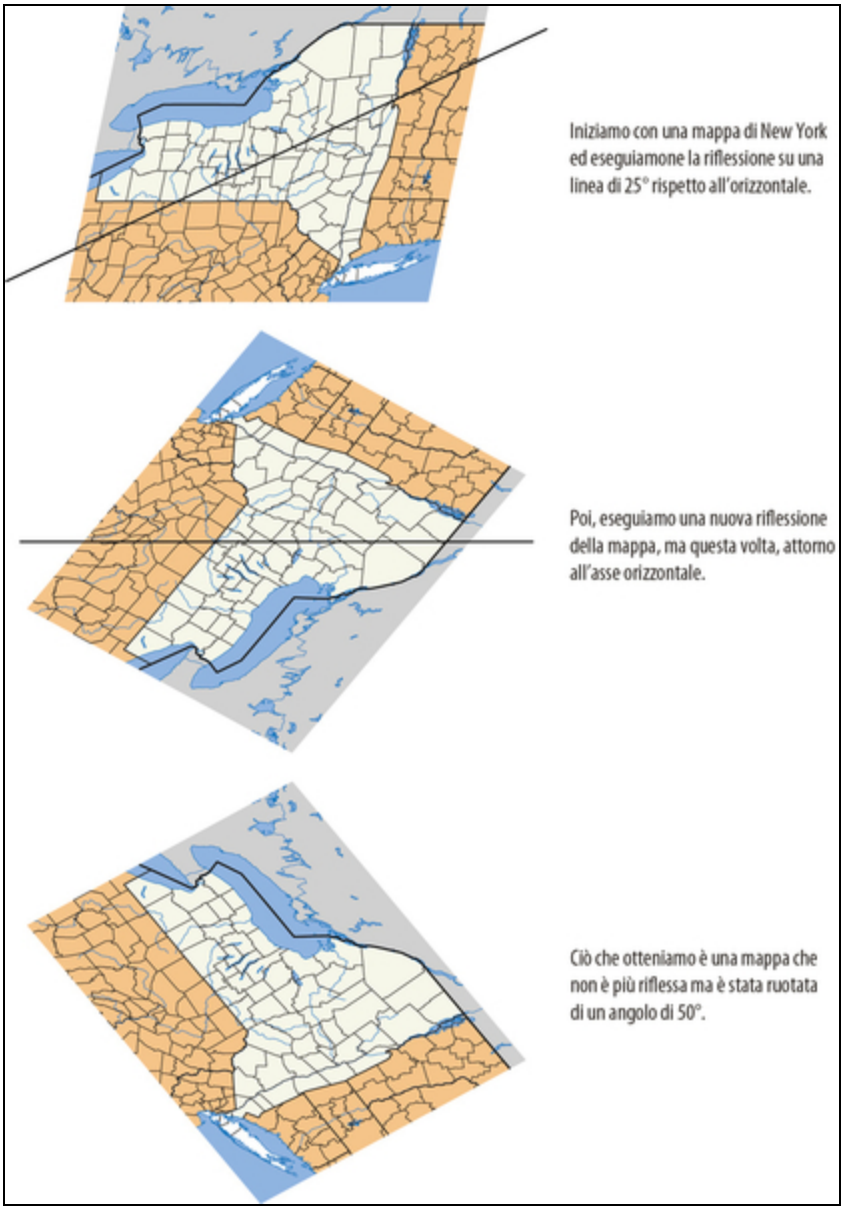


Figura 11.6 Due riflessioni possono effettuare una rotazione. Se la riflessione della mappa è prima su una linea a 25° rispetto al piano orizzontale e poi su una linea orizzontale, avremo una rotazione della mappa di 50° verso il basso rispetto al piano orizzontale.

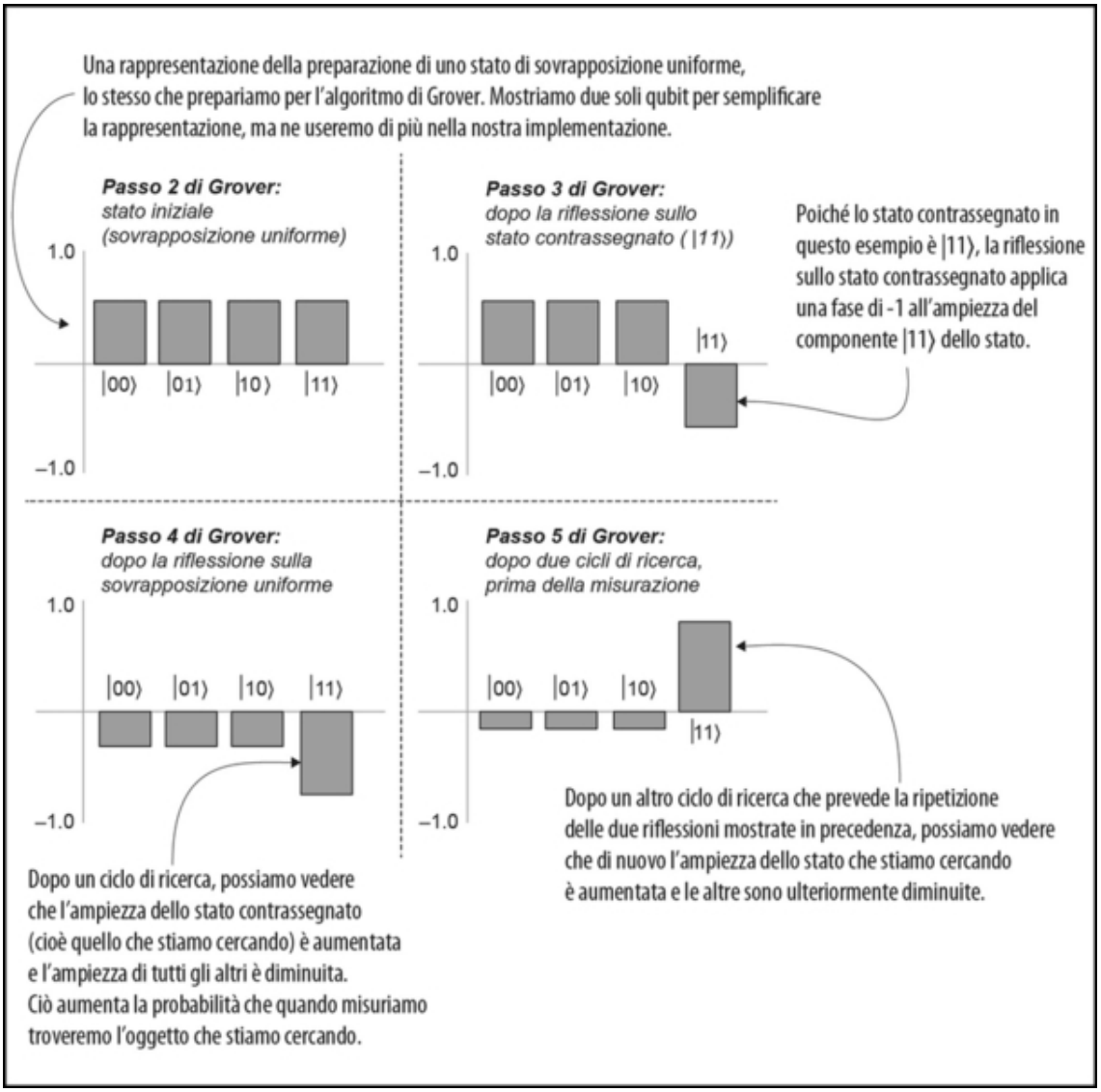


Figura 11.7 Uno schema che mostra come cambiano le ampiezze dello stato del nostro registro di qubit mentre ripetiamo l'algoritmo di Grover. Con le continue riflessioni, alcune ampiezze vengono amplificate e le altre vengono ridotte.

Esempi di amplificazione più generali dell'ampiezza

Insieme alla stima della fase, l'amplificazione dell'ampiezza è una delle tecniche fondamentali utilizzate negli algoritmi quantistici. Nei 25 anni trascorsi da quando l'algoritmo di Grover ha introdotto per la prima volta il concetto di amplificazione dell'ampiezza, è stato sviluppato un numero enorme di varianti per coprire un'ampia gamma di problemi, come quando ci sono più elementi contrassegnati,

quando si desidera ottimizzare una funzione piuttosto che trovare un elemento contrassegnato, o anche quando solo a volte possiamo preparare correttamente lo stato iniziale. Alcuni di questi sono disponibili nel namespace `Microsoft.Quantum.Amplitude.Amplification` nelle librerie standard Q#.

Stima delle risorse

Abbiamo accennato in precedenza, quando abbiamo semplificato il nostro scenario da 2,5 milioni di chiavi a 8, che ci sarebbe stato un vantaggio nell'usare l'algoritmo di Grover con un numero maggiore di chiavi. Quindi, quanto tempo ci vuole, all'atto pratico, per eseguire l'algoritmo di Grover? Questa si rivela essere una domanda piuttosto complicata, e si potrebbero anche scrivere interi libri sull'argomento. In parte, questa è una domanda complicata perché la stima dei requisiti in termini di risorse dipende necessariamente da molte parti del nostro stack di quantum computing.

Per esempio, gli errori sono piuttosto comuni nei dispositivi quantistici, quindi dobbiamo utilizzare la correzione degli errori per proteggere il nostro calcolo mentre viene eseguito. Il metodo di correzione degli errori utilizzato per proteggere i calcoli ha un impatto enorme su ciò che è necessario predisporre per eseguire il nostro programma. Proprio per questo motivo sono state dedicate intere conferenze alla ricerca di codici di correzione degli errori.

Fortunatamente, Q# e il Quantum Development Kit forniscono alcuni degli strumenti di cui abbiamo bisogno per iniziare a capire di che cosa abbiamo bisogno per eseguire diversi programmi quantistici. Invece di eseguire il nostro programma su un simulatore che modella il funzionamento di un vero computer quantistico, possiamo eseguirlo su uno *stimatore di risorse*, che ci dice quante operazione intrinseche dobbiamo richiamare, quanti qubit saranno necessari al nostro programma e quante delle operazioni quantistiche del nostro

programma possono essere richiamate in parallelo. Vediamo quindi un piccolo esempio, utilizzando ciò che abbiamo appreso nel Capitolo 8 sull'algoritmo di Deutsch-Jozsa.

Listato 11.11 Ridefinire l'algoritmo di Deutsch-Jozsa.

```
In [1]: operation ApplyNotOracle(control : Qubit, target : Qubit) : Unit
{ (1)
            within {
                X(control);
            } apply {
                CNOT(control, target);
            }
        }
Out[1]: - ApplyNotOracle
In [2]: open Microsoft.Quantum.Measurement; (2)
        operation CheckIfOracleIsBalanced(oracle : ((Qubit, Qubit) =>
Unit)) : Bool { (3)
            use control = Qubit();
            use target = Qubit();
            H(control);
            within {
                X(target);
                H(target);
            } apply {
                oracle(control, target);
            }
            return MResetX(control) == One;
        }
Out[2]: - CheckIfOracleIsBalanced
In [3]: operation RunDeutschJozsaAlgorithm() : Bool { (4)
            return CheckIfOracleIsBalanced(ApplyNotOracle);
        }
Out[3]: - RunDeutschJozsaAlgorithm
```

(1) La stessa `ApplyNotOracle` che abbiamo visto in precedenza, tranne per il fatto che ora utilizza un flusso `within`/`apply`.

(2) Ricordate che quando si utilizzano notebook Jupyter Q#, dobbiamo aprire i namespace in ogni cella in cui vogliamo usarli.

(3) L'operazione `CheckIfOracleIsBalanced` è la stessa di prima, tranne per il fatto che viene nuovamente utilizzato un blocco `within`/`apply` per sostituire le operazioni `H` e `X` ripetute.

(4) Nei notebook Q#, abbiamo bisogno di un'operazione senza argomenti da usare con i comandi `%simulate` ed `%estimate`.

Quando eseguiamo il comando magico `%estimate` in un notebook IQ#, otteniamo una tabella molto simile a quella mostrata nella Figura 11.8. Questa tabella riporta i tipi di risorse che il Quantum Development Kit stima necessarie per l'esecuzione del nostro programma.

Tabella 11.1 Tipi di risorse tracciate dal comando magico %estimate.

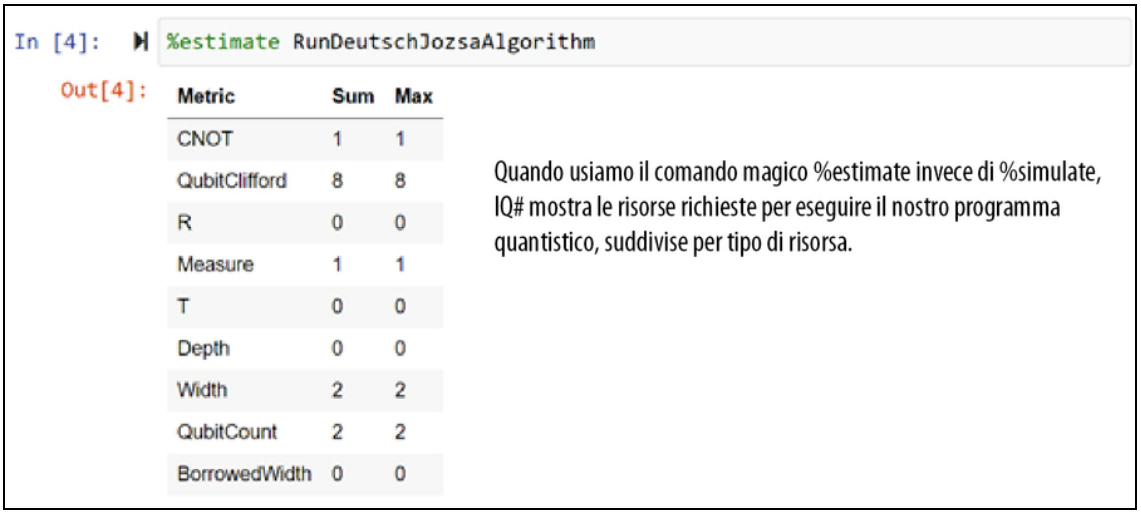
Tipo di risorsa	Descrizione
CNOT	Quante volte viene richiamata l'operazione `CNOT`.
QubitClifford	Quante volte vengono richiamate le operazioni `X`, `Y`, `Z`, `H` e `S`.
R	Quante volte vengono richiamate le operazioni di rotazione mono-qubit.
Measure	Quante operazioni di misurazione vengono richiamate.
T	Quante volte viene richiamata l'operazione `T`.
Depth	Quante operazioni `T` devono essere chiamate di seguito su un singolo qubit.
Width	Quanti qubit richiede il nostro programma.
BorrowedWidth	Quanti qubit il nostro programma deve poter prendere in prestito (una tecnica più avanzata di quella trattata in questo libro).

SUGGERIMENTO

Possiamo anche stimare le risorse da Python. Utilizzate il metodo `estimate_resources` invece del metodo `simulate` che abbiamo imparato nei capitoli precedenti.

Come possiamo vedere dall'esecuzione di `%estimate` nel notebook e nella Tabella 11.1, alcune categorie hanno probabilmente senso, come *Width*, *Measure* e `R`. Altri sono nuovi, come *T* e *Depth*. Non abbiamo

mai visto le operazioni `T` prima d'ora, ma non sono altro che un tipo di operazioni mono-qubit.



Metric	Sum	Max
CNOT	1	1
QubitClifford	8	8
R	0	0
Measure	1	1
T	0	0
Depth	0	0
Width	2	2
QubitCount	2	2
BorrowedWidth	0	0

Figura 11.8 Output dall'esecuzione di %estimate RunDeutschJozsaAlgorithm sul programma del Listato 11.11. Il comando estimate% fornisce un conteggio dei vari tipi di risorse che il nostro dispositivo quantistico (o simulatore) dovrebbe fornire per rendere possibile l'esecuzione del programma. La Tabella 11.1 descrive il significato di ciascuno di questi valori.

Parliamo di T

Come la maggior parte delle operazioni che abbiamo visto finora in questo libro, anche `T` può essere simulata da una matrice unitaria:

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & (1+i)/\sqrt{2} \end{pmatrix}$$

Cioè, `T` è una rotazione di 45° (π / 4) attorno all'asse *Z*.

Un altro modo di considerare l'operazione `T` è come la quarta radice dell'operazione `Z` di cui abbiamo già parlato. Poiché 45° × 4 = 180°, se applichiamo `T` quattro volte di seguito, abbiamo un modo costoso per applicare un'unica operazione `Z`.

In Q#, l'operazione `T` è resa disponibile attraverso `Microsoft.Quantum.Intrinsic.T` e ha il tipo `Qubit => Unit in Adj + Ctl`.

ESERCIZIO 11.6 QUATTRO T FORMANO UNA Z

Utilizzate `AssertOperationsEqualReferenced` per dimostrare che l'operazione `T` applicata quattro volte fa la stessa cosa dell'operazione `Z` applicata una volta. C'è un'altra operazione, `S`, che può essere considerata la radice quadrata di `Z` (una rotazione di 90° attorno all'asse *Z*); verificate che applicare `T` due volte è uguale ad applicare `S` una volta.

Ciò che rende in qualche modo speciali le operazioni `T` e quindi degne di essere considerate nella stima delle risorse è che sono costose da utilizzare con i metodi di correzione degli errori necessari quando si opera su grandi dispositivi quantistici. La maggior parte delle operazioni che abbiamo utilizzato finora fa parte del *gruppo Clifford*: operazioni facili da usare con la correzione degli errori. Come abbiamo già detto, non è negli scopi di questo libro entrare nei dettagli della correzione degli errori; ma in breve, più abbiamo operazioni che non sono di Clifford, più sarà difficile implementare il nostro programma su hardware con correzione dagli errori. Quindi è importante contare il numero di operazioni "costose" (come `T`) da eseguire sul nostro hardware attualmente disponibile.

SUGGERIMENTO

A un livello elevato, il numero di operazioni `T` che devono essere applicate in sequenza (ovvero, che non possono essere eseguite in parallelo) è un'approssimazione piuttosto buona del tempo impiegato da un programma quantistico per essere eseguito su computer quantistici con correzione degli errori. Questo è riportato dallo stimatore di risorse nella metrica *Depth*.

Quindi, quali sono i valori tipici o eccezionali per le risorse che possiamo valutare con lo stimatore di risorse in Q#? Per un programma semplice come `RunDeutschJozsaAlgorithm`, le risorse richieste sono molto modeste. Osservando la Tabella 11.1, tuttavia, ci si deve concentrare sull'operazione `T`, quindi approfondiamola un po' per vedere che cos'è e perché è essenziale per la stima delle risorse. La Figura 11.9 mostra

l'output dalla stima delle risorse necessarie per chiamare l'operazione `CCNOT` di cui abbiamo parlato nel Capitolo 9.

ESERCIZIO 11.7 ESEGUIRE IL RESET DEI REGISTRI

Perché non dobbiamo eseguire il reset del registro dei qubit allocati in `EstimateCcnotResources`, come illustrato nella Figura 11.9?

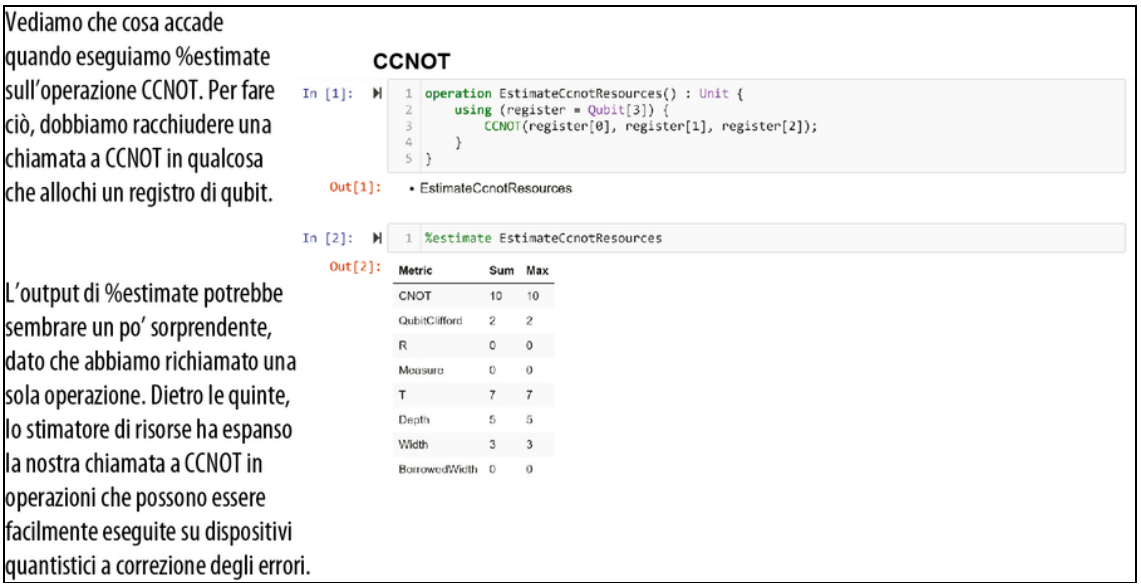


Figura 11.9 L'output dalla stima delle risorse necessarie per richiamare CCNOT. Dal codice, può sembrare che dovrebbe esserci una sola operazione, ma in realtà CCNOT viene scomposta in operazioni implementate più facilmente a seconda della macchina target.

Questo risultato è un po' sorprendente, in quanto il nostro minuscolo programma richiede ben 10 operazioni `CNOT`, 5 operazioni mono-qubit e 7 operazioni `T`, anche se non ne richiamiamo nemmeno una. A quanto pare, è molto difficile applicare direttamente delle operazioni come `CCNOT` in un programma quantistico con correzione degli errori. Quindi, lo stimatore di risorse Q# prima trasforma il nostro programma in qualcosa di più vicino a ciò che verrebbe effettivamente eseguito sull'hardware, utilizzando chiamate a operazioni più semplici, come `CNOT` e `T`.

ESERCIZIO 11.8 SCALING DI T

Come cambia il numero di chiamate a operazioni `T` all'aumentare del numero di qubit `control`? È sufficiente una tendenza approssimativa.

Suggerimento: come abbiamo visto in precedenza, un'operazione NOT-controllato con un numero arbitrario di qubit può essere scritta come `ControlledX(Most(qs), Tail(qs));` utilizzando le funzioni fornite dal namespace `Microsoft.Quantum.Arrays`.

Ciò è utile per stimare le risorse necessarie per eseguire un programma troppo grande per essere simulato su un computer classico. La Figura 11.10 mostra l'output dell'esecuzione dell'algoritmo di Grover su una lista di 20 qubit (circa 1 milione di elementi).

Se lo eseguiamo per varie dimensioni dell'elenco, otteniamo una curva come quella mostrata nella Figura 11.11. Per il nostro scenario con 2,5 milioni di chiavi, il numero di passi quantistici è molto inferiore al costo dei passi classici. Non è tutto, ovviamente, poiché ogni passo su un computer quantistico sarà probabilmente molto più lento di un passo corrispondente su un computer classico; ma questo aiuta a capire che cosa sarebbe necessario per eseguire nella pratica i programmi quantistici.

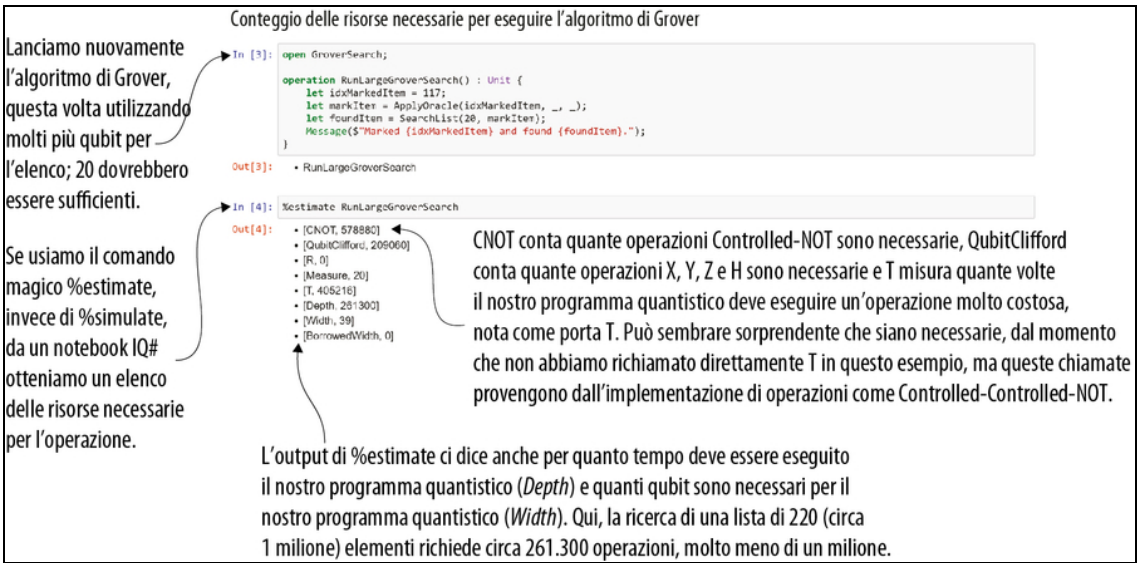


Figura 11.10 Il risultato dell'esecuzione dello stimatore di risorse sull'algoritmo di Grover. Questi conteggi delle risorse chiariscono perché non possiamo simulare direttamente un'istanza di Grover così grande, poiché avremmo bisogno di ben 39 qubit. Tuttavia, possiamo utilizzare questi dati con più dimensioni di ricerca per farci un'idea dell'estensione di scala della nostra implementazione del Grover.

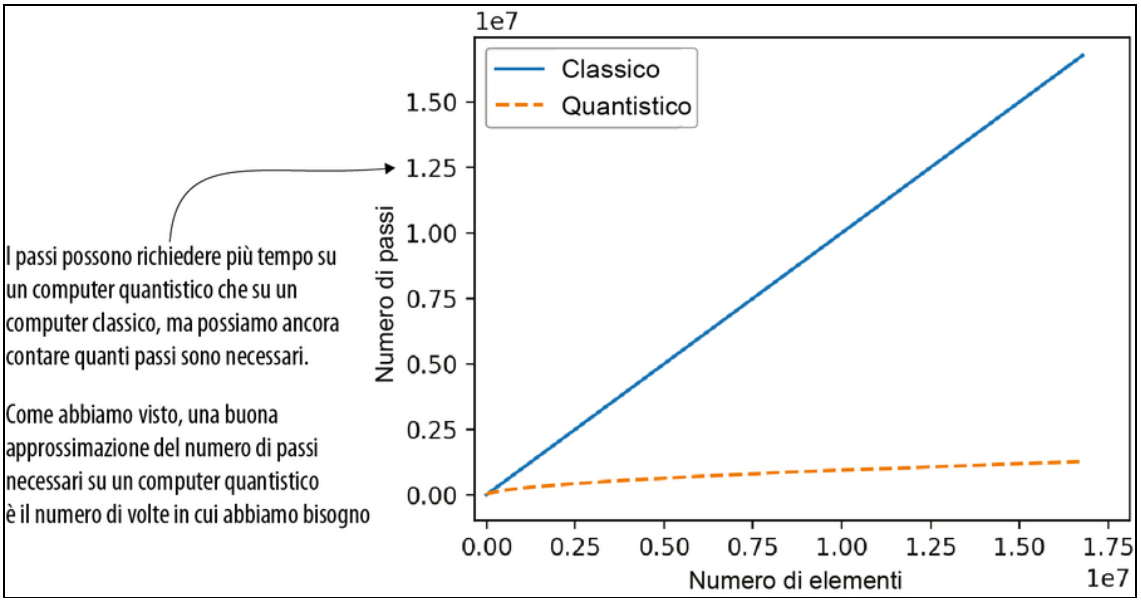


Figura 11.11 Output della stima delle risorse necessarie per eseguire l'algoritmo di Grover per vari elenchi di dimensioni differenti.

Ora abbiamo imparato a combinare gli oracoli che abbiamo conosciuto nel Capitolo 7 con un nuovo tipo di operazione quantistica (le riflessioni) per cercare gli input delle funzioni. Ciò si è rivelato utile per aiutare a trovare più rapidamente le chiavi di decrittografia in caso di vincoli di tempo ristretti.

Nel prossimo capitolo, useremo le nozioni di questo capitolo per rispondere a una delle domande più importanti poste dal quantum computing: quanto tempo impiegherebbe un computer quantistico per violare l'attuale crittografia?

Riepilogo

- Un'altra applicazione per i computer quantistici consiste nella ricerca, in una funzione "opaca", di quell'input che produce l'output desiderato (ovvero un input *contrassegnato*). Possiamo usare l'algoritmo di Grover per eseguire la ricerca utilizzando meno chiamate al nostro oracolo di quanto sia possibile fare con il calcolo classico.
- L'algoritmo di Grover utilizza le riflessioni, operazioni quantistiche in cui la fase di uno stato di input viene capovolta mentre tutti gli altri stati di input vengono lasciati invariati. Possiamo costruire diversi tipi di riflessioni usando le rotazioni insieme ai modelli "calze e scarpe" che abbiamo introdotto nel Capitolo 8.
- Utilizzando le varie rotazioni fornite da Q# insieme a `within` e `apply`, possiamo definire un oracolo che contrassegna un determinato elemento e quindi eseguire la riflessione su tale elemento contrassegnato utilizzando una singola chiamata all'oracolo. Prese insieme, queste tecniche ci consentono di implementare l'algoritmo di Grover.
- Per verificare che l'algoritmo di Grover superi gli approcci classici per problemi sufficientemente grandi, possiamo sottoporre il nostro programma Q# allo stimatore di risorse. A differenza del simulatore fornito con il Quantum Development Kit, lo stimatore di risorse non simula i programmi quantistici, ma conta quanti qubit richiedono e quante operazioni richiamerebbero su un dispositivo quantistico.

Aritmetica con computer quantistici

Nel Capitolo 11, abbiamo utilizzato nell'algoritmo di Grover una tecnica di programmazione quantistica chiamata amplificazione di ampiezza per accelerare la ricerca su set di dati non strutturati. Sebbene quello di Grover non fosse l'approccio di ricerca più efficiente per set di dati piuttosto piccoli, nel momento in cui abbiamo esteso la scala a problemi più grandi, il nostro approccio quantistico ha offerto un chiaro vantaggio. In questo ultimo capitolo svilupperemo le competenze che abbiamo presentato in tutto il libro per affrontare uno dei più famosi algoritmi quantistici: l'algoritmo di Shor. Implementeremo Shor e mostreremo come esso ci può offrire un vantaggio quando cerchiamo di fattorizzare numeri interi di grandi dimensioni. Potrebbe non sembrare un compito particolarmente interessante, ma in realtà la difficoltà di fattorizzare gli interi è alla base di gran parte della nostra attuale infrastruttura crittografica.

Fattorizzazione: quantum computing e sicurezza

Nella Parte I del libro, abbiamo visto come i concetti quantistici possano essere impiegati per inviare dati in modo sicuro utilizzando tecniche come la distribuzione quantistica delle chiavi. Anche senza

QKD, tuttavia, in Internet vengono continuamente condivisi dati critici. Internet viene utilizzato per condividere dati di pagamento, dati sanitari, preferenze sessuali e orientamenti politici. In questo capitolo vedremo come i computer classici possono proteggere la nostra privacy e come il quantum computing cambia il modo in cui decidiamo quali strumenti utilizzare per proteggere i nostri dati.

> **NOTA**
>
> Ora che abbiamo ciò di cui abbiamo bisogno per risolvere i problemi più difficili con i computer quantistici, lo scenario di questo capitolo si fa un po' più complicato rispetto alla maggior parte dei giochi e degli scenari precedenti. Non preoccupatevi se le cose non avranno subito perfettamente senso: prendetevi il vostro tempo e leggete le pagine più lentamente. Vi promettiamo che ne varrà la pena!

Per iniziare, vediamo lo stato dell'arte per la protezione dei dati con i computer classici. Esistono molti problemi, nella matematica classica, alcuni dei quali sono davvero facili da risolvere (per esempio, "Quanto fa 2 + 2?"), mentre altri sono davvero difficili da risolvere (per esempio, "P è uguale a NP?"). Tra questi due estremi, troviamo quei problemi che sarebbero difficili da risolvere, a meno che qualcuno non ci dia un suggerimento, nel qual caso diventano facili. Questi problemi tendono a essere considerati un modo utile per nascondere i dati: per risolvere un calcolo dobbiamo conoscere un segreto condiviso o utilizzare enormi quantità di tempo di calcolo.

> **SUGGERIMENTO**
>
> Nel Capitolo 3, abbiamo parlato di distribuzione quantistica delle chiavi, un ottimo metodo per condividere le informazioni in modo sicuro, basato sulla meccanica quantistica invece che su enigmi. Tuttavia, potremmo non essere sempre in grado di inviare i qubit ai nostri amici, quindi capire il modo in cui utilizzare gli enigmi per comunicare in modo sicuro e privato è ancora un problema importante.

Come vedremo più avanti nel capitolo, la fattorizzazione dei numeri può essere uno di questi enigmi, su cui gli algoritmi crittografici

possono fare affidamento per garantire la sicurezza. Attualmente sono in uso numerosi algoritmi e protocolli crittografici molto importanti, basati sul fatto che è difficile per i computer risolvere enigmi che implicano la fattorizzazione di grandi numeri. Se avete intuito che i computer quantistici possono aiutarci a calcolare la fattorizzazione di grandi numeri, siete sulla strada giusta.

Parliamo dell'*algoritmo di Shor*. Con un computer classico, possiamo ridurre il problema di trovare i fattori di numeri interi alla soluzione di un enigma sulla velocità con cui le funzioni ripetono se stesse quando si utilizza l'*aritmetica modulare* (detta anche *aritmetica dell'orologio*, come vedremo più avanti nel capitolo). Se usiamo l'algoritmo di Shor, stimare la velocità con cui le funzioni si ripetono è esattamente il tipo di enigma che possiamo risolvere facilmente su un computer quantistico. Analizziamo i passaggi di tale algoritmo e poi vediamo un esempio del suo utilizzo.

SCENARIO: FATTORIZZAZIONE DI UN NUMERO INTERO N

Supponiamo di dover cercare di fattorizzare l'intero N; sappiamo in anticipo che N ha esattamente due fattori primi. Utilizzando Q#, implementiamo l'algoritmo di Shor per il fattore N.

Coprìmi e semiprimi

Introduciamo un po' di terminologia utile: due numeri che non condividono fattori comuni diversi da 1 sono *coprìmi*. Per esempio, né 15 né 16 sono numeri primi, ma sono coprìmi l'uno rispetto all'altro.

Allo stesso modo, diciamo che un numero avente esattamente due fattori primi è *semiprimo*. Per esempio, 15 è un numero semiprimo, poiché $15 = 3 \times 5$ e sia 3 sia 5 sono primi. Al contrario, 28 non è un numero semiprimo, poiché $28 = 4 \times 7 = 2 \times 2 \times 7$. I numeri semiprimi vengono frequentemente usati in crittografia, quindi spesso è utile fare questa ipotesi nei nostri scenari.

Possiamo eseguire i passi dell'Algoritmo 12.1 (rappresentato come diagramma di flusso nella Figura 12.1) per utilizzare ciò che abbiamo

appreso sulla stima della fase nei Capitoli 9 e 10 insieme ad alcuni calcoli classici per trovare i fattori di *N*.

L'operatore modulo

Nell'Algoritmo 12.1, abbiamo bisogno di un altro po' di matematica classica: l'operatore modulo, mod. Se non l'avete mai visto prima, non preoccupatevi; lo esamineremo più in dettaglio più avanti nel capitolo.

ALGORITMO 12.1 PSEUDOCODICE PER FATTORIZZARE UN FILE INTERO CON L'ALGORITMO DI SHOR

1. Scegli un intero casuale *g*, che chiamiamo *generatore*.
2. Verifica se il generatore è accidentalmente un fattore, determinando se *g* e *N* sono coprìmi. Se condividono un fattore, allora abbiamo un nuovo fattore di *N*; altrimenti continua con il resto dell'algoritmo.
3. Utilizza la stima della fase iterativa per trovare la *frequenza* della funzione classica $f(x) = g^x \bmod N$. La frequenza ci dice quanto spesso *f* restituisce lo stesso valore all'aumentare di *x*.
4. Utilizza un algoritmo classico noto come *espansione continua delle frazioni* per convertire la frequenza del Passo 3 in un periodo (*r*). Il periodo *r* dovrebbe avere la proprietà che $f(x) = f(x + r)$ per tutti gli input *x*.
5. Se il periodo *r* che troviamo è dispari, torna al Passo 1 e fai una nuova ipotesi. Se *r* è pari, vai al Passo 6.
6. *Adesso, o* $g^r/2 - 1$ o $g^r/2 + 1$ condividono un fattore con *N*.

NOTA

Nell'Algoritmo 12.1, è importante notare che solo il Passo 3 ha a che fare con il quantum computing. Molti dei passi per l'algoritmo di Shor sono più adatti a un hardware classico e dimostrano come verrà probabilmente utilizzato l'hardware quantistico. In pratica, l'hardware e gli algoritmi quantistici funzionano bene come *subroutine* di algoritmi combinati classici e quantistici.

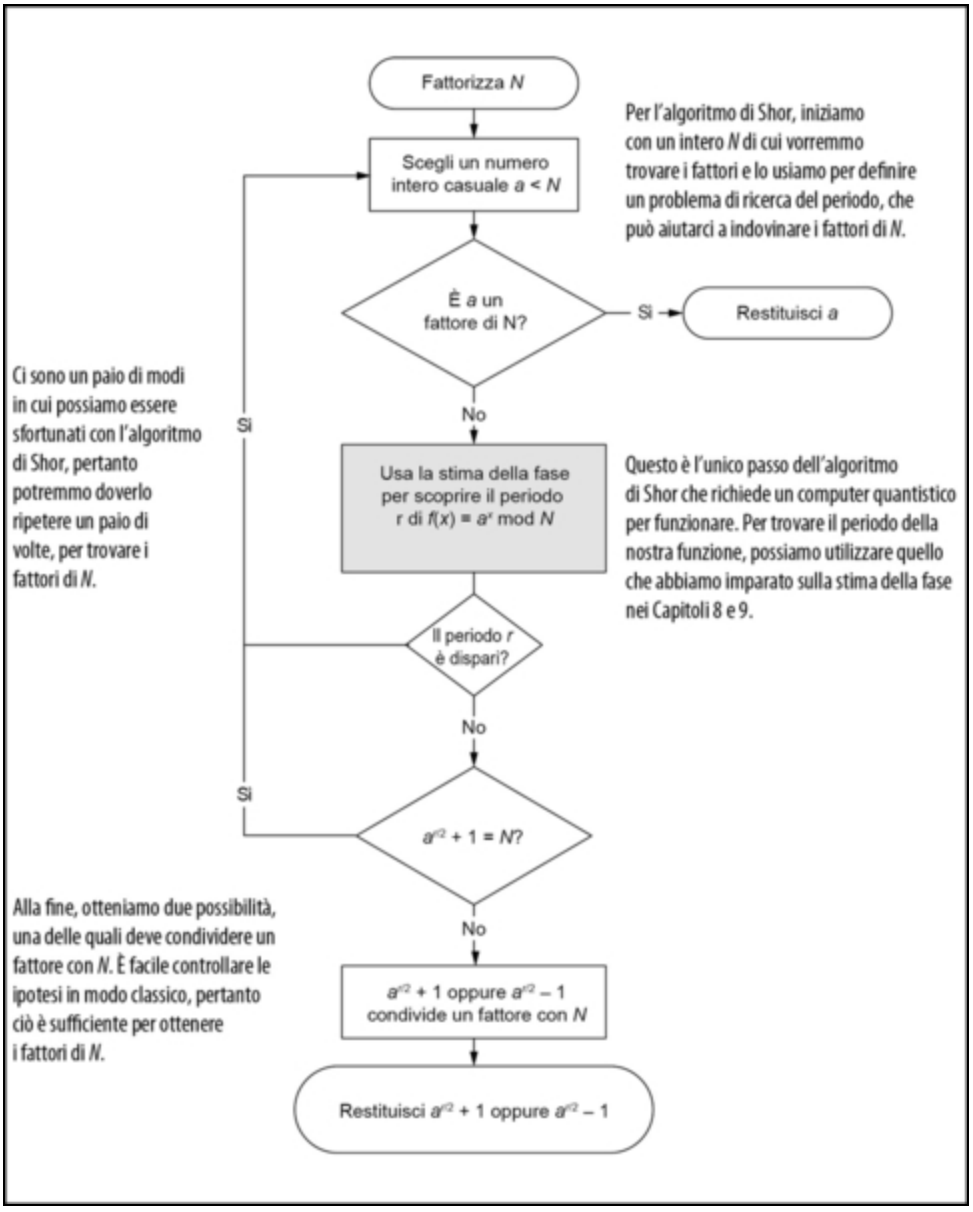


Figura 12.1 Algoritmo di Shor rappresentato come un diagramma di flusso. Per fattorizzare un intero *N*, l'algoritmo di Shor utilizza la stima della fase e un computer quantistico per trovare il periodo di una funzione che prende le potenze di un altro intero a utilizzando l'aritmetica modulare modulo *N*. Dopo un po' di post-elaborazione classica, tale periodo può essere usato per trovare i fattori di *N*.

Ora che abbiamo visto i passi dell'algoritmo di Shor, il Listato 12.1 mostra come potrebbe essere l'implementazione finale. L'operazione `FactorSemiprimeInteger` rappresenta il punto d'ingresso nell'algoritmo:

prende in input il numero intero che vogliamo fattorizzare e restituisce i suoi due fattori.

Listato 12.1 Codice Q# per fattorizzare un intero semiprimo.

```
operation FactorSemiprimeInteger(number : Int) : (Int, Int) {
  if (number % 2 == 0) { (1)
    Message("An even number has been given; 2 is a factor.");
    return (number / 2, 2);
  }
  mutable factors = (1, 1);
  mutable foundFactors = false;
  repeat {
    let generator = DrawRandomInt(3,number - 2); (2)
    if (IsCoprimeI(generator, number)) {
      Message($"Estimating period of {generator}...");
      let period = EstimatePeriod(generator, number); (3)
      set (foundFactors, factors) = MaybeFactorsFromPeriod(generator,
period, number); (4)
    } else {
      let gcd = GreatestCommonDivisorI(number, generator);
      Message(
        $"We have guessed a divisor of {number} to be " +
        $"{gcd} by accident. Nothing left to do."
      );
      set foundFactors = true;
      set factors = (gcd, number / gcd);
    }
  }
  until (foundFactors) (5)
  fixup {
    Message(
      "The estimated period did not yield a valid factor, " +
      "trying again."
    );
  }
  return factors; (6)

}
```

(1) Innanzitutto controlla se ci è stato chiesto di fattorizzare un numero pari, dal momento che 2 deve essere un fattore.

(2) In base al Passo 1 dell'Algoritmo 12.1, scegliamo un numero casuale per definire la funzione periodica che utilizziamo per fattorizzare `number`.

(3) In questo capitolo, impareremo a scrivere un'operazione `EstimatePeriod` per gestire i Passi 3 e 4 dell'Algoritmo 12.1, usando ciò

che abbiamo imparato sulla stima della fase.

(4) Una volta ottenuto il periodo, possiamo utilizzare i Passi 5 e 6 dell'Algoritmo 12.1 per indovinare i fattori di `number`; scriveremo `MaybeFactorsFromPeriod` più avanti nel capitolo.

(5) Se qualcosa va storto (per esempio, il nostro generatore ha un periodo dispari), usiamo un ciclo `repeat`/`until` per riprovare.

(6) Restituisce i due fattori di `number` che abbiamo trovato con il nostro programma quantistico.

> **La fortuna non è tutto, ma può aiutare!**
> Nel Listato 12.1, usiamo `IsCoprimeI` per controllare se `generator` è un fattore di `number`, prima di procedere con il resto dell'algoritmo di Shor. Se siamo davvero fortunati, allora `generator` è già un fattore, nel qual caso non abbiamo bisogno di un computer quantistico per fattorizzare `number`.
> Anche se possiamo essere fortunati abbastanza spesso nei piccoli esempi che possiamo simulare su un laptop o un desktop, man mano che cresce `number`, diventa sempre più difficile indovinare i fattori giusti per puro caso; pertanto l'algoritmo di Shor si rivela spesso molto utile.

Poiché questo è l'ultimo capitolo del libro, conosciamo già tutti i concetti *quantistici* di cui abbiamo bisogno per capire che cosa sta succedendo nel Listato 12.1; le uniche cose che mancano sono le parti classiche, che collegano ciò che abbiamo imparato finora al problema della scomposizione in fattori di numeri semiprimi, nonché alcune parti utili delle librerie Q#. Come accennavamo prima, solo un passaggio qui utilizza la tecnologia quantistica e lo fa creando un oracolo che implementa la funzione classica che vogliamo conoscere. Usando uno stato di sovrapposizione, applicando l'oracolo ed eseguendo la stima della fase, possiamo apprendere le proprietà della funzione classica, in questo caso il periodo. Nel resto del capitolo, esamineremo in dettaglio l'Algoritmo 12.1 e tratteremo gli ultimi elementi di cui abbiamo bisogno per eseguire l'algoritmo di Shor. Il

primo argomento che dobbiamo affrontare per capire l'Algoritmo 12.1 è una parte della matematica classica chiamata aritmetica modulare.

Il legame fra matematica modulare e fattorizzazione

Un modo per trovare enigmi che possano essere impiegati in contesti di sicurezza consiste nel considerare il funzionamento dell'aritmetica modulare. A differenza della normale aritmetica, nell'aritmetica modulare, tutto si ripete come nelle ore di un orologio. Per esempio, se qualcuno ci chiedesse che ora viene due ore dopo le undici, possiamo certamente rispondere "Le ore tredici", ma molto più probabilmente risponderemo "L'una".

Usando l'aritmetica modulare, possiamo catturare questa idea dicendo che 11 + 2 = 1 mod 12. In quell'equazione, *mod 12* indica che vogliamo che tutto ciò che va oltre il 12 riprenda dall'inizio (Figura 12.2).

Quando si usa l'aritmetica in questo modo, può essere difficile capire dove hanno avuto origine i calcoli. Se stiamo lavorando con i normali numeri reali, per esempio, è facile calcolare b se ci vengono dati a e a^b; possiamo prendere il logaritmo di a^b per trovare b. Se proviamo a risolvere lo stesso problema in aritmetica modulare, la cosa può diventare complicata. Per esempio, se calcolate modulo 21, le potenze di 5 sono 1, 5, 4, 20, 16, 17, 1, …. A prima vista, 5, 4 e 16 non sembrano potenze dello stesso numero, tanto meno in ordine crescente, ma se sappiamo di aver calcolato un esponente modulo 21, abbiamo più possibili punti di partenza, indizi da controllare.

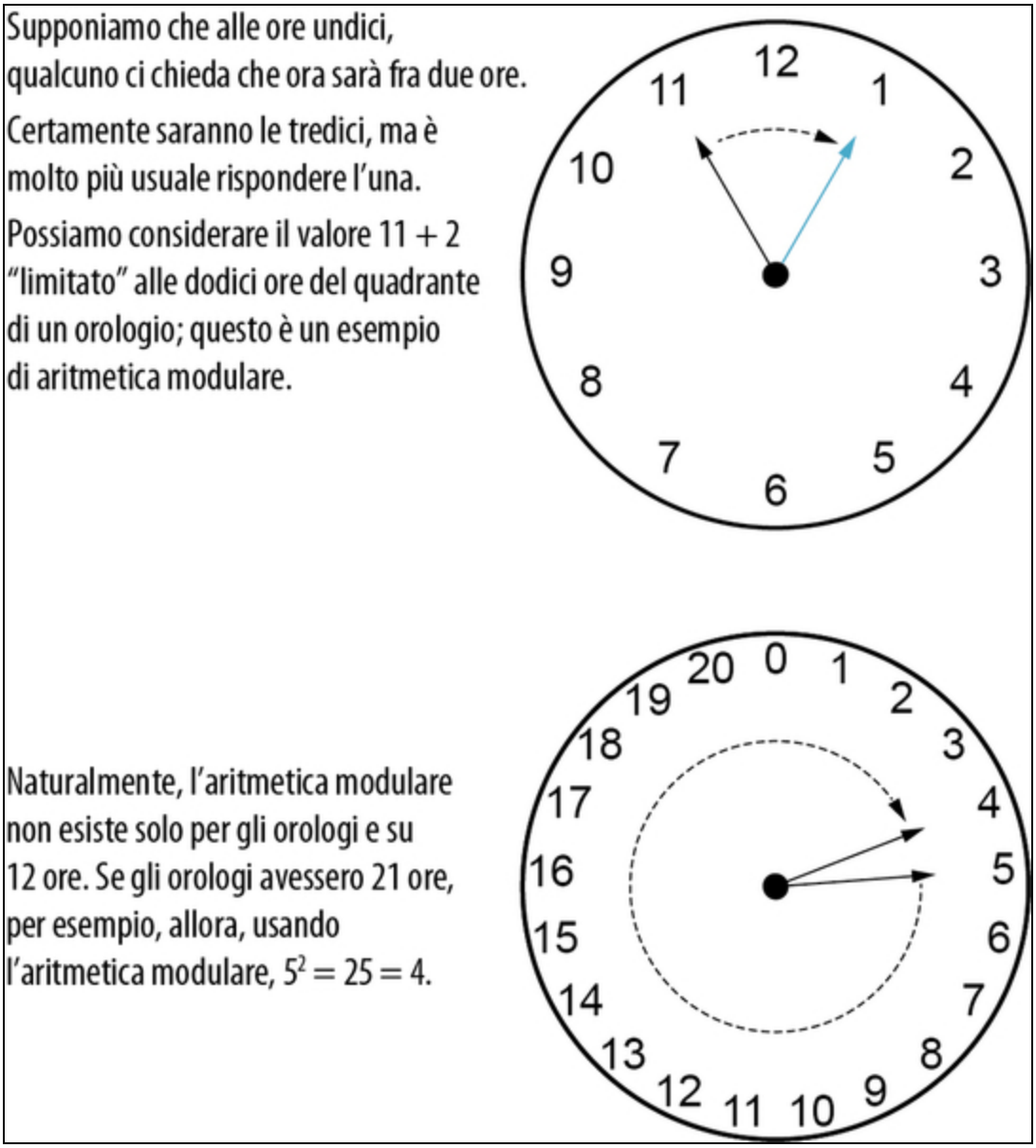


Figura 12.2 Il quadrante dell'orologio e l'aritmetica modulare. Quando si sommano e moltiplicano dei numeri "mod *N*", si può considerare una normale sequenza numerica come se fosse avvolta attorno al quadrante di orologio di *N* ore.

ESERCIZIO 12.1 POTENZE DI 11

Quali sono le potenze di 11 calcolate modulo 21? Quanto tempo ci vuole per tornare a $11^0 = 1$?

Cambia qualcosa se si calcola il modulo 21 alla fine o a ogni passaggio?

Suggerimento: Python o Q# sono perfetti per questi calcoli, poiché entrambi definiscono l'operatore modulo, `%`.

SOLUZIONI DEGLI ESERCIZI

Tutte le soluzioni degli esercizi di questo libro si trovano nel suo archivio di codice: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Basta andare nella cartella del capitolo in cui ci troviamo e aprire il notebook Jupyter con il nome che menziona le soluzioni degli esercizi.

L'osservazione che trovare la potenza b dato a^b mod N è difficile ci dà già un enigma che possiamo usare per nascondere alcuni dati! Questo enigma è comunemente chiamato *problema del logaritmo discreto*. Se Alice vuole condividere un segreto con noi, possiamo iniziare concordando pubblicamente un numero piccolo come $g = 13$ e un numero grande come $N = 71$. Quindi ognuno di noi sceglie un numero segreto a caso: supponiamo che Alice scelga $a = 4$ e noi scegliamo $b = 5$. Alice quindi ci invia g^a mod $N = 19$, e noi restituiamo g^b mod $N = 34$. Se calcoliamo $(g^a)^b$ mod $N = 195$ mod $71 = 45$ e Alice calcola $(g^b)^a$ mod $71 = 344$ mod $71 = 45$, otteniamo entrambi lo stesso numero, ma per risolverlo un estraneo dovrebbe invece risolvere l'enigma del salto del quadrante che abbiamo appena visto (vedi Figura 12.3). Poiché $g^{ab} = 45$ è un numero che noi e Alice (e nessun altro) conosciamo, possiamo usare g^{ab} come una chiave per nascondere i nostri messaggi usando ciò che abbiamo imparato nel Capitolo 3.

Figura 12.3 Utilizzo del problema del logaritmo discreto come un enigma per nascondere messaggi segreti. Qui, i messaggi che condividiamo con Alice sono protetti fintantoché è difficile per un estraneo calcolare operazioni aritmetiche modulari come la funzione esponenziale.

Avvertenza: non provateci a casa

Molti dei vincoli tecnici di questo protocollo vanno ben oltre gli scopi di questo libro. Una scelta sbagliata di g e N può minare la sicurezza offerta da questa tecnica, rendendo del tutto banale l'intervento di un utente malintenzionato. È

anche molto facile introdurre bug, quindi considerate questo ragionamento un esempio puramente concettuale.
Se siete interessati a saperne di più sugli aspetti pratici dell'utilizzo di questo tipo di enigma per proteggere i dati, *Cryptography Engineering* di Niels Ferguson, Bruce Schneier e Tadayoshi Kohno (Wiley, 2010) è un ottimo testo di riferimento.

A differenza della tecnica QKD, questo modo di condividere dati segreti (noto come *protocollo Diffie-Hellman*) si basa sul presupposto che l'enigma che noi e Alice abbiamo utilizzato sia difficile da risolvere senza un indizio, cui il nostro hacker non ha accesso. Se qualcuno potesse risolvere in modo efficiente enigmi come $g^a \bmod N$ per determinati g e N, i nostri dati diventerebbero subito pubblici.

Un altro enigma comunemente utilizzato per proteggere i dati è l'*algoritmo RSA*, che utilizza la matematica classica più avanzata per creare un enigma dalla fattorizzazione di numeri interi veramente grandi. Proprio come possiamo violare Diffie-Hellman risolvendo $g^a \bmod N$, possiamo violare RSA risolvendo $N = pq$ per p o per q dato solo N. Nell'enigma RSA, chiamiamo N una chiave pubblica e i fattori p e q la chiave privata; se riuscissimo a fattorizzare facilmente N, potremmo ottenere l'accesso alle chiavi private date le sole chiavi pubbliche. Detto questo, possiamo definire meglio tale scenario.

SCENARIO: VIOLAZIONE DELL'ALGORITMO RSA

Supponiamo di conoscere la chiave pubblica N. Utilizzando Q#, implementare l'algoritmo di Shor per fattorizzare N per recuperare le chiavi private p e q.

Provando e riprovando

Questo scenario può sembrare un po' malevolo rispetto a quelli dei capitoli precedenti. Tuttavia, è essenziale comprendere il funzionamento degli attacchi agli strumenti e ai protocolli che utilizziamo per mantenere i nostri dati al sicuro, in modo da poter adattare il nostro approccio di conseguenza. Se utilizziamo un algoritmo crittografico come RSA per proteggere i dati, l'implementazione di attacchi quantistici su quell'algoritmo può aiutarci a capire quali dovrebbero essere le caratteristiche di un dispositivo quantistico di cui i nostri aggressori avrebbero

bisogno per violare i nostri dati. Dopotutto, le affermazioni in circolazione vanno da un “Non ci sono problemi con l’RSA” a un “Dovremmo essere terrorizzati”; per capire la situazione reale occorre scoprire di quali risorse avrebbe bisogno un utente malintenzionato.

In altre parole, esplorando uno scenario che ci pone per un momento nei panni di un “cattivo”, possiamo capire quanta potenza di calcolo quantistica sarebbe necessaria per violare con successo l’algoritmo RSA. Torneremo su questo punto alla fine del capitolo, ma per ora cerchiamo di calarci nei panni di un hacker. Capire il modo in cui i computer quantistici potrebbero attaccare la crittografia classica è un ottimo scenario per esercitarsi ad applicare le nostre abilità di quantum computing.

Tuttavia, è meglio essere cauti nell’usare questo esempio. L’impatto del quantum computing sulla sicurezza delle informazioni dipende dalle ipotesi che facciamo sugli algoritmi classici, dai miglioramenti apportati agli algoritmi quantistici, dai progressi nello sviluppo dell’hardware quantistico, dal tempo disponibile per violare la segretezza e da molte altre preoccupazioni simili. Sfortunatamente, la trattazione di tutti questi aspetti al fine di prendere decisioni responsabili sul modo migliore in cui utilizzare la crittografia richiederebbe molto spazio, quindi considerate lo scenario RSA trattato in questo capitolo come un esempio e non un’analisi completa dell’argomento.

Anche se Diffie-Hellman e RSA sembrano molto diversi, possiamo usare un po’ di matematica classica per trasformare l’enigma di fattorizzazione RSA in un altro esempio di aritmetica modulare (l’enigma Diffie-Hellman). Questo problema può essere risolto facilmente su un computer quantistico utilizzando ciò che abbiamo imparato nel Capitolo 10. Esaminiamo un rapido esempio dell’uso dell’algoritmo di Shor per fattorizzare un intero piccolo, in modo da poter vedere all’opera tutte le sue parti.

Esempio di fattorizzazione con l’algoritmo di Shor

I passi elencati per l’algoritmo di Shor possono sembrare molto astratti, quindi prima di spiegare come funzionano, cerchiamo di provare un esempio, utilizzando ciò che abbiamo appena appreso

sull'aritmetica modulare. Supponiamo che il numero che vogliamo fattorizzare sia 21; le vere chiavi pubbliche RSA sono molto più grandi, ma limitiamoci a 21 per semplificare i passaggi matematici. Fidatevi: così la matematica rimane molto, molto più semplice.

Ecco i passaggi dall'Algoritmo 12.1 per fattorizzare il numero 21.

1. Scegli come generatore un numero intero casuale; immaginiamo di usare 11.
2. Possiamo verificare che, poiché 11 non condivide fattori con 21, possiamo usarlo come generatore per il passo successivo.
3. Sfortunatamente non possiamo eseguire il passo quantistico "a mente", quindi usiamo l'operazione di stima della fase iterativa Q# per stimare la fase generata applicando un oracolo che implementa la funzione classica $f(x) = 11^x \bmod 21$. Restituisce una fase φ che possiamo convertire nella frequenza 427 grazie alla seguente equazione: (φ * 29)) / 2π.
4. Applichiamo a 427 l'algoritmo delle frazioni continue per ottenere un'ipotesi per quale potrebbe essere il periodo. Facendolo a mano, otteniamo una stima di 6, per il periodo.
5. Il periodo che abbiamo rilevato è pari, quindi possiamo procedere con il passo successivo.
6. Utilizzando il periodo 6 si ottiene che 116/2 - 1 mod 21 = 7 oppure 116/2 + 1 mod 21 = 9 condividono un fattore con 21. Controlliamo e verifichiamo entrambe le possibilità e confermiamo che 7 è effettivamente un fattore di 21.

ESERCIZIO 12.2 RICERCA DI FATTORI COMUNI

Provate il Passo 6 della procedura precedente, ma utilizzando 35 come numero da fattorizzare, 17 come generatore e 12 come periodo. Verificate che una o entrambe le risposte che ottenete dal Passo 6 condividano un fattore comune con 35.

Usando Python o Q#, provate la stessa cosa con $N = 143$, $g = 19$ e il periodo $r = 60$.

Nota: nel prossimo paragrafo, vedremo come utilizzare facilmente un computer classico per fattorizzare un numero dato un altro numero che condivide alcuni dei suoi fattori.

Sebbene questo sembri molto lavoro per fattorizzare un numero piccolo, come 21, 35 o 143, lo stesso identico processo funziona anche per numeri interi molto più grandi, come quelli che potremmo incontrare provando a risolvere l'enigma impiegato dell'algoritmo RSA per proteggere i dati.

Il resto del capitolo esamina in dettaglio ciascuno di questi passi e mostra come sono in grado di fattorizzare i numeri interi. Per avviare la procedura, vediamo come la matematica classica su cui si basa la ricerca del periodo ci aiuta a fattorizzare i numeri interi e come possiamo usare il linguaggio Q# per implementare tali passaggi matematici classici.

Algebra classica e fattorizzazione

Tenendo presente l'esempio concreto dell'uso dell'algoritmo di Shor, possiamo vedere come l'aritmetica e l'algebra classiche aiutino a trarre vantaggio dal quantum computing. Prima di addentrarci nella parte quantistica dell'algoritmo, è utile esplorare la parte classica, per capire perché il fatto di trovare il periodo del generatore aiuti a fattorizzare gli interi.

Possiamo ricordare dall'algebra che per ogni numero x, vale che $x2 - 1 = (x + 1) (x - 1)$. Questo funziona anche nell'aritmetica modulare. Se troviamo che il periodo r del nostro generatore g è pari:

- questo significa che esiste un intero k…
- … tale che $gr = g2^k \bmod N = 1$.

Sottraendo 1 da ogni lato, otteniamo $(g2^k - 1) \bmod N = 0$:

- quindi usando $x2 - 1 = (x + 1) (x - 1)$ abbiamo che…

- … $(g^k + 1)(g^k - 1) \bmod N = 0$.

Perché questo è importante? Se abbiamo che $x \bmod N = 0$, questo ci dice che x è un multiplo di N. Ripensando all'analogia dell'orologio, 0, 12, 24, 36 e così via sono tutti uguali a 0 mod 12. In altre parole, se $x \bmod N = 0$, allora esiste un numero intero y tale che $x = yN$. Usando questo risultato con quello che abbiamo ottenuto per il periodo, sappiamo che esiste un numero intero y tale che $(g^k + 1)(g^k - 1) = yN$. Se $g^k - 1$ o $g^k + 1$ è un multiplo di N, non abbiamo scoperto molto; ma in ogni altro caso, questo ci dice che o $g^k - 1$ o $g^k + 1$ devono condividere un fattore con N.

Per capire se è $g^k - 1$ o è $g^k + 1$ a condividere un fattore con N, possiamo calcolare il *massimo comune divisore* (MCD) di ogni ipotesi di N. Questo è semplice da fare con un computer classico, impiegando una tecnica chiamata *algoritmo di Euclide*.

> **NOTA**
>
> Poiché il MCD è così facile da calcolare in modo classico, perché abbiamo bisogno di un computer quantistico per la fattorizzazione? A questo punto nell'algoritmo di Shor, abbiamo già ristretto i fattori potenziali a due ipotesi molto buone e stiamo applicando il MCD solo a quelle ipotesi. Se non avessimo ristretto le cose così bene, avremmo dovuto applicare il MCD a molte, molte più ipotesi per avere buone possibilità di trovare i fattori di N. Anche se è facile determinare il MCD, dobbiamo comunque trovare un buon modo per restringere il campo a una buona serie di ipotesi.

In Q#, possiamo calcolare il MCD usando la funzione `GreatestCommonDivisorI`, come illustrato nel Listato 12.2, dove il codice viene eseguito in un Jupyter Notebook Q#. Un modo per verificare di aver ottenuto l'output corretto da `GreatestCommonDivisorI` consiste nell'iniziare con due numeri interi espressi come un prodotto di fattori primi: per esempio, $a = 2 \times 3 \times 113$ e $b = 2 \times 3 \times 5 \times 13$. Poiché questi due numeri interi condividono solo i fattori 2 e 3, il loro MCD dovrebbe essere $2 \times 3 = 6$.

La documentazione della libreria standard Q#

Come al solito, il Listato 12.2 inizia con le istruzioni `open` che ci consentono di usare le funzioni e le operazioni fornite dalla libreria standard Q#. In questo caso, la funzione Q# che calcola il MCD di due interi si trova nel namespace `Microsoft.Quantum.Math`, quindi iniziamo aprendo quel namespace per rendere disponibile tale funzionalità. Allo stesso modo, i fatti e le affermazioni di cui abbiamo bisogno per sottoporre a test la nostra nuova funzione `GcdExample` possono essere utilizzati aprendo il namespace `Microsoft.Quantum.Diagnostics`.

Per un elenco completo di ciò che è disponibile nella libreria standard Q#, consultate https://docs.microsoft.com/en-us/qsharp/api.

Listato 12.2 Trovare il massimo comune divisore di due numeri interi.

```
In [1]: open Microsoft.Quantum.Math;
        open Microsoft.Quantum.Diagnostics;
        function GcdExample() : Unit { (1)
            let a = 2 * 3 * 113;
            let b = 2 * 3 * 5 * 13;
            let gcd = GreatestCommonDivisorI(a, b); (2)
            Message($"The GCD of {a} and {b} is {gcd}.");
            EqualityFactI(gcd, 6, "Got the wrong GCD."); (3)
        }
Out[1]: - GcdExample
In [2]: %simulate GcdExample (4)
The GCD of 678 and 390 is 6.

Out[2]: ()
```

(1) Questa funzione è un semplice test per vedere come funziona il MCD.

(2) Per calcolare il MCD, richiamiamo `GreatestCommonDivisorI` del namespace `Microsoft.Quantum.Math`, che abbiamo aperto in precedenza.

(3) Utilizza la funzione `EqualityFactI` per confermare che la risposta che abbiamo ottenuto corrisponda a ciò che ci aspettavamo ($2 \times 3 = 6$).

(4) Come al solito, possiamo usare `%simulate` per eseguire una funzione o un'operazione su un simulatore. Qui, otteniamo l'output `()` poiché `GcdExample` restituisce un output di tipo `Unit`.

Tipi di input e convenzioni di denominazione in Q#

Avrete notato la `I` alla fine del nome `GreatestCommonDivisorI`. Questo ci dice che `GreatestCommonDivisorI` funziona su input di tipo `Int`. Quando si utilizza l'algoritmo di Shor, N sarà molto, molto più grande di qualsiasi numero che possiamo inserire in un normale valore `Int`; Q# fornisce a tale scopo il tipo `BigInt`. Per lavorare con input `BigInt`, Q# fornisce anche la funzione `GreatestCommonDivisorL`. Perché `L` e non `B`? In questo caso, `L` sta per "long", per non creare equivoci, dato che esistono altri tipi che iniziano con `B`, come `Bool`.
Questa convenzione viene utilizzata in tutte le librerie standard Q#. Per esempio, il fatto di uguaglianza che abbiamo usato in precedenza confrontava due numeri interi, e quindi si chiama `EqualityFactI`. Il fatto corrispondente per il confronto di due `BigInt` si chiama `EqualityFactL`, mentre il fatto per confrontare due valori `Result` è `EqualityFactR`.

ESERCIZIO 12.3 MASSIMO COMUNE DENOMINATORE

Qual è il MCD di 35 e 30? Questo ci aiuta a trovare i fattori di 35?
Suggerimento: pensate al Passo 2 dell'esercizio precedente.

Mettendo tutto insieme, se abbiamo il periodo del nostro generatore, il seguente listato mostra come possiamo usarlo per scrivere `MaybeFactorsFromPeriod` in Q#. Il nome della funzione inizia con quel "forse" perché è possibile che il periodo trovato non soddisfi le condizioni necessarie per apprendere qualcosa sui fattori del numero.

Listato 12.3 operations.qs: calcolo dei possibili fattori da un periodo.

```
function MaybeFactorsFromPeriod(
    generator : Int, period : Int, number : Int) (1)
: (Bool, (Int, Int)) { (2)
    if period % 2 == 0 { (3)
        let halfPower = ExpModI(generator, period / 2, number); (4)
        if (halfPower != number - 1) { (5)
            let factor = MaxI( (6)
                GreatestCommonDivisorI(halfPower - 1, number),
                GreatestCommonDivisorI(halfPower + 1, number)
            );
            return (true, (factor, number / factor));
        } else {
            return (false, (1, 1));
        }
    } else {
```

```
        return (false, (1, 1));
    }
}
```

(1) Per calcolare i possibili fattori da un periodo, dobbiamo prendere come input il numero *N* che stiamo cercando di fattorizzare, il periodo *r* e il generatore.
(2) Se $g^r/2 + 1$ oppure $g^r/2 - 1$ è un multiplo di *N*, non possiamo trovare alcun fattore e dobbiamo riprovare. L'output `Bool` consente al chiamante di sapere di dover riprovare.
(3) Se `period` è dispari, non possiamo usare il trucco x2 - 1 = (*x* + 1) (*x* - 1), quindi iniziamo controllando che `period` sia pari.
(4) La funzione Q# `Microsoft.Quantum.Math.ExpModI` restituisce esponenziali aritmetici modulari nella forma g^x mod *N*. Possiamo usarla per trovare $g^r/2$ mod *N* dati *g*, *r* e *N*.
(5) Verifica che $g^r/2 + 1$ non sia un multiplo di *N*, così sappiamo di poter continuare
(6) Il MCD (`GreatestCommonDivisorI`) ci dice se una delle nostre ipotesi ha un fattore comune con *N*. Se la nostra ipotesi non ha fattori comuni, il MCD restituisce 1. Questo controlla entrambe le ipotesi e prende quella che dà qualcosa di diverso da 1.

Ora che sappiamo come convertire un periodo nei potenziali fattori, esaminiamo il nucleo dell'algoritmo di Shor: usando la stima della fase per stimare il periodo del nostro generatore. Per farlo, utilizzeremo ciò che abbiamo appreso nel resto del libro insieme a un paio di nuove operazioni Q# per eseguire operazioni aritmetiche su un computer quantistico.

Approfondimento: parliamo di Euclide
In precedenza, abbiamo utilizzato la funzione `GreatestCommonDivisorI` delle librerie standard Q# per calcolare il MCD di due numeri interi. Questa funzione sfrutta l'algoritmo di Euclide, che tenta ricorsivamente di dividere un intero per un altro fino a quando non rimane più alcun resto.

Supponiamo di voler trovare il MCD di due interi a e b. Iniziamo l'algoritmo di Euclide trovando due interi aggiuntivi q e r (rispettivamente "quoziente" e "resto") tali che $a = qb + r$. È semplice trovare q e r utilizzando le istruzioni di divisione intera, quindi questo passaggio non è troppo difficile da eseguire su un computer classico. A quel punto, se $r = 0$, abbiamo finito: b è un divisore sia di a sia di se stesso, quindi non può esistere un divisore comune più grande. In caso contrario, sappiamo che il MCD di a e b deve essere anche un divisore di r, quindi possiamo applicare la ricorsione trovando invece il MCD di b e r. Ma questo processo deve finire, poiché gli interi di cui stiamo cercando il MCD diventano sempre più piccoli man mano che procediamo, ma non diventano mai negativi.
Per rendere le cose un po' più concrete, esaminiamo l'esempio del Listato 12.2, con la tabella seguente.

Tabella 12.1 Uso dell'algoritmo di Euclide per trovare il MCD di 678 e 390.

a	b	q	r
678	390	1	288
390	288	1	102
288	102	2	84
102	84	1	18
84	18	4	12
18	12	1	6
12	6 (risultato)	2	0 (fine)

Aritmetica quantistica

Ormai abbiamo esaminato parecchie parti delle librerie standard del linguaggio Q# e, dato l'accento di questo capitolo sull'aritmetica, ha senso introdurre alcune funzioni e operazioni del namespace `Microsoft.Quantum.Arithmetic` fornito dalla `libreria Numerics` Q#. Come potete immaginare, questo namespace fornisce molte funzioni, operazioni e tipi utili, che semplificano l'esecuzione di operazioni aritmetiche nei sistemi quantistici. In particolare, possiamo usare

implementazioni di operazioni come l'addizione e la moltiplicazione di numeri rappresentati in registri di qubit con il supporto di codifiche in registri multi-qubit come `BigEndian` (dove il bit meno significativo è a sinistra) e `LittleEndian` (dove il bit meno significativo è a destra). Vediamo per esempio del codice che utilizza la `libreria Numerics` Q# per sommare due numeri interi.

> **NOTA**
> Un notebook Q# nell'archivio degli esempi (https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp) contiene tutti questi frammenti di codice.

Innanzitutto, poiché il package `Numerics` non viene caricato di default, dobbiamo chiedere al kernel di Q# di caricarlo con il comando magico `%package`. Il comando `%package` aggiunge un nuovo package alla nostra sessione IQ#, rendendo disponibili nella sessione le funzioni, le operazioni e i tipi definiti dall'utente implementati da tale package.

Per rendere le cose più facili, possiamo anche disattivare la rappresentazione dei valori piccoli dagli output diagnostici di `DumpMachine`, come illustrato nel prossimo listato.

Listato 12.4 Caricamento di package e impostazione delle preferenze in IQ#.

```
In [1]: %package Microsoft.Quantum.Numerics (1)
Adding package Microsoft.Quantum.Numerics: done!
Out[1]: - Microsoft.Quantum.Standard::0.15.2101125897 (2)
        - Microsoft.Quantum.Standard.Visualization::0.15.2101125897
        - Microsoft.Quantum.Numerics::0.15.2101125897
In [2]: %config dump.truncateSmallAmplitudes = "true" (3)

Out[2]: "true"
```

(1) Utilizza `%package` per caricare il package `Microsoft.Quantum.Numerics`, che fornisce operazioni e funzioni aggiuntive per lavorare con i numeri rappresentati da registri di qubit.

(2) Dopo aver eseguito `%package`, IQ# elenca i package attualmente disponibili nella nostra sessione IQ#. I numeri di versione

probabilmente varieranno.

❸ Il comando magico `%config` imposta varie preferenze per la nostra attuale sessione IQ#. Qui, per esempio, possiamo usare `%config` per dire a `DumpRegister` e `DumpMachine` di tralasciare le parti di ogni vettore di stato che hanno ampiezze molto piccole. Ciò aiuta a visualizzare gli stati di diversi qubit, poiché la visualizzazione di ogni stato computazionale di base può diventare rapidamente "ingombrante".

> **SUGGERIMENTO**
>
> Quando lavoriamo con Q# nei Jupyter Notebook, il kernel IQ# fornisce vari altri comandi magici per aiutare a scrivere programmi quantistici, oltre ai comandi `%simulate`, `%package` e `%config` che abbiamo visto finora. Per un elenco completo, consultate la documentazione su https://docs.microsoft.com/qsharp/api/iqsharp-magic.

Addizioni con i qubit

Ora, passiamo alla programmazione di un esempio che somma due interi codificati in registri di qubit. Il Listato 12.5 utilizza l'operazione `AddI` per sommare il contenuto di due registri quantistici. Questo listato utilizza l'UDT `LittleEndian` fornito dalla libreria standard Q# per indicare il fatto che ogni registro di qubit è pensato per essere interpretato come un numero intero usando la codifica *little-endian* (noto anche come *least-significant order*). Cioè, quando si interpreta un registro `LittleEndian` come un numero intero, si considera l'indice di qubit più basso come il bit meno significativo. Per esempio, per rappresentare l'intero 6 come uno stato quantistico a tre qubit in notazione little-endian, scriviamo $|011\rangle$, poiché $6 = 0 \times 2^0 + 1 \times 2^1 + 1 \times 2^2 = 2 + 4$.

Listato 12.5 Uso della libreria Numerics per sommare numeri interi codificati in qubit.

```
In [3]: open Microsoft.Quantum.Arithmetic;
        open Microsoft.Quantum.Diagnostics;
        open Microsoft.Quantum.Math;
        operation AddCustom(num1 : Int, num2 : Int) : Int {
            let bitSize = BitSizeI(MaxI((num1, num2))) + 1; (1)
            use reg1 = Qubit[bitSize];
            use reg2 = Qubit[bitSize];
            let qubits1 = LittleEndian(reg1); (2)
            let qubits2 = LittleEndian(reg2); (2)
            ApplyXorInPlace(num1, qubits1); (3)
            ApplyXorInPlace(num2, qubits2);
            Message("Before addition:");
            DumpRegister((), reg2);
            AddI(qubits1, qubits2); (4)
            Message("After addition:");
            DumpRegister((), reg2);
            ResetAll(reg1); (5)
            return MeasureInteger(qubits2);
        }

Out[3]: - AddCustom
```

(1) I nostri registri devono essere abbastanza grandi per contenere la somma massima possibile dei due numeri interi. Al massimo, abbiamo bisogno di un bit in più di quanto richiesto per rappresentare l'input più grande.

(2) Indica che vogliamo interpretare `reg1` e `reg2` come numeri interi, rappresentati utilizzando la codifica little-endian.

(3) Prepara la rappresentazione `LittleEndian` di un intero nel registro di qubit, poiché $x \oplus 0 = x$, indipendentemente dal fatto che x sia 0 o 1.

(4) Utilizzando l'operazione `AddI` caricata dal package `Numerics`, possiamo sommare interi rappresentati dai due registri di input `qubits1` e `qubit2`.

(5) Esegue il reset del primo registro, in modo che possa essere deallocato, poi misura il registro con i risultati.

Possiamo vedere l'output dell'esecuzione di questo frammento nella Figura 12.4.

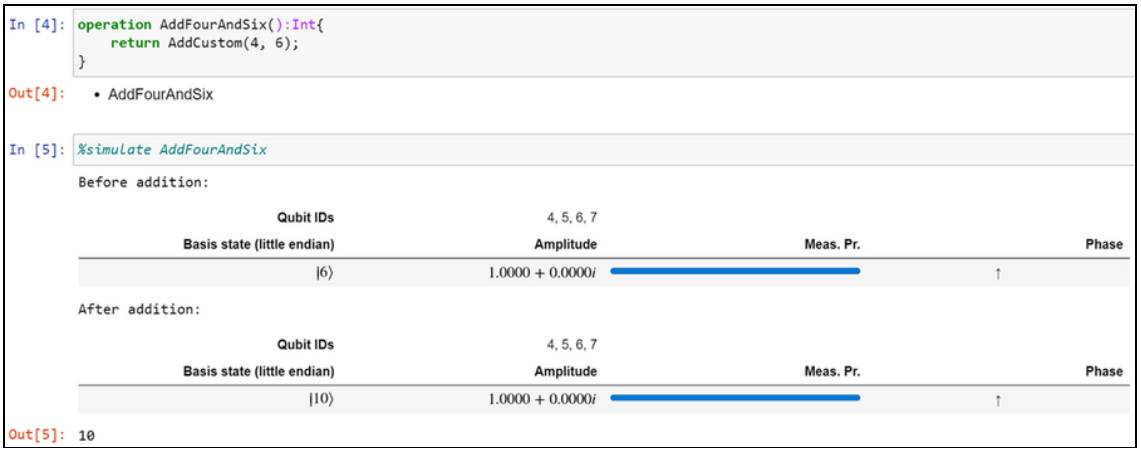


Figura 12.4 Risultato dell'utilizzo della libreria Numerics per sommare numeri interi codificati in registri di qubit.

Moltiplicazione di qubit in sovrapposizione

Abbiamo visto come eseguire un po' di aritmetica modulare di base in Q#, ma come per la maggior parte dei nostri algoritmi quantistici, a meno di usare accuratamente proprietà/operazioni quantistiche, abbiamo a che fare con calcoli molto, molto costosi. In questo paragrafo, useremo il fatto che possiamo avere qubit in *sovrapposizioni* di numeri per trarre vantaggi dall'utilizzo dell'algoritmo di Shor. Fortunatamente, `AddI` e molte altre operazioni aritmetiche simili lavorano in sovrapposizione: una proprietà di cui abbiamo bisogno per poter usare queste operazioni aritmetiche insieme alla stima della fase nel prosieguo del capitolo. Ma prima di tutto, cerchiamo di vedere che cosa significa sommare o moltiplicare numeri interi in sovrapposizione.

Qui vediamo come applicare ciò che abbiamo imparato su sovrapposizione e aritmetica utilizzando come esempio `MultiplyByModularInteger`. Utilizzeremo la stessa operazione per costruire l'oracolo di cui abbiamo bisogno per l'algoritmo di

fattorizzazione di Shor, quindi questa è un'applicazione abbastanza utile.

Per prima cosa, diamo un'occhiata a un'operazione che possiamo usare per preparare un registro in una sovrapposizione di due numeri interi. Il Listato 12.6 mostra come farlo utilizzando ciò che abbiamo appreso sull'operazione `ApplyXorInPlace` nel paragrafo precedente e sul funtore `Controlled` nel Capitolo 9.

Come abbiamo visto con altri usi di `Controlled`, le operazioni controllate fanno qualcosa quando i loro registri di controllo sono in uno stato tutti-1 ($|11\cdots1\rangle$). Il Listato 12.6 controlla invece lo stato 0, usando l'operazione `X` per mappare lo stato $|0\rangle$ sullo stato $|1\rangle$. Inserendo la chiamata a `X` in un blocco `within`/`apply`, ci assicuriamo che Q# annulli la chiamata a `X` dopo aver applicato la nostra operazione controllata.

SUGGERIMENTO

Utilizzando `within`/`apply` in questo modo si ottiene qualcosa di molto simile alla funzione `ControlledOnInt` che abbiamo usato nel Capitolo 11 ed è il modo in cui tale funzione viene implementata nelle librerie standard Q#.

Listato 12.6 Preparazione di un registro in una sovrapposizione di interi.

```
open Microsoft.Quantum.Arithmetic;
open Microsoft.Quantum.Diagnostics;
open Microsoft.Quantum.Math;
operation PrepareSuperpositionOfTwoInts( (1)
    intPair : (Int, Int),
    register : LittleEndian,
) : Unit is Adj + Ctl {
    use ctrl = Qubit();
    H(ctrl); (2)
    within {
        X(ctrl); (3)
    } apply {
        Controlled ApplyXorInPlace( (4)
            [ctrl],
            (Fst(intPair), register)
        );
    }
    Controlled ApplyXorInPlace( (5)
            [ctrl],
            (Snd(intPair), register)
```

```
        );
    (ControlledOnInt(Snd(intPair), Y))(register!, ctrl); (6)
}
```

(1) Prende un registro e una coppia di numeri interi e prepara quel registro in una sovrapposizione di quegli interi in codifica `LittleEndian`.
(2) Prepara il nostro qubit `ctrl` nello stato $|+\rangle = (|0\rangle + |1\rangle) / \sqrt{2}$, in modo che quando poi controlliamo le operazioni su tale qubit, anch'esse siano in sovrapposizione.
(3) Come notato in precedenza, l'utilizzo di `X` in un blocco `within`/`apply` ci consente di controllare lo stato $|0\rangle$ invece dello stato $|1\rangle$.
(4) Aggiunge un nuovo input per il registro di controllo (vedi Capitolo 9). L'altro input è una tupla con gli argomenti originali: l'intero che vogliamo preparare come stato e il registro sul quale vogliamo preparare tale stato.
(5) Fa la stessa cosa con il secondo intero di `intPair`, e lo codifica nel registro come controllato dal qubit `ctrl`.
(6) Aggiunge qualche fase ad uno dei due rami della nostra sovrapposizione utilizzando l'operazione `Y` di rotazione, controllata dal fatto che il nostro qubit `ctrl` sia nello stato $|1\rangle$.

> **SUGGERIMENTO**
>
> L'operazione `Y` non è necessaria in questo caso, ma ci aiuta a vedere come la fase applicata dall'operazione `Y` controllata si propaghi nei passi successivi.

Una volta che abbiamo un registro quantistico che rappresenta la sovrapposizione di due numeri interi, possiamo applicare altre operazioni aritmetiche su tale sovrapposizione. Nel listato seguente, per esempio, utilizziamo `DumpMachine` per vedere come cambia lo stato del nostro registro quando si utilizza l'operazione `MultiplyByModularInteger` fornita dalle librerie standard Q#.

Listato 12.7 Uso della libreria Numerics per moltiplicare in sovrapposizione.

```
operation MultiplyInSuperpositionMod(
    superpositionInts : (Int, Int),
    multiplier : Int,
    modulus : Int
) : Int {
    use target = Qubit[BitSizeI(modulus - 1)]; (1)
    let register = LittleEndian(target);
    PrepareSuperpositionOfTwoInts(superpositionInts, register); (2)
    Message("Before multiplication:");
    DumpMachine();
    MultiplyByModularInteger( (3)
        multiplier, modulus, register
    );
    Message("After multiplication:");
    DumpMachine();
    return MeasureInteger(register); (4)

}
```

(1) La prima cosa che dobbiamo fare è allocare un registro abbastanza grande. Qui, poiché stiamo eseguendo la moltiplicazione modulare, il valore più grande possibile che il nostro registro deve contenere è modulo -1.

(2) Utilizza l'operazione che abbiamo definito nel Listato 12.6 per preparare il nostro registro di destinazione in una sovrapposizione di numeri interi.

(3) Il package `Numerics` fornisce `MultiplyByModularInteger`, che prende un registro `LittleEndian` e lo moltiplica per il valore di un moltiplicatore classico, applicando poi un determinato modulo.

(4) Misura un registro e restituisce l'`Int` classico rappresentato da quel registro.

Se eseguiamo il codice dei Listati 12.6 e 12.7 nel notebook di esempio, otteniamo l'output rappresentato nella Figura 12.5: i registri mostrano correttamente una sovrapposizione di 2 e 3 prima della moltiplicazione e poi mostrano una sovrapposizione di 1 e 6. Quale dovrebbe essere l'output corretto? Se moltiplichiamo come al solito, dovrebbe essere 6 e 9; ma poiché stiamo operando in aritmetica modulo 8, il 9 vale 1. Quando poi misuriamo quel registro, otteniamo 6

metà delle volte e 1 l'altra metà delle volte, poiché sono in una sovrapposizione uguale, come possiamo vedere dalle barre di ampiezza dello stato rappresentate nella Figura 12.5.

ESERCIZIO 12.4 MOLTIPLICAZIONE MODULARE

Supponiamo di aver preparato un registro nello stato 1 / √2 (|2⟩ + |7⟩), dove ogni ket rappresenta un intero in codifica *little-endian*. In quale stato sarebbe il vostro registro dopo aver moltiplicato per 5 modulo 9? Scrivete un programma Q# che utilizzi `DumpMachine` per confermare la vostra risposta.

Figura 12.5 Output della moltiplicazione di 3 per 2 e 3 in sovrapposizione mod 8.

ESERCIZIO 12.5 BONUS

Se eseguite lo stesso programma del precedente esercizio, ma tentate di moltiplicare per 3 modulo 9, ottenete un errore. Perché?

Suggerimento: considerate ciò che avete imparato nel Capitolo 8 sulle funzioni classiche rappresentabili da un oracolo quantistico. Se siete in difficoltà, trovate la risposta di seguito.

Risposta: la moltiplicazione per 3 mod 9 non è reversibile. Per esempio, sia 1 volta 3, sia 4 volte 3, mod 9 dà 0, anche se 1 e 4 non sono uguali mod 9. Poiché le funzioni classiche devono essere reversibili per poter essere rappresentate da operazioni quantistiche, `MultiplyByModularInteger` genera un errore, in questo caso.

NOTA

Notate che cosa abbiamo fatto qui: abbiamo utilizzato un programma quantistico per moltiplicare un numero intero rappresentato da un registro quantistico per un numero intero classico. Il calcolo avviene interamente sul dispositivo quantistico e non utilizza alcuna misurazione; ciò significa che quando il nostro registro parte in sovrapposizione, *anche la moltiplicazione avviene in sovrapposizione*.

Moltiplicazione modulare nell'algoritmo di Shor

Ora che abbiamo parlato della libreria `Numerics`, torniamo allo scenario della fattorizzazione. I punti principali dell'Algoritmo 12.1 in cui dobbiamo fare un po' di aritmetica modulare e utilizzare la libreria `Numerics` sono i Passi 3 e 4 (che riportiamo di seguito). Possiamo implementare tre operazioni per velocizzare l'implementazione di tali passi dell'algoritmo di Shor e aiutarci a fattorizzare gli interi.

La prima operazione che possiamo considerare dal codice di esempio di questo capitolo implementa il Passo 3 dell'Algoritmo 12.1, l'operazione `EstimateFrequency` (Listato 12.8).

ALGORITMO 12.1 PASSI 3 E 4 (PSEUDOCODICE PER FATTORIZZARE UN INTERO CON L'ALGORITMO DI SHOR)

3. Utilizza la stima della fase iterativa per trovare la *frequenza* della funzione classica $f(x) = g^x \bmod N$. La frequenza ci dice quanto spesso f restituisce lo stesso valore all'aumentare di x.
4. Utilizza un algoritmo classico noto come *espansione continua delle frazioni* per convertire la frequenza del Passo 3 in un periodo (r). Il periodo r dovrebbe avere la proprietà che $f(x) = f(x + r)$ per tutti gli input x.

SUGGERIMENTO

Questa operazione utilizza le operazioni di stima della fase fornite dalle librerie standard Q# e che abbiamo visto nel Capitolo 10. Se avete bisogno di un ripasso, tornate al Capitolo 9, dove troverete una panoramica della stima della fase, o al Capitolo 10, che mostra come eseguire la stima della fase con le librerie standard.

Listato 12.8 operations.qs: apprendimento della frequenza di un generatore con stima della fase.

```
operation EstimateFrequency(
    inputOracle : ((Int, Qubit[]) => Unit is Adj+Ctl),
    nBitsPrecision : Int,
    bitSize : Int)
: Int {
    use register = Qubit[bitSize]; (1)
    let registerLE = LittleEndian(register); (2)
    ApplyXorInPlace(1, registerLE); (3)
    let phase = RobustPhaseEstimation( (4)
        nBitsPrecision,
        DiscreteOracle(inputOracle), (5)
        registerLE!
    );
    ResetAll(register); (6)
    return Round( (7)
        (phase * IntAsDouble(2 ^ nBitsPrecision)) / (2.0 * PI())
    );

}
```

(1) Poiché questo è il passaggio principale che utilizza qubit, dobbiamo allocare un registro sufficientemente grande da rappresentare il modulo.

(2) Il registro appena allocato deve specificare come codificare gli interi che dovrà rappresentare, in modo che possiamo usare l'UDT `LittleEndian` per ospitare il registro appena allocato.

(3) Prende un `Int` e applica lo XOR con il numero intero memorizzato nel registro fornito nel secondo argomento. Poiché `registerLE` inizia da 0, prepara il registro con 1.

(4) Utilizza `RobustPhaseEstimation` (vedi Capitolo 10) per conoscere la fase di `inputOracle` e passa un registro quantistico e il numero di bit di precisione per la stima della fase.

(5) Inserendo *inputOracle* nell'UDT `DiscreteOracle` chiariamo a `RobustPhaseEstimation` che vogliamo che `inputOracle` sia interpretato come un oracolo.

(6) Una volta terminata la stima della fase, azzera tutti i qubit del registro.
(7) La fase che abbiamo stimato è proprio questo: una fase. Questa equazione la converte in una frequenza: `(phase * 2nBitsPrecision - 1) /` π.

Ora che abbiamo la struttura per `EstimateFrequency`, possiamo esaminare un'operazione che implementa l'oracolo di cui abbiamo bisogno per questo algoritmo. L'operazione `ApplyPeriodFindingOracle` fa proprio questo: è strutturata come un oracolo per la funzione *f*(`power`) = `generatorpower` mod `modulus`. Il prossimo listato mostra un'implementazione di `ApplyPeriodFindingOracle`.

Listato 12.9 operations.qs: implementazione di un oracolo per la funzione f.

```
operation ApplyPeriodFindingOracle(
    generator : Int, modulus : Int, power : Int, target : Qubit[])
: Unit is Adj + Ctl {
    Fact( (1)
        IsCoprimeI(generator, modulus),
        "The generator and modulus must be co-prime."
    );
    MultiplyByModularInteger( (2)
        ExpModI(generator, power, modulus), (3)
        modulus,
        LittleEndian(target) (4)
    );

}
```

(1) Fa qualche controllo sull'input, che `generator` e `modulus` siano coprìmi.
(2) Lo stesso del Listato 12.7. Qui, aiuta questo oracolo a moltiplicare il numero intero rappresentato nel registro target per *f*(`power`) = `generatorpower` mod `modulus`.
(3) `Microsoft.Quantum.Math` offre anche la funzione `ExpModI`, che ci consente di calcolare facilmente *f*(`power`) = `generatorpower` mod `modulus`.

(4) `LittleEndian` ci dice che il registro di qubit preso da `ApplyPeriodFindingOracle` deve essere interpretato come un numero intero in codifica *little endian*.

Le due operazioni precedenti costituiscono la base per il Passo 3 dell'Algoritmo 12.1. Ora abbiamo bisogno di un'operazione che si occupi del Passo 4, dove convertiamo in un periodo la frequenza stimata del generatore. L'operazione `EstimatePeriod` nel listato seguente fa proprio questo: dato un generatore e un modulo, ripete la stima della frequenza utilizzando `EstimateFrequency` e utilizza l'algoritmo delle frazioni continue per garantire che la frequenza stimata produca un periodo valido.

Listato 12.10 operations.qs: stima dei periodi dalle frequenze.

```
operation EstimatePeriod(generator : Int, modulus : Int) : Int {
    Fact( (1)
        IsCoprimeI(generator, modulus), (2)
        "`generator` and `modulus` must be co-prime"
    );
    let bitSize = BitSizeI(modulus); (3)
    let nBitsPrecision = 2 * bitSize + 1; (4)
    mutable result = 1; (5)
    mutable frequencyEstimate = 0
    repeat { (6)
        set frequencyEstimate =
            EstimateFrequency( (7)
                ApplyPeriodFindingOracle( (8)
                    generator, modulus, _, _
                ),
                nBitsPrecision, bitSize
            );
        if frequencyEstimate != 0 { (9)
            set result =
                PeriodFromFrequency( (10)
                    frequencyEstimate, nBitsPrecision,
                    modulus, result
                );
        } else {
            Message("The estimated frequency was 0, trying again.");
        }
    }
    until ExpModI(generator, result, modulus) == 1 (11)
    fixup {
        Message( (12)
            "The estimated period from continued fractions failed, " +
```

```
            "trying again."
        );
    }
    return result;
}
```

(1) Fa qualche controllo sull'input: che il generatore e il modulo forniti siano coprìmi.

(2) La funzione `IsCoprimeI` del namespace `Microsoft.Quantum.Math` semplifica il controllo se il generatore e il modulo sono coprìmi.

(3) L'intero più grande che un registro di qubit dovrebbe contenere è il modulo, quindi usiamo `BitSizeI` per calcolare il numero di bit in modo tale che `modulus` $\leq 2\{\#\text{ di bit}\}$.

(4) Per usare un numero in virgola mobile per rappresentare k/r dove r è il periodo e k è un altro numero intero, abbiamo bisogno di un numero sufficiente di bit di precisione per approssimare k/r: cioè un bit in più di quanto richiesto per rappresentare k e r.

(5) La variabile `mutable result` tiene traccia della nostra migliore ipotesi corrente per il periodo, mentre ripetiamo il blocco `repeat`.

(6) Ripete i passi di stima della frequenza tante volte quante sono necessarie per garantire che abbiamo una stima appropriata del periodo, con cui procedere.

(7) Richiama l'operazione `EstimateFrequency` che abbiamo esaminato in precedenza e le passa gli argomenti appropriati.

(8) Applica parzialmente `ApplyPeriodFindingOracle` per garantire che `EstimateFrequency` possa applicarla ai valori corretti di potenza e dei registri.

(9) Se `frequencyEstimate` è `0`, dobbiamo riprovare, poiché non ha senso avere un periodo (1/0). Se otteniamo 0, il blocco `repeat` viene eseguito di nuovo, poiché la condizione `until` non è soddisfatta.

(10)> Cattura il Passo 4 dell'Algoritmo 12.1, che utilizza l'algoritmo delle frazioni continue nella libreria standard Q# per calcolare il

periodo a partire dalla frequenza.
(11) Ripete la stima della frequenza e il calcolo del periodo finché non avremo un periodo tale che `ExpModI(generator, result, modulus) == 1`.
(12) Se la condizione `until` non viene soddisfatta, viene eseguito il blocco `fixup`, che produce semplicemente un messaggio che sta per riprovare.

Con quest'ultima operazione, abbiamo tutto il codice di cui abbiamo bisogno per implementare completamente l'algoritmo di Shor! Nel prossimo paragrafo, metteremo tutto insieme ed esploreremo le implicazioni di questo algoritmo nella fattorizzazione di numeri interi.

Soluzione completa

Ora abbiamo appreso e messo in pratica tutte le conoscenze necessarie per programmare ed eseguire l'algoritmo di Shor. La parte quantistica dell'algoritmo di Shor era abbastanza familiare, grazie a ciò che abbiamo appreso nei Capitoli 9 e 10 sulla stima della fase. Abbiamo esaminato l'algebra classica che collega fra loro i compiti di fattorizzazione dei numeri e di trovare il periodo di un generatore.

Frazioni continue: convergenze

Potreste aver notato un altro principio matematico classico all'opera nell'algoritmo di Shor, ma non ne abbiamo ancora parlato. In particolare, la funzione `PeriodFromFrequency` viene richiamata sull'output che otteniamo dalla stima della fase:

```
function PeriodFromFrequency(
    frequencyEstimate : Int, nBitsPrecision : Int,
    modulus : Int, result : Int)
: Int {
    let continuedFraction = ContinuedFractionConvergentI(
        Fraction(frequencyEstimate, 2^nBitsPrecision), modulus
    );
    let denominator = AbsI(Snd(continuedFraction!));
    return (denominator * result) / GreatestCommonDivisorI(
        result, denominator
    );

}
```

Questo è necessario perché la stima della fase non ci dice esattamente di che cosa abbiamo bisogno. Invece di dirci quanto tempo impiega la nostra funzione a "girare intorno a un orologio" (il periodo della nostra funzione), ci dice quanto velocemente la nostra funzione ruota su tale orologio: qualcosa di più simile a una frequenza. Sfortunatamente, non possiamo prendere il reciproco della frequenza per ottenere il periodo, poiché il periodo che stiamo cercando è un numero intero. Quindi, se otteniamo una stima della frequenza di f, dobbiamo cercare vicino a $f/2n$ la frazione più vicina della forma N/r per trovare il periodo r.
Questo è un problema aritmetico del tutto classico e fortunatamente è stato risolto usando una tecnica nota come convergenti di frazioni continue. Questa soluzione è disponibile nelle librerie standard Q# grazie alla funzione `ContinuedFractionConvergentI`, che semplifica il passaggio dalla stima della fase al periodo della nostra funzione.

Esaminiamo un attimo l'operazione `FactorSemiprimeInteger` che abbiamo visto all'inizio del capitolo, alla luce di ciò che abbiamo appreso. Per un ripasso, consultate la Figura 12.6.

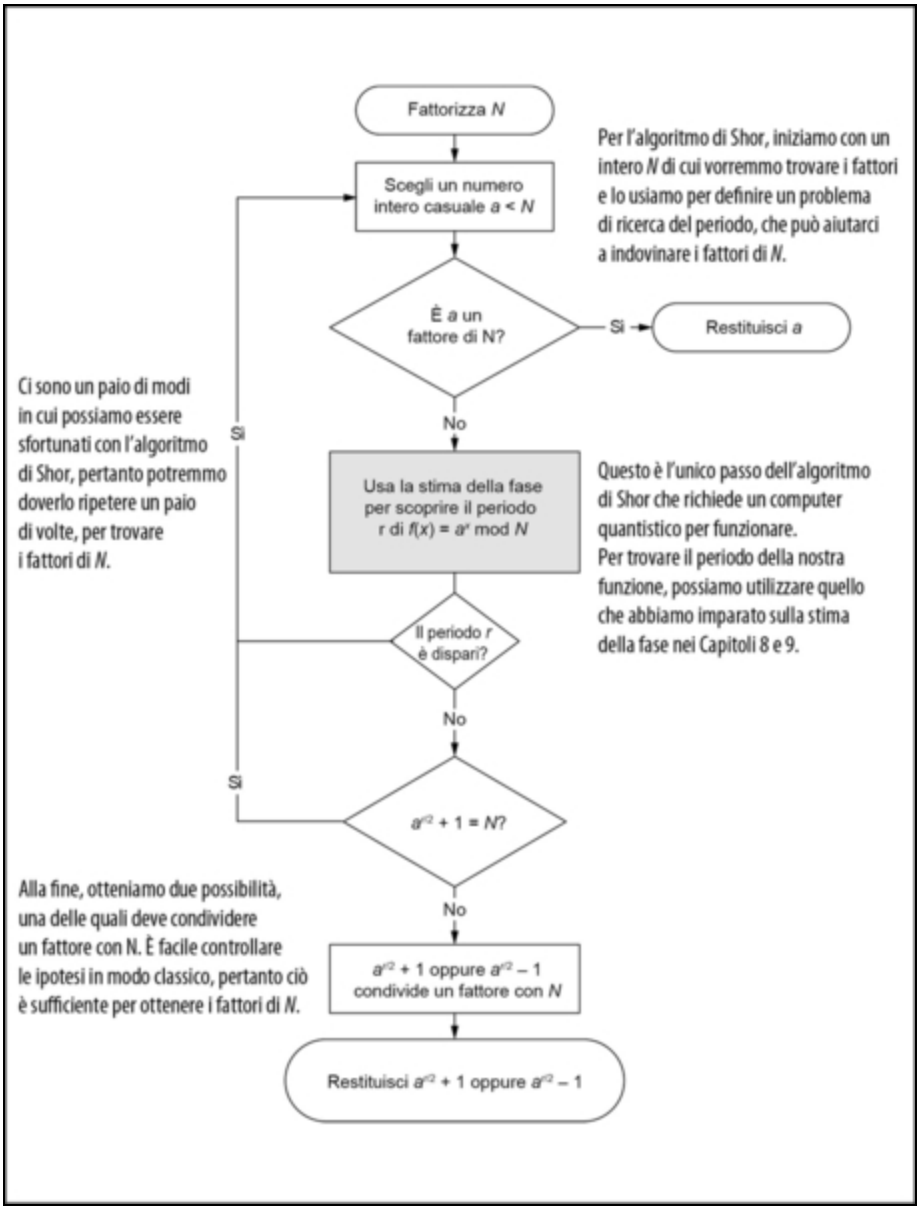


Figura 12.6 Algoritmo di Shor rappresentato come un diagramma di flusso. Per fattorizzare un intero *N*, l'algoritmo di Shor utilizza la stima della fase e un computer quantistico per trovare il periodo di una funzione che prende le potenze di un altro intero a utilizzando l'aritmetica modulare modulo *N*. Dopo un po' di post-elaborazione classica, tale periodo può essere usato per trovare i fattori di *N*.

Listato 12.11 operations.qs: fattorizzazione di interi semiprimi con l'algoritmo di Shor.

```
operation FactorSemiprimeInteger(number : Int) : (Int, Int) {
    if number % 2 == 0 { (1)
        Message("An even number has been given; 2 is a factor.");
        return (number / 2, 2);
    }
    mutable factors = (1, 1); (2)
    mutable foundFactors = false; (3)
    repeat {
        let generator = DrawRandomInt(3, number - 2); (4)
        if IsCoprimeI(generator, number) { (5)
            Message($"Estimating period of {generator}...");
            let period = EstimatePeriod(generator, number); (6)
            set (foundFactors, factors) =
                MaybeFactorsFromPeriod(
                    generator, period, number (7)
                );
        } else {
            let gcd = GreatestCommonDivisorI(number, generator); (8)
            Message(
                $"We have guessed a divisor of {number} to be " +
                $"{gcd} by accident. Nothing left to do."
            );
            set foundFactors = true;
            set factors = (gcd, number / gcd);
        }
    }
    until foundFactors (9)
    fixup { (10)
        Message(
            "The estimated period did not yield a valid factor, " +
            "trying again."
        );
    }
    return factors; (11)

}
```

(1) Controlla se il numero intero da fattorizzare è pari. Se è così, allora 2 è un fattore e possiamo fermarci presto.

(2) Utilizza la variabile `mutable factors` per registrare i fattori che troviamo per `number` mentre eseguiamo l'algoritmo.

(3) Utilizza il flag `mutable foundFactors` per registrare se troviamo dei fattori per `number` mentre eseguiamo l'algoritmo.

(4) Implementa il Passo 1 dell'Algoritmo 12.1, utilizzando `DrawRandomInt` per selezionare un numero intero casuale nell'intervallo fra `1` e `number - 1` come nostro generatore.

(5) Il Passo 2 dell'Algoritmo 12.1, dove verifichiamo che `generator` è coprìmo nei confronti dell'intero che vogliamo fattorizzare; se non lo è, la clausola `else` si occupa di restituire il fattore comune tra i due.

(6) Copre i Passi 3 e 4 dell'Algoritmo 12.1. Restituisce un periodo che calcola con le frazioni continue dalla stima della frequenza all'interno di `EstimatePeriod`.

(7) Utilizza l'algebra per trasformare il periodo stimato in numeri interi che potrebbero essere fattori. A volte non riesce, quindi restituisce un valore `Bool` che indica se l'operazione è riuscita.

(8) Gestisce il caso in cui il generatore che abbiamo indovinato all'inizio abbia un fattore comune con il numero che stiamo cercando di fattorizzare.

(9) Aggiunge la condizione per quanto tempo dovremmo ripetere il blocco precedente. Qui vogliamo continuare a cercare fino a trovare i fattori che stiamo cercando.

(10)> Dice al programma cosa fare prima di ripetere il ciclo principale. Qui vogliamo solo essere informati del fatto che il programma Q# proverà di nuovo.

(11) Restituisce la tupla di fattori per il numero di input.

Ecco il risultato eseguendo questa operazione in un notebook IQ# fino per fattorizzare 21.

Listato 12.12 Output dell'esecuzione di FactorSemiprimeInteger.

```
In [1]: open IntegerFactorization; (1)
        operation Factor21() : (Int, Int) {
            return FactorSemiprimeInteger(21);
        }
Out[1]: - Factor21
In [2]: %simulate Factor21
        We have guessed a divisor of 21 to be 3 by accident. Nothing
left to do.
Out[2]: (3, 7)
In [3]: %simulate Factor21
        Estimating period of 19...
        The estimated period from continued fractions failed, trying
again.
```

```
Out[3]: (7, 3)
In [4]: %simulate Factor21
        Estimating period of 17...
        The estimated period did not yield a valid factor, trying again.
        We have guessed a divisor of 21 to be 3 by accident. Nothing
left to do.

Out[4]: (3, 7)
```

(1) Per richiamare l'operazione `FactorSemiprimeInteger` da un notebook, è utile scrivere una nuova operazione che fornisca l'input 21.

Ogni volta, il nostro codice ha restituito correttamente i fattori primi di 21: 3 e 7. L'operazione è stata ripetuta tre volte per mostrare alcuni possibili risultati che potremmo ottenere. In `In [2]`, quando abbiamo provato a indovinare un generatore, la chiamata a `DrawRandomInt` ha finito per restituire un numero intero che non era coprìmo con 21. Pertanto, siamo stati in grado di utilizzare `GreatestCommonDivisorI` per trovare una fattorizzazione. In `In [3]`, la chiamata a `DrawRandomInt` ha selezionato il generatore 19 e quindi ha dovuto eseguire la stima della frequenza due volte per garantire che l'algoritmo delle frazioni continue avesse successo. Nella sessione finale, `In [4]`, è stato completato un round completo dell'attività di ricerca del periodo che però non è riuscito a fornire un fattore corretto; e quando ha provato a scegliere un nuovo generatore, ha indovinato un fattore per puro caso.

> **NOTA**
>
> Date le limitazioni dell'esecuzione su simulatori o piccoli dispositivi hardware, spesso ci capiterà di indovinare i fattori corretti quando selezioneremo il generatore. Inoltre saremo sfortunati più spesso con i numeri interi piccoli, indovinando fattori banali come 1. Questi casi limite si verificano meno spesso man mano che il numero che stiamo cercando di fattorizzare aumenta di dimensioni.

Utilizzando un simulatore su un laptop, un desktop o nel cloud, probabilmente non saremo in grado di fattorizzare nulla di molto grande con l'algoritmo di Shor. Per esempio, sarebbe piuttosto difficile fattorizzare un numero di 30 bit simulando l'algoritmo di Shor su un

computer classico, ma i numeri di 40 bit erano considerati gravemente insufficienti e non in grado di resistere agli algoritmi di fattorizzazione classici già nel 1992. Potrebbe sembrare che l'algoritmo di Shor sia inutile, ma quello che ci dice è che è difficile usare un computer classico per simulare grossi programmi quantistici, e abbiamo visto il perché nei Capitoli 4 e 5.

Infatti, poiché lo stesso algoritmo funziona per numeri molto più grandi (non è esagerato usare chiavi a 4.096 bit per proteggere i dati personali), come quelli comunemente usati per proteggere i dati online, capire come funzionano l'algoritmo di Shor e altri algoritmi quantistici simili può aiutarci ad apprezzare i presupposti degli usi correnti della crittografia e per capire di che cosa abbiamo bisogno per il futuro.

Il passo successivo per garantire la privacy

Dato quello che abbiamo imparato sull'algoritmo di Shor, potrebbe sembrare che la crittografia che oggi protegge di tutto, dalle cartelle cliniche alla cronologia delle chat, sia ormai condannata. Fortunatamente, esistono sia tecnologie quantistiche (come la distribuzione quantistica delle chiavi, di cui abbiamo parlato nel Capitolo 3) sia nuove tecnologie classiche destinate a resistere anche ad algoritmi come quello di Shor. Quest'ultima classe di tecnologie, la *crittografia post-quantistica*, è oggetto di molte ricerche ed esplorazioni.

Q# può svolgere un ruolo importante nella ricerca sulla crittografia, aiutando a capire quanto dovrebbe essere grande un computer quantistico per essere in grado di attaccare un determinato sistema crittografico. Per esempio, i ricercatori di Google hanno recentemente usato Q# e Python per migliorare il costo richiesto per implementare la fase di moltiplicazione modulare dell'algoritmo di Shor, (Craig Gidney, *Asymptotically Efficient Quantum Karatsuba Multiplication* (2018), https://arxiv.org/abs/1904.07356) cosa che li ha aiutati a stimare che sarebbero necessari 20 milioni di qubit per attaccare le istanze RSA utilizzando gli attuali algoritmi quantistici (Craig Gidney e Martin Ekerå, How to Factor 2048-bit RSA Integers in 8 Hours Using 20 Million Noisy Qubits (2019), https://arxiv.org/abs/1905.09749). Allo stesso modo, https://github.com/Microsoft/grover-blocks è un ottimo esempio di utilizzo di Q#

per capire il modo in cui l'algoritmo di Grover (vedi Capitolo 11) può operare sugli algoritmi a chiave simmetrica come AES.
In entrambi i casi, Q# è uno strumento prezioso per capire quanta potenza di calcolo quantistica sarebbe necessaria per violare gli attuali sistemi crittografici. Insieme alle ipotesi sulla velocità con cui gli algoritmi quantistici e l'hardware continueranno a progredire, su quanta potenza di calcolo quantistica sarà possibile acquistare per un hacker e sui requisiti di resistenza temporale di algoritmi come RSA per garantire la nostra privacy, le conoscenze sviluppate utilizzando Q# possono aiutarci a scoprire la velocità con cui gli attuali sistemi crittografici dovranno essere sostituiti. Come qualsiasi aspetto nella sicurezza delle informazioni, la garanzia della privacy contro gli aggressori quantistici è un argomento molto complesso, senza risposte facili. Fortunatamente, strumenti come Q# e il Quantum Development Kit aiutano a rendere il problema un po' più trattabile.

Riepilogo

- La crittografia moderna funziona nascondendo dei segreti tramite enigmi matematici difficili da risolvere per i computer classici, come la fattorizzazione dei numeri. I grandi computer quantistici possono però essere in grado di fattorizzare i numeri, cosa che cambia il modo in cui consideriamo la crittografia.
- L'aritmetica modulare generalizza il moto delle lancette dell'orologio: per esempio, 25 + 5 farebbe 3 su un quadrante di 27 ore.
- Gli interi con esattamente due fattori primi sono detti *semiprimi* e possono essere fattorizzati utilizzando computer quantistici, cosa che permette di risolvere problemi aritmetici modulari grazie alla stima della fase.
- La libreria Q# `Numerics` fornisce funzioni e operazioni utili per lavorare con numeri interi modulari su computer quantistici.

- L’algoritmo di Shor combina le fasi classiche di pre e post-elaborazione con la stima della fase su un computer quantistico per fattorizzare rapidamente gli interi utilizzando l’aritmetica modulare.

Conclusioni

Prima di salutarci, è utile fare un passo indietro e apprezzare come tutte le varie conoscenze che abbiamo appreso in tutto il libro si siano riunite in questo capitolo per aiutarci a comprendere un’applicazione davvero reale per i computer quantistici. Nella Parte I, abbiamo appreso le basi su come descrivere e simulare i computer quantistici e sugli effetti quantistici che rendono unico il quantum computing. Nel Capitolo 3 abbiamo imparato a utilizzare i singoli qubit e le sovrapposizioni per condividere le chiavi crittografiche in modo sicuro con la distribuzione quantistica delle chiavi. Nei Capitoli da 4 a 6, abbiamo considerato l’entanglement di più qubit per giocare e per trasferire i dati su un dispositivo quantistico. Abbiamo persino costruito il nostro simulatore quantistico in Python, per implementare questi giochi e conoscere la matematica necessaria per descrivere gli effetti quantistici.

Con tutte queste basi, nella Parte II, abbiamo iniziato a scrivere algoritmi quantistici per aiutare i personaggi di Camelot a svolgere ad alcuni giochi. Nel Capitolo 7 abbiamo parlato di Q#, un nuovo linguaggio di programmazione progettato specificamente per scrivere facilmente programmi per computer quantistici. Nel Capitolo 8, abbiamo implementato l’algoritmo di Deutsch-Jozsa per la scelta del nuovo re, ma lungo la strada abbiamo anche imparato a conoscere gli oracoli e come possono aiutarci a valutare le funzioni classiche in un programma quantistico. Abbiamo anche sviluppato il nostro

programma di stima della fase, nel Capitolo 9, dove abbiamo imparato a manipolare la fase e usarla con il *phase kickback* per esaminare le attività dei nostri programmi quantistici.

Avendo a nostra disposizione tutte queste tecniche di programmazione quantistica, abbiamo affrontato alcune delle applicazioni più entusiasmanti del quantum computing. Nel Capitolo 10, abbiamo imparato la simulazione hamiltoniana e come possiamo usare i sistemi quantistici per simulare i livelli di energia di varie sostanze chimiche. Nel Capitolo 11, abbiamo implementato l'algoritmo di Grover, per la ricerca di informazioni in dati non strutturati, grazie all'amplificazione dell'ampiezza. In questo capitolo, abbiamo utilizzato tutto, dalle funzioni e operazioni diagnostiche Q# alla stima della fase e tutto ciò che riguarda i blocchi `within`/`apply` alle rappresentazioni di oracoli delle funzioni classiche alla fattorizzazione dei numeri su un computer quantistico. Usando ciò che abbiamo appreso nel resto del libro, la maggior parte delle parti davvero difficili della scrittura dell'algoritmo di Shor si sono rivelate quelle parti *classiche*, necessarie per collegare il problema della scomposizione dei numeri a quello della ricerca del periodo.

Anche se questo libro non ha esaurito tutto ciò che c'è da imparare sul quantum computing – dopotutto sono successe molte cose dal 1985 – ciò che avete imparato vi ha dato tutto ciò di cui avete bisogno per continuare a imparare, esplorare e sviluppare le vostre conoscenze nel campo del quantum computing. Usando Python e Q#, avete gli strumenti per prendere parte a uno dei progressi più entusiasmanti dell'informatica, per aiutare i vostri colleghi a imparare insieme a voi e per costruire una comunità in grado di mettere a frutto il quantum computing. Buon divertimento, quindi!

E ora?

Sebbene ci sia sempre qualcosa da imparare sul quantum computing, ora avete tutto ciò di cui avete bisogno per iniziare a sviluppare applicazioni quantistiche utilizzando Python e Q#. Se siete interessati a scoprire di più sul quantum computing, ecco alcune risorse utili per procedere.

- *Q# Community* (`qsharp.community`): una community open source sulla programmazione quantistica in Q#, comprendente blog, archivi di codice e meetup online.
- *Microsoft Quantum Docs* (`https://docs.microsoft.com/azure/quantum`): documentazione di riferimento su tutto ciò che riguarda il Quantum Development Kit.
- *arXiv* (`arxiv.org`): un archivio di documenti e manoscritti scientifici, con molte ricerche sul quantum computing.
- *Unitary Fund* (*unitary.fund*): società non-profit che fornisce borse di studio e sostegno finanziario per il software quantistico open source, insieme a utili suggerimenti per progetti open source cui prendere parte.
- *Quantum Open Source Foundation* (`www.qosf.org`): fondazione per lo sviluppo di software quantistico open source, comprendente un elenco di progetti e risorse in corso per migliorare l'apprendimento.
- *QC Ethics* (`qcethics.org`): risorse per l'etica nel quantum computing.
- *Q-Turn* (`q-turn.org`): una serie di conferenze sul quantum computing inclusivo.
- *Quantum Algorithm Zoo* (`quantumalgorithmzoo.org`): elenco di algoritmi quantistici noti, con link a vari documenti.
- *Quantum Computing: A Gentle Introduction* (Jack D.Hidary, Springer, 2019): tutti i dettagli sulla matematica di base degli algoritmi quantistici che abbiamo trattato in questo libro.

Molte università e college offrono anche corsi o programmi di ricerca che potrebbero interessarvi mentre continuate a esplorare il quantum computing. Comunque decidiate di continuare, speriamo che vi divertiate e che possiate riuscire a rendere sempre più ricca la comunità del quantum computing!

Installazione del software

L'inizio di quasi tutti i progetti implica la ricerca o la configurazione di un ambiente di sviluppo sul computer. Questa appendice ha lo scopo di aiutarvi a svolgere i preparativi per poter utilizzare gli esempi di questo libro (con Binder o GitHub Codespaces) e per installare un ambiente Python e Microsoft Quantum Development Kit che possiate impiegare in locale. In caso di problemi, consultate la documentazione più aggiornata per l'utilizzo del Quantum Development Kit (https://docs.microsoft.com/azure/quantum/install-overview-qdk) e/o aprite un'issue nel repository di questo libro (https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp).

Esecuzione degli esempi online

Se intendete provare gli esempi di questo libro senza installare nulla, avete due ottime opzioni:

- Binder (mybinder.org), un servizio gratuito per esplorare i notebook Jupyter;
- GitHub Codespaces, un ambiente di sviluppo basato su cloud.

Binder

Per utilizzare Binder, andate semplicemente su

https://mybinder.org/v2/gh/crazy4pi314/learn-qc-with-python-and-

`qsharp/master`. Potrebbero essere necessari alcuni istanti, ma Binder avvierà una nuova installazione di Jupyter Notebook, completa dei package Python necessari e del supporto Q#.

> **ATTENZIONE**
>
> Il servizio Binder è dedicato solo all'esplorazione e *cancellerà* le modifiche dopo circa 20 minuti di inattività. Sebbene Binder sia un ottimo modo per iniziare, se desiderate continuare a sviluppare programmi quantistici, dovrete utilizzare GitHub Codespaces o installare Python e il Quantum Development Kit localmente, sulla vostra macchina.

GitHub Cocespaces

Al momento della stesura di questo testo, GitHub Codespaces è ancora in early preview su `https://github.com/features/codespaces`. Per istruzioni su come utilizzare gli esempi di codice di questo libro con Codespaces, consultate l'archivio degli esempi su `https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp`.

Installazione con Anaconda

Nella Parte I del libro, facciamo un uso massiccio di Python come strumento per esplorare la programmazione quantistica, mentre nelle Parti II e III usiamo Python e Q#. In tal modo, facciamo affidamento sul Quantum Development Kit e su diverse librerie Python che semplificano la scrittura di programmi scientifici. Di conseguenza, durante l'installazione, può essere molto più semplice utilizzare una distribuzione software scientifica, come Anaconda (`anaconda.com`), per gestire Python e altri strumenti di programmazione scientifica.

Installazione di Anaconda

Per installare Anaconda, seguite le istruzioni su https://docs.anaconda.com/anaconda/install.

ATTENZIONE

Al momento della stesura di questo testo, la distribuzione Anaconda è fornita con Python 2.7 o 3.8. Python 2.7 ha ufficialmente raggiunto la fine del suo ciclo di vita a partire da gennaio 2020 ed è fornito solo per motivi di compatibilità. Per questo libro prevediamo l'impiego di Python 3.7 o versione successiva, quindi assicuratevi di installare una versione di Anaconda che fornisca Python 3.

Installazione di package con Anaconda

I package sono un ottimo modo per collaborare e risparmiare tempo quando stiamo cercando di imparare o di sviluppare nuovo codice. Sono un modo per raccogliere il codice correlato e "metterlo insieme" in modo che sia facile condividerlo con altri. I package possono essere installati sulla macchina con un gestore di package, e ve ne sono alcuni piuttosto comuni per Python. Potete scegliere il gestore che preferite. Cominciamo osservando i gestori di package che abbiamo già installato come parte di Anaconda.

Per default, Anaconda viene fornito con due gestori di package: pip e conda. Dato che abbiamo installato Anaconda, il gestore di package conda ha alcune funzionalità aggiuntive che lo rendono una buona scelta predefinita per la gestione dei package. Conda offre il supporto per l'installazione automatica delle dipendenze, quando installate un package; inoltre ha il concetto di ambiente, che può essere davvero utile per creare sandbox Python dedicate allo specifico progetto al quale stiamo lavorando. Una buona strategia generale è installare i package da conda quando sono disponibili, e installare gli altri package da pip.

NOTA

Il gestore di package conda può essere utilizzato con gli ambienti a riga di comando più comuni. Ma per usare conda con PowerShell, è necessaria la versione 4.6.0 o successiva. Per controllare la vostra versione, eseguite `conda --version`. Se avete bisogno di aggiornarlo, eseguite `conda update conda`.

Ambienti conda

Potreste imbattervi in situazioni in cui i package di cui avete bisogno per due diversi progetti si "rifiutano" a vicenda. Per isolare i progetti l'uno dall'altro, la distribuzione Anaconda fornisce `conda env`, uno strumento in grado di gestire più ambienti. Ogni ambiente è una copia completamente indipendente di Python, dotato dei soli package necessari per un determinato progetto o applicazione. Gli ambienti possono persino utilizzare versioni diverse di Python: un ambiente potrebbe utilizzare Python 2.7 e un altro Python 3.8. Gli ambienti sono inoltre ideali per collaborare con altri: possiamo inviare ai nostri colleghi un unico piccolo file di testo, `environment.yml`, per dire al loro `conda env` come creare un ambiente identico al nostro.

Per ulteriori informazioni, vedere https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html.

Seguendo tale strategia, cerchiamo di iniziare con un nuovo ambiente dotato dei package necessari. Gli esempi di codice per questo libro, disponibili su https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp, vengono forniti con un file `environment.yml` che dice a conda quali package avete bisogno di avere nel vostro nuovo ambiente. Clonate o scaricate il codice dall'archivio di questo libro, quindi eseguite quanto segue dalla riga di comando:

```
conda env create -f environment.yml
```

Questo comando crea un nuovo ambiente chiamato `qsharp-book` che utilizza i package del canale `conda-forge` e installa Jupyter Notebook, Python (versione 3.6 o successiva), il kernel IQ# per Jupyter, l'interprete IPython, NumPy, il motore grafico Matplotlib e QuTiP. Il gestore di package conda vi chiederà di confermare l'elenco dei package che

verranno installati. Per procedere, premete “y” e poi Invio; poi prendetevi una tazza di caffè.

> **SUGGERIMENTO**
>
> Il file `environment.yml` fornito su https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp installa anche il package `qsharp`, che fornisce l’integrazione tra Python e il Quantum Development Kit (che installerete nel passaggio successivo).

Una volta che conda avrà terminato di creare il vostro nuovo ambiente, provatelo. Per sottoporre a test il vostro nuovo ambiente, dovete prima attivarlo:

```
conda activate qsharp-book
```

Una volta che l’ambiente `qsharp-book` è stato attivato, il comando `python` dovrebbe richiamare la versione installata in tale ambiente. Per verificarlo, potete visualizzare il percorso del comando `python` per l’ambiente. Eseguite `python`, quindi eseguite quanto segue al prompt di Python:

```
>>> import sys; print(sys.executable)
C:\Users\Chris\Anaconda3\envs\qsharp-book\python.exe (1)
```

(1) Potreste vedere un percorso diverso, a seconda del vostro sistema.

Se il vostro ambiente è stato creato con successo, potete usarlo dalla riga di comando con IPython o nel vostro browser con Jupyter Notebook. Per iniziare con IPython, eseguite `ipython` dalla riga di comando (assicuratevi di aver attivato prima `qsharp-book`):

```
$ ipython
In [1]: import qutip as qt
In [2]: qt.basis(2, 0)
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket
Qobj data =
[[1.]
 [0.]]
```

Per saperne di più sull’uso di NumPy, consultate il Capitolo 2. Per saperne di più sull’uso di QuTiP, consultate il Capitolo 5. Per saperne di più sul Quantum Development Kit, continuate a leggere.

Per maggiori informazioni

- Documentazione di Anaconda: https://docs.anaconda.com/anaconda.
- Documentazione di NumPy: http://numpy.org.
- Documentazione di Jupyter Notebook: https://jupyter-notebook.readthedocs.io/en/stable/notebook.html.

Installazione del Quantum Development Kit

SUGGERIMENTO

L'ultima versione delle istruzioni di installazione per il Quantum Development Kit è disponibile su https://docs.microsoft.com/azure/quantum/install-overview-qdk. Per utilizzare gli esempi di codice di questo libro, assicuratevi di installare il supporto per Python e Jupyter Notebook quando seguite la guida all'installazione del Quantum Development Kit.

Il Quantum Development Kit Microsoft è un insieme di strumenti per lavorare e programmare in Q#, un nuovo linguaggio per la programmazione quantistica. Se avete installato Python con Anaconda, potete iniziare a utilizzare il Quantum Development Kit usando conda:

```
conda install -c quantum-engineering qsharp
```

Se preferite utilizzare il codice Q# con programmi autonomi o fuori dall'ambiente conda, potete seguire le istruzioni fornite in questo paragrafo per installare il Quantum Development Kit localmente.

NOTA

Questo libro si concentra sull'uso di Visual Studio Code, ma il Quantum Development Kit può essere utilizzato anche con qualsiasi altro editor di testi, seguendo le istruzioni della riga di comando nel testo principale. Il Quantum Development Kit può essere utilizzato anche con Visual Studio 2019 o versioni successive, utilizzando l'estensione disponibile in https://marketplace.visualstudio.com/items?itemName=quantum.DevKit.

Utilizzando l'installazione di Visual Studio Code dalla configurazione dell'ambiente Python, dovrete eseguire alcune operazioni per utilizzare il Quantum Development Kit con C#, Python e Jupyter Notebook.

1. Installare il .NET Core SDK.
2. Installare i template per progetti Q#.
3. Installare l'estensione Quantum Development Kit per Visual Studio Code.
4. Installare il supporto Q# per Jupyter Notebook.
5. Installare il package `qsharp` per Python.

Una volta fatto tutto questo, avrete tutto il necessario per scrivere ed eseguire programmi quantistici in Q#.

Che cos'è .NET?

Rispondere a questa domanda è diventato un po' più complicato da un po' di tempo a questa parte. Storicamente, *.NET* era un'abbreviazione ragionevole per *.NET Framework*, una macchina virtuale e un'infrastruttura di compilazione per qualsiasi linguaggio .NET (in particolare, C#, F# e Visual Basic .NET). .NET Framework è solo per Windows, ma esistono reimplementazioni di terze parti come Mono per altre piattaforme, fra cui macOS e Linux.

Un paio di anni fa, tuttavia, Microsoft e .NET Foundation hanno reso open source una nuova versione di .NET chiamata .NET Core. A differenza di .NET Framework, .NET Core è già pronto per l'uso multipiattaforma. .NET Core è anche molto più compatto, e molte delle sue funzionalità si trovano in altri package, opzionali. In questo modo è molto più semplice avere più versioni di .NET Core sulla stessa macchina e introdurre nuove funzionalità di .NET Core senza sollevare problemi di compatibilità.

Tuttavia, la biforcazione di .NET in Framework e Core ha anche le sue debolezze. Per far funzionare al meglio .NET Core come ambiente di programmazione multipiattaforma, alcuni aspetti delle librerie standard .NET sono stati modificati in modi che non sono del tutto compatibili con .NET Framework. Per risolvere questo problema, .NET Foundation ha introdotto il concetto di .NET Standard, un insieme di API offerte sia da .NET Framework sia da .NET Core. Il .NET Core SDK può quindi essere usato per creare librerie per .NET Core o per .NET Standard e per creare applicazioni per .NET Core. Molte delle librerie fornite nel Quantum Development Kit hanno come destinazione .NET Standard, in modo che i programmi Q# possano

essere utilizzati dalle tradizionali applicazioni .NET Framework o dalle nuove applicazioni create utilizzando il .NET Core SDK.
La versione più recente di .NET si chiama semplicemente ".NET 5", anche se segue la versione.NET Core 3.1. Con .NET 5 esiste una sola piattaforma .NET, invece di .NET Framework, .NET Core e .NET Standard, cosa che riduce gran parte della confusione. Per ulteriori informazioni, visitate `https://devblogs.microsoft.com/dotnet/introducing-net-5`.
Per ora, tuttavia, .NET Core 3.1 è ancora la versione a lungo termine più recente della piattaforma .NET, un aspetto che la rende la scelta migliore per scrivere software di produzione stabile, quanto meno fino all'uscita della prossima versione a lungo termine, .NET 6. Di conseguenza, il Quantum Development Kit è scritto per utilizzare il .NET Core SDK 3.1.

Installazione del .NET Core SDK

Per installare il .NET Core SDK, andate su `https://dotnet.microsoft.com/download/dotnet/3.1` e selezionate il vostro sistema operativo. Nella colonna "Build Apps – SDK", scaricate il programma di installazione per il vostro sistema operativo, che dovrebbe occuparsi di tutto.

Installazione dei template dei progetti

Una cosa che potrebbe essere diversa da quella cui siete abituati è che lo sviluppo .NET è incentrato sull'idea di un progetto che specifica come viene richiamato un compilatore per creare un nuovo file binario. Per esempio, un file di progetto C# (`*.csproj`) dice al compilatore C# quali file di codice sorgente dovrebbero essere compilati, quali librerie sono necessarie, quali warning sono attivati e disattivati e così via. In questo modo, i file di progetto funzionano in modo simile ai makefile o ad altri sistemi di gestione della compilazione. La grande differenza sta nel modo in cui i file .NET Core fanno riferimento alle librerie.

Q# riutilizza questa infrastruttura per semplificare l'ottenimento di nuove librerie per i programmi quantistici, come quelle elencate in https://docs.microsoft.com/it-it/azure/quantum/user-guide/libraries/additional-libraries o quelle rese disponibili dagli sviluppatori della community su https://qsharp.community/projects. Utilizzando questa infrastruttura, un file di progetto Q# può specificare uno o più riferimenti a package in `NuGet.org`, un repository di package per le librerie software. Ogni package può quindi fornire un certo numero di librerie. Quando viene compilato un progetto che dipende da un package NuGet, il .NET Core SDK scarica automaticamente il package corretto e quindi usa le librerie di quel package per compilare il progetto.

Dal punto di vista della programmazione quantistica, ciò consente di distribuire il Quantum Development Kit come un piccolo numero di package NuGet che possono essere installati non su una macchina, ma in ogni progetto. Ciò semplifica l'utilizzo di versioni diverse del Quantum Development Kit per progetti differenti o l'inclusione delle sole parti necessarie per un determinato progetto. Per aiutarvi a iniziare con un set ragionevole di package NuGet, il Quantum Development Kit viene fornito con i template per la creazione di nuovi progetti, che fanno riferimento a tutto ciò di cui potete aver bisogno.

> **SUGGERIMENTO**
>
> Se preferite lavorare all'interno di un IDE, l'estensione Visual Studio Code per Quantum Development Kit (vedi di seguito) può essere utilizzata anche per creare nuovi progetti.

Per installare i template dei progetti, lanciate il seguente comando:

```
dotnet new -i "Microsoft.Quantum.ProjectTemplates"
```

Una volta installati i template dei progetti, potete utilizzarli eseguendo di nuovo `dotnet new`:

```
dotnet new console -lang Q# -o ProjectName (1)
```

(1) Assicuratevi di sostituire a `ProjectName` il nome del progetto da creare.

Installazione dell'estensione di Visual Studio Code

Dopo aver installato Visual Studio Code da `https://code.visualstudio.com`, dovrete installare l'estensione Quantum Development Kit per ottenere il supporto dell'editor per Q#, comprendente il completamento automatico, l'evidenziazione degli errori di sintassi e così via.

Per installare l'estensione, aprite una nuova finestra di Visual Studio Code e premete Ctrl-Maiusc-X o Cmd-Maiusc-X per visualizzare la barra laterale delle estensioni. Nella barra di ricerca, digitate `Microsoft Quantum Development Kit` e fate clic sul pulsante Install. Dopo che Visual Studio Code avrà installato l'estensione, il pulsante Install si trasformerà in Reload. Fare clic per chiudere Visual Studio Code e riaprire la finestra, con l'estensione Quantum Development Kit installata. In alternativa, premete Ctrl-P o Cmd-P per visualizzare la finestra Go To, digitate `ext install quantum.quantum-devkit-vscode` e premete Invio.

In entrambi i casi, una volta installata l'estensione, per usarla, aprite una cartella (Ctrl-Maiusc-O o Cmd-Maiusc-O) contenente il progetto Q# su cui desiderate lavorare. A questo punto, dovreste avere tutto il necessario per iniziare a programmare con il Quantum Development Kit.

Per maggiori informazioni

- Documentazione del Quantum Development Kit: `https://docs.microsoft.com/azure/quantum`.
- Utilizzo del comando `dotnet`: `https://docs.microsoft.com/dotnet/core/tools/dotnet`.
- Introduzione a Visual Studio Code: `https://code.visualstudio.com/docs/introvideos/basics`.

Installazione di IQ# per Jupyter Notebook

Eseguite quanto segue dalla riga di comando:

```
dotnet tool install -g Microsoft.Quantum.IQSharp
dotnet iqsharp install
```

SUGGERIMENTO

In alcune distribuzioni di Linux, potrebbe essere necessario eseguire `dotnet iqsharp install --user`.

Ciò renderà Q# disponibile come linguaggio per i notebook Jupyter.

Glossario e riferimenti rapidi

Questa appendice ripresenta in modo sintetico molti dei concetti quantistici trattati nel libro e alcune nozioni di linguaggio Q# (versione 0.15). I contenuti di questa appendice sono stati perlopiù trattati nel corpo principale del libro, ma sono raccolti qui per comodità.

Glossario

- **adjoint**
  Un'operazione quantistica che inverte o annulla perfettamente l'azione di un'altra operazione quantistica. Le operazioni che sono anche le proprie adjoint, come `X` e `H`, sono dette self-adjoint.
  Se un'operazione può essere simulata dalla matrice unitaria U, la sua operazione adjoint può essere simulata dalla trasposizione complessa di U, nota anche come adjoint di U ed è scritta come U†. In Q#, le operazioni che hanno operazioni adjoint sono indicate con `is Adj`.
- **algoritmo**
  Una procedura per risolvere un problema, tipicamente specificata tramite una sequenza di passi.
- **autofase**
  La fase globale assegnata a un autostato da un'operazione quantistica. Per esempio, l'autostato $|-\rangle = (|0\rangle - |1\rangle) / \sqrt{2}$

dell'operazione x ha un'autofase di -1,poiché $X|-\rangle = (|1\rangle - |0\rangle) / \sqrt{2} = -|-\rangle$.

- **autostato**
  Uno stato che non viene modificato dall'applicazione di un'operazione quantistica, fino a una determinata fase globale. Per esempio, lo stato $|+\rangle$ è un autostato dell'operazione x, poiché $X|+\rangle = |+\rangle$.
- **autovalore**
  Data una matrice A, un numero λ è un autovalore di A se $A\vec{x} = \lambda \vec{x}$ per qualche vettore $\vec{x}$.
- **autovettore**
  Data una matrice A, un vettore $\vec{x}$ di A se $A\vec{x} = \lambda\vec{x}$ per qualche numero λ.
- **BB84**
  Abbreviazione di "Bennett and Brassard 1984". Un protocollo per la distribuzione quantistica delle chiavi (Quantum Key Distribution – QKD) inviando singoli qubit per volta.
- **bit classico**
  La più piccola unità funzionale di memorizzazione ed elaborazione in un computer classico. Un bit classico può trovarsi nello stato "0" o "1".
- **Born (regola)**
  Espressione matematica che può essere utilizzata per prevedere la probabilità di una misurazione quantistica, data una descrizione di tale misurazione e dello stato del registro misurato.
- **classico (computer)**
  Un computer convenzionale, che esegue i propri calcoli utilizzando le leggi della fisica classica.
- **complesso (numero)**
  Un numero nella forma $z = a + bi$, dove $i^2 = -1$.

- **computer**
  Un dispositivo che prende in input dei dati e svolge qualche tipo di operazione su tali dati.
- **computer quantistico**
  Un dispositivo quantistico progettato e utilizzato per risolvere problemi computazionali difficili per i computer classici.
- **controllata (operazione)**
  Un'operazione quantistica applicata in base allo stato di un registro di controllo senza misurazione, in modo tale che le sovrapposizioni siano correttamente preservate. Per esempio, l'operazione `CNOT` è un'operazione NOT controllata. Allo stesso modo, l'operazione Fredkin è un'operazione di SWAP controllata. In Q#, le operazioni che possono essere controllate sono indicate con `is Ctl`.
- **coprìmi**
  Due numeri interi positivi che non condividono fattori primi. Per esempio, $21 = 3 \times 7$ e $10 = 2 \times 5$ sono coprìmi, mentre $21 = 3 \times 7$ e $15 = 3 \times 5$ condividono il fattore 3 e quindi non sono coprìmi.
- **dispositivo quantistico**
  Un sistema quantistico costruito per raggiungere uno scopo o per eseguire un compito.
- **entanglement**
  Quando gli stati di due o più qubit non possono essere scritti indipendentemente. Per esempio, se due qubit sono nello stato $(|00\rangle + |11\rangle) / \sqrt{2}$, allora non ci sono due stati mono-qubit $|\psi\rangle$ e $|\varphi\rangle$ tali che $(|00\rangle + |11\rangle) / \sqrt{2} = |\psi\rangle \otimes |\varphi\rangle$, e i due qubit si dicono entangled.
- **fase**
  Un numero complesso di grandezza 1 (cioè, a + bi, dove $|a|^2 + |b|^2 = 1$). Una fase può essere scritta come $e^{i}\theta$, dove θ è un numero

reale. Notate che, come scorciatoia, quando è chiaro dal contesto, a volte θ stesso è noto come fase.

- **fase globale**
  Due stati quantistici che sono uguali fino alla moltiplicazione per un numero complesso di grandezza 1 differiscono di una fase globale. In quel caso, i due stati sono completamente equivalenti. Per esempio, (|0⟩ - |1⟩) / √2 e (|1⟩ - |0⟩) / √2 rappresentano lo stesso stato, in quanto differiscono per una fase di -1 = $e^{i\pi}$.
- **matrice unitaria**
  Una matrice U tale che UU† = 𝟙, dove U† è la coniugata trasposta, o la adjoint di U. Simile a una tabella di verità classica, una matrice unitaria descrive il modo in cui un'operazione quantistica trasforma lo stato del suo input così che può essere utilizzata per simulare tale operazione per input arbitrari.
- **misurazione**
  Un'operazione quantistica che restituisce dati classici relativi allo stato di un registro quantistico.
- **operazione quantistica**
  Una subroutine in un programma quantistico, che rappresenta una sequenza di istruzioni inviate a un dispositivo quantistico e un flusso di controllo classico. Alcune operazioni quantistiche, come `X` e `H` sono incorporate nel dispositivo quantistico e dunque sono intrinseche.
- **operazione unitaria**
  Un'operazione quantistica che può essere rappresentata da una matrice unitaria. In Q#, le operazioni unitarie sono adjoint-abili e controllabili (`is Adj + Ctl`).
- **oracolo**
  Un'operazione quantistica che implementa una funzione classica applicata ai registri quantistici.

- **Pauli (matrici di)**
  Le matrici unitarie mono-qubit, X, Y e Z.
- **phase kickback**
  Una tecnica di programmazione quantistica per associare la fase applicata da un'operazione quantistica controllata allo stato del registro `control` invece che allo stato del registro `target`. Questa tecnica può essere utilizzata per convertire quella che altrimenti sarebbe una fase globale applicata da un'operazione unitaria in una fase osservabile fisicamente.
- **programma**
  Una sequenza di istruzioni che può essere interpretata da un computer classico per eseguire il compito desiderato.
- **programma quantistico**
  Un programma classico che controlla un dispositivo quantistico inviando istruzioni a tale dispositivo ed elaborando i dati di misurazione restituiti dal dispositivo. In genere, i programmi quantistici sono scritti in un linguaggio di programmazione quantistico come Q#.
- **Quantum Key Distribution (QKD)**
  Un protocollo di comunicazione che ha lo scopo di condividere numeri casuali tra due parti. Quando è eseguito da dispositivi che funzionano correttamente, la sicurezza del protocollo è garantita dalla meccanica quantistica (in particolare, dal teorema di non clonazione).
- **qubit**
  La più piccola unità funzionale di un computer quantistico. Un singolo qubit può essere nello stato $|0\rangle$, nello stato $|1\rangle$ o in qualsiasi sovrapposizione.
- **registro quantistico**
  Una raccolta di qubit. Il registro può essere in qualsiasi stato

computazionale di base, etichettato da stringhe di bit classici o da qualsiasi loro sovrapposizione.

- **reversibile**
  Una funzione classica che può essere perfettamente invertita. Per esempio, f(x) = ¬x può essere invertita, poiché f(f(x)) = x. Allo stesso modo, g(x, y) = (x, x $\oplus$ y) è reversibile, poiché g(g(x, y)) = (x, y). Al contrario, h(x, y) = (x, x AND y) non è reversibile, poiché h(0, 0) = h(0, 1) = (0, 0); in tal modo non possiamo determinare l'input di h dato l'output (0, 0).
- **semiprimo**
  Un numero intero positivo avente esattamente due fattori primi. Per esempio, 21 = 3 × 7 è semiprimo, mentre 105 = 3 × 5 × 7 ha tre fattori primi e quindi non è semiprimo.
- **sistema quantistico**
  Un sistema fisico che richiede la meccanica quantistica per essere descritto e simulato.
- **sovrapposizione**
  Uno stato quantistico che può essere scritto come una combinazione lineare di altri stati si trova in una sovrapposizione di quegli stati. Per esempio, $|+\rangle = (|0\rangle + |1\rangle) / \sqrt{2}$ è una sovrapposizione degli stati $|0\rangle$ e $|1\rangle$, mentre $|0\rangle$ è una sovrapposizione di $|+\rangle$ e $|-\rangle = (|0\rangle - |1\rangle) / \sqrt{2}$.
- **stato**
  Una descrizione di un sistema o dispositivo fisico sufficientemente completa da consentire la sua simulazione.
- **stato computazionale di base**
  Uno stato etichettato da una stringa di bit classici. Per esempio, $|01101\rangle$ è uno stato computazionale di base su un registro di cinque qubit.

- **stato quantistico**
  Lo stato di un registro quantistico (ovvero di un registro di qubit), tipicamente scritto come un vettore di 2^n numeri complessi, dove n è il numero di qubit contenuti nel registro.
- **stima della fase**
  Qualsiasi algoritmo quantistico per l'apprendimento dell'autofase associata a un dato autostato di un'operazione quantistica.
- **teorema di non clonazione**
  Dimostra che non può esistere un'operazione quantistica che copi perfettamente l'informazione quantistica. Per esempio, è impossibile eseguire un'operazione che trasforma lo stato $|\psi\rangle \otimes |0\rangle$ in $|\psi\rangle \otimes |\psi\rangle$ per stati quantistici arbitrari $|\psi\rangle$.

Notazione di Dirac

Spesso, nel quantum computing, usiamo una notazione compressa per vettori e matrici nota come notazione di Dirac. L'argomento è trattato dettagliatamente in tutto il libro, ma alcuni punti chiave della notazione di Dirac sono riassunti nella seguente tabella.

Notazione di Dirac	Notazione a matrici

Notazione di Dirac	Notazione a matrici

Operazioni quantistiche

Questo paragrafo riassume alcune operazioni quantistiche comuni che avete trovato in tutto il libro. In particolare, mostriamo come richiamare ogni operazione da Q# passando dei qubit come input, come simulare tale operazione in QuTiP con una matrice che agisce sugli stati, la matrice unitaria che usiamo per simulare ogni operazione e alcuni esempi del modo in cui tale operazione si comporta matematicamente.

> **NOTA**
> Per un elenco completo di tutte le operazioni standard del linguaggio Q#, vedi il riferimento all'API di Q# su `https://docs.microsoft.com/qsharp/api`.

Per tutti gli esempi di codice Q#, abbiamo presupposto la seguente dichiarazione `open`:

```
open Microsoft.Quantum.Intrinsic;
```

Per tutti gli esempi di codice Python/QuTiP, abbiamo presupposto le seguenti istruzioni `import`:

```
import qutip as qt

import qutip.qip.operations as qtops
```

La seconda convenzione non è usata nel libro, ma è impiegata qui per brevità.

Le operazioni Q# operano su qubit, mentre QuTiP rappresenta le operazioni per matrici unitarie che moltiplicano stati. Pertanto, a

differenza delle operazioni Q#, gli oggetti QuTiP non elencano esplicitamente i propri input.

Descrizione	Codice (Q# e QuTiP)	Matrice unitaria	Esempi matematici

Inversione dei bit (X di Pauli)	`X(target); // Q#`
	`qt.sigmax() # QuTiP`		

Inversione di bit e di fase (Y di Pauli)	`Y(target); // Q#`
	`qt.sigmay() # QuTiP`		

Inversione di fase (Z di Pauli)	`Z(target); // Q#`
	`qt.sigmaz() # QuTiP`		

Hadamard	`H(target); // Q#`
	`qtops.hadamard_transform() // # QuTiP`		

Controlled-NOT (CNOT)	`CNOT(control, target); // Q# (abbreviata)`
	`ControlledX([control], target); // Q#`
	`qtops.cnot() # QuTiP`		

CCNOT (Toffoli)	`CCNOT(control1, control2, target); // Q# (abbreviata)`
	`ControlledX([control1, control2], target); // Q#`
	`qtops.toffoli() # QuTiP`		

Rotazione X	`Rx(angle, target); // Q#`
	`qtops.rx(angle) # QuTiP`		

Rotazione Y	`Ry(angle, target); // Q#`
	`qtops.ry(angle) # QuTiP`		

Rotazione Z	`Rz(angle, target); // Q#`
	`qtops.rz(angle) # QuTiP`		

Misurazione di un singolo qubit	`M(target); // Q#`
	`n/a`
		`n/d`	`n/d`

Linguaggio Q#

Tipi

Nella tabella seguente, utilizziamo il *`carattere corsivo a spaziatura fissa`* per indicare un segnaposto. Per esempio, il segnaposto *`TipoBase`* in *`TipoBase`*`[]` può essere `Int`, `Double`, `Qubit`, `(Qubit, Qubit[])` o qualsiasi altro tipo Q#.

Per enfasi, in alcuni degli esempi abbiamo aggiunto il tipo di ogni esempio come annotazione dopo ogni valore. Per esempio, `Sin : Double -> Double` indica che `Sin` è un valore il cui tipo è `Double -> Double`.

Descrizione	Tipo Q#	Esempi
Interi	`Int`	`3 -42 108`
Virgola mobile	`Double`	`-3.1415 2.17`
Booleani	`Bool`	`true false`
Intervalli di interi	`Range`	`0..3` `0..Length(arr)12..-1..0`
Tupla vuota	`Unit`	`()`

Risultati di una misurazione	`Result`	`Zero One`
Operatori di Pauli	`Pauli`	`PauliI PauliX PauliY PauliZ`
Stringhe	`String`	`"Hello, world!"`
Qubit	`Qubit`	(Vedi l'istruzione `use`)
Array	*`TipoBase`*`[]`	`new Qubit[0][42, -101]`
Tuple	`(`*`T1`*`), (`*`T1`*`, `*`T2`*`), (`*`T1`*`, `*`T2`*`, `*`T3`*`)` e così via `(PauliX, "X") (1, true, Zero)`	
Funzioni	*`TipoInput`* `->` *`OutputType`*	`Sin : Double -> Double` `Message : String -> Unit`
Operazioni	*`TipoInput`* `=>` *`TipoOutput TipoInput`* `=> Unit is Adj` (se adjoint-abile) *`TipoInput`* `=> Unit is Ctl` (se controllabile)*`TipoInput`* `=> Unit is Adj + Ctl` (se adjoint-abile e controllabile)	`H : Qubit => Unit is Adj + Ctl CNOT : (Qubit, Qubit) => Unit is Adj + Ctl M : Qubit => Result Measure : (Pauli[], Qubit[]) => Result`

Dichiarazioni e istruzioni Q#

Nella tabella seguente, utilizziamo il *`carattere corsivo a spaziatura fissa`* per indicare un segnaposto. Per esempio, il segnaposto *`NomeFunzione`* nella funzione *`NomeFunzione`*`(input1: `*`TipoInput1`*`): `*`TipoOutput`* rappresenta il nome della funzione da definire, mentre i segnaposto *`TipoInput1`* e *`TipoOutput`* rappresentano i tipi della tabella precedente.

Descrizione	Sintassi Q#

Descrizione	Sintassi Q#
Commento fino a fine riga	`// testo del commento`
Commento di documentazione (prima dell'operazione o della funzione)	`/// # Summary /// testo di riepilogo ////// # Description /// testo descittivo ////// # Input /// ## input1 /// descrizione degli input`
Dichiarazione di namespace	`namespace NomeNamespace { // ...}`
Dichiarazione di funzione	`function NomeFunzione( input1 : TipoInput1, input2 : TipoInput2, ...) : TipoOutput { // corpo della funzione }`
Dichiarazione di operazione	`operation NomeOperazione( input1 : TipoInput1, input2 : TipoInput2, ...) : TipoOutput { // corpo dell'operazione }`
Dichiarazione di operazione (adjoint-abile e controllabile)	`operation NomeOperazione( input1 : TipoInput1, input2 : TipoInput2, ...) : Unit is Adj + Ctl { // corpo dell'operazione }`
Dichiarazione di tipo definito dall'utente	`newtype NomeTipo = ( TipoElemento1, TipoElemento2, ...);`
Dichiarazione di tipo definito dall'utente con elementi denominati	`newtype NomeTipo = ( NomeElemento1: TipoElemento1, NomeElemento2: TipoElemento2, ...);`
Apertura namespace (rende disponibili in un file Q# o in una cella del notebook gli elementi nel namespace)	`open NomeNamespace;`

Descrizione	Sintassi Q#
Apertura namespace con alias	`open` *`NomeNamespace`* `as` *`NomeAlias`*`;` Esempio: `open Microsoft.Quantum.Diagnostics as Diag;`
Dichiarazione di variabile locale	`let` *`NomeVariabile`* `=` *`Valore`*`;` Esempio:`let foo = "Bar";`
Dichiarazione di variabile mutabile	`mutable` *`NomeVariabile`* `=` *`Valore`*`;`
Riassegnazione (aggiornamento) di variabile mutabile	`set` *`NomeVariabile`* `=` *`NuovoValore`*`;`
Applicazione e riassegnamento di variabile mutabile	`set` *`NomeVariabile operatore=`* *`Espressione`*`;` Esempi:`set count += 1;set array w/= 2 <- PauliX;`(vedi operatore `w/`)
Istruzione condizionale classica	`if` *`Condizione`* `{ // ...}if` *`Condizione`* `{ // ...} else { // ...}if` *`Condizione`* `{ // ...} elseif` *`Condizione`* `{ // ...} else { // ...}`
Iterazione su array	`for` *`Elemento`* `in` *`NomeArray`* `{ // corpo del ciclo }` Nota: *`NomeArray`* deve essere un valore di tipo array.
Iterazione su intervallo	`for` *`Indice`* `in` *`NomeRange`* `{ // corpo del ciclo }` Nota: *`NomeRange`* deve essere un valore di tipo `Range`.
Ciclo `repeat-until`-successo	`loop repeat { // corpo del ciclo }until` *`Condizione`*`;`

Descrizione	Sintassi Q#
Ciclo `repeat-until`-successo con blocco `fixup`	`repeat { // corpo del ciclo }until` *`Condizione`* `fixup { // corpo del fixup }`
Ciclo `while` (solo nelle funzioni)	`while` *`Condizione`* `{ // corpo del ciclo }`
Termina con errore	`fail "messaggio";`
Restituzione del valore da funzione o operazione	`return` *`Valore`*`;`
Applicazione modello “calze e scarpe” (vedi Capitolo 7)	`within { // corpo esterno } apply { // corpo interno }`
Allocazione nuovo qubit (solo nelle operazioni)	`use` *`Nome`* `= Qubit();`
Allocazione di array di qubit (solo nelle operazioni)	`use` *`Nome`* `= Qubit[`*`dimenzioni`*`];`
Allocazione di una tupla di qubit e registri (solo nelle operazioni)	`use (`*`Nome1`*`,` *`Nome2`*`, ...) = (`*`QubitOArray1`*`,` *`QubitOArray2`*`, ...);`
Allocazione di un qubit con scope esplicito (solo nelle operazioni)	`use` *`Nome`* `= Qubit() { // ...` *`// Nome`* `viene rilasciato qui.}`

Espressioni e operatori Q#

Nella tabella seguente, utilizziamo il *`carattere corsivo a spaziatura fissa`* per indicare un segnaposto. Per esempio, il segnaposto *`Tipo`* in `new` *`Tipo`* `[`*`lunghezza`*`]` indica il tipo base del nuovo array, mentre *`lunghezza`* indica il numero di elementi dell’array.

```
$"Measurement result was {result}"
[10, 100, 1000] w/ 1 <- 200 // [10, 200, 1000]
let imagUnit = Complex(0.0, 1.0);
Message($"{imagUnit::Real}");
// mostra 1.0
```

```
let imagUnit = Complex(0.0, 1.0);
Message($"{imagUnit!}");
// mostra (0.0, 1.0)
let imagUnit = Complex(0.0, 1.0);
let onePlusI = imagUnit w/ Real <- 1.0;
Message($"{onePlusI!}");
// mostra (1.0, 1.0)
```

Descrizione	Sintassi Q#
Operatori aritmetici	`+`, `-`, `*`, ...

Formattazione di valori come una stringa	`$"... {espressione} ..."`
	Esempio:
	`$"Measurement result was {result}"`

Concatenamento di due array	`array1 + array2`
Allocazione array	`new Tipo[lunghezza]`
Estrazione di un elemento di un array	`array[indice]`

Estrazione di parti di un array	`array[inizio...]`
	`array[...fine]`
	`array[inizio..fine]`
	`array[inizio..passo..fine]`

Copia e aggiornamento degli elementi di un array	`array w/ indice <- nuovoValore`
	Esempio:
	`[10, 100, 1000] w/ 1 <- 200`
	`// [10, 200, 1000]`

Accesso a un elemento per nome in un tipo definito dall'utente	`valore::nomeElemento`

Descrizione	Sintassi Q#

Estrae un valore da un tipo definito dall'utente	*valore*!
	Esempio:
	`let imagUnit = Complex(0.0, 1.0);`
	`Message($"{imagUnit!}");`
	`// mostra (0.0, 1.0)`

Copia e aggiornamento di un elemento con nome da un tipo definito dall'utente	*valore* `w/` *nomeElemento* `<-` *nuovoValore*
	Esempio:
	`let imagUnit = Complex(0.0, 1.0);`
	`let onePlusI = imagUnit w/ Real <- 1.0;`
	`Message($"{onePlusI!}");`
	`// mostra (1.0, 1.0)`

Librerie standard Q#

Abbiamo presupposto l'impiego delle seguenti dichiarazioni `open`:

```
open Microsoft.Quantum.Intrinsic;
open Microsoft.Quantum.Canon;
open Microsoft.Quantum.Arrays as Arrays;

open Microsoft.Quantum.Diagnostics as Diag;
```

Per un elenco completo delle funzioni, operazioni e tipi di Q#, vedi https://docs.microsoft.com/qsharp/api.

I suffissi `A`, `C` e `CA` indicano, rispettivamente, le operazioni che supportano input che sono adjoint-abili, controllabili o entrambe le cose.

- fisicamente possibile solo su simulatori;
- non esegue la misurazione, lasciando il qubit immutato.

Descrizione	Funzione o operazione	Esempi
Applica un'operazione a ogni elemento di un array	`ApplyToEachCA`	`ApplyToEachCA(H, register);`
Richiama un'operazione più volte	`Repeat`	`Repeat(PrintRandomNumber, 10, ());`
Restituisce il primo o il secondo elemento di una coppia	`Fst` o `Snd`	• `Fst((1.0, false)) // 1.0` • `Snd((1.0, false)) // false`
Applica un'operazione a ogni elemento di un array e raccoglie i risultati	`Arrays.ForEach`	`let results = ForEach(M, register);`
Richiama una funzione con ogni elemento di un array e raccoglie i risultati	`Arrays.Mapped`	`let sines = Mapped(Sin, angles);`
Fallisce se una condizione è falsa	`Diag.Fact`	`Fact(2 == 2, "Expected two to equal two.");`
Fallisce se due risultati di misurazioni non sono uguali	`Diag.EqualityFactR`	`EqualityFactR(M(qubit), Zero, "Expected qubit to be in |0⟩ state.");`

Descrizione	Funzione o operazione	Esempi
Fallisce se una misurazione ipotetica non fornisce il risultato desiderato. Note: - fisicamente possibile solo su simulatori; - non esegue la misurazione, lasciando il qubit immutato.	`Diag.AssertMeasurement`	`AssertMeasement([PauliZ], [target], Zero, "Expected qubit to be in |0⟩ state.");`
Fallisce se due operazioni sono differenti	`Diag.AssertOperations EqualReferenced`	`AssertOperationsEqual Referenced(2, actualOperation, expectedOperation);`
Chiede al simulatore di mostrare informazioni diagnostiche su tutti i qubit allocati	`Diag.DumpMachine`	`DumpMachine();`
Chiede al simulatore di mostrare informazioni diagnostiche su un determinato registro	`Diag.DumpRegister`	`DumpRegister((), register);`

Comandi magici QI#

Per un elenco completo dei comandi magici IQ#, vedi https://docs.microsoft.com/qsharp/api/iqsharp-magic.

Descrizione	Comando magico	Esempio
Simula una funzione o un'operazione	`%simulate`	`%simulate PlayMorganasGame winProbability=0.999`
Aggiunge un nuovo package alla sessione IQ#	`%package`	`%package Microsoft.Quantum.Numerics`
Ricarica i file Q# con i package correnti	`%workspace reload`	
Elenca tutti i comandi magici disponibili	`%lsmagic`	
Elenca i namespace attualmente aperti	`%lsopen`	
Esegue lo stimatore di risorse su un'operazione	`%estimate`	`%estimate FindMarkedItem`
Imposta le opzioni di configurazione di IQ#	`%config`	`%config dump.truncateSmallAmplitudes = true`
Elenca tutte le funzioni e le operazioni attualmente definite	`%who`	

Ripasso di algebra lineare

In questa appendice, l'obiettivo è quello di trattare rapidamente alcune delle conoscenze di algebra lineare utili per questo libro. Discuteremo di vettori e matrici, vedremo come impiegarli per rappresentare funzioni lineari, e come utilizzare Python e NumPy per lavorare con vettori e matrici.

I vettori

Prima di poter parlare di qubit, dobbiamo comprendere il concetto di vettore.

> Supponiamo che un nostro amico ci inviti a casa sua per festeggiare la riparazione del citofono. A noi piacerebbe molto trovare casa sua e festeggiare. Come può il nostro amico aiutarci a trovare casa sua?

Il problema è rappresentato nella Figura C.1. I vettori sono uno strumento matematico che può essere utilizzato per rappresentare una varietà di concetti, in pratica tutto ciò che possiamo registrare creando un elenco ordinato di numeri:

- punti su una mappa;
- colori dei pixel in un display;
- elementi danneggiati in un videogame;
- velocità di un aeroplano;
- orientamento di un giroscopio.

Per esempio, se ci perdiamo in una città sconosciuta, qualcuno può dirci dove andare dandoci un "vettore", che ci indica di andare per a isolati a est e poi per b isolati a nord. Scriviamo queste istruzioni con il vettore `[[a], [b]]` (Figura C.2).

Come si fa per i normali numeri, possiamo anche sommare più vettori.

Figura C.1 Alla ricerca della festa in casa del nostro amico.

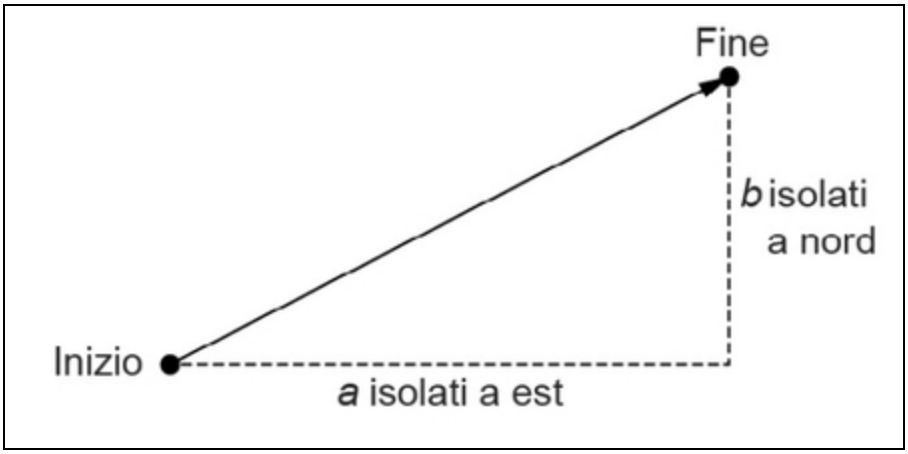


Figura C.2 Vettori come coordinate.

NOTA

Un numero semplice è spesso chiamato *scalare*, per distinguerlo da un vettore.

Usando questo ragionamento sui vettori, possiamo pensare che l'addizione di vettori sia definita dal punto di vista degli elementi. Cioè, interpretiamo `[[a], [b]] + [[c], [d]]` come istruzioni per andare per `a` isolati a est, per `b` isolati a nord, per `c` isolati a est, infine per `d` isolati a nord. Poiché non importa in quale ordine procediamo, ciò equivale a procedere per `a + c` isolati a est e poi per `b + d` isolati a nord, quindi scriviamo che `[[a], [b]] + [[c], [d]]` è `[[a + c], [b + d]]` (Figura C.3).

Un vettore $\vec{v}$ in *d* dimensioni può essere scritto come un elenco di *d* numeri. Per esempio, $\vec{v}$ = [[2], [3]] è un vettore in 2 dimensioni (Figura C.4).

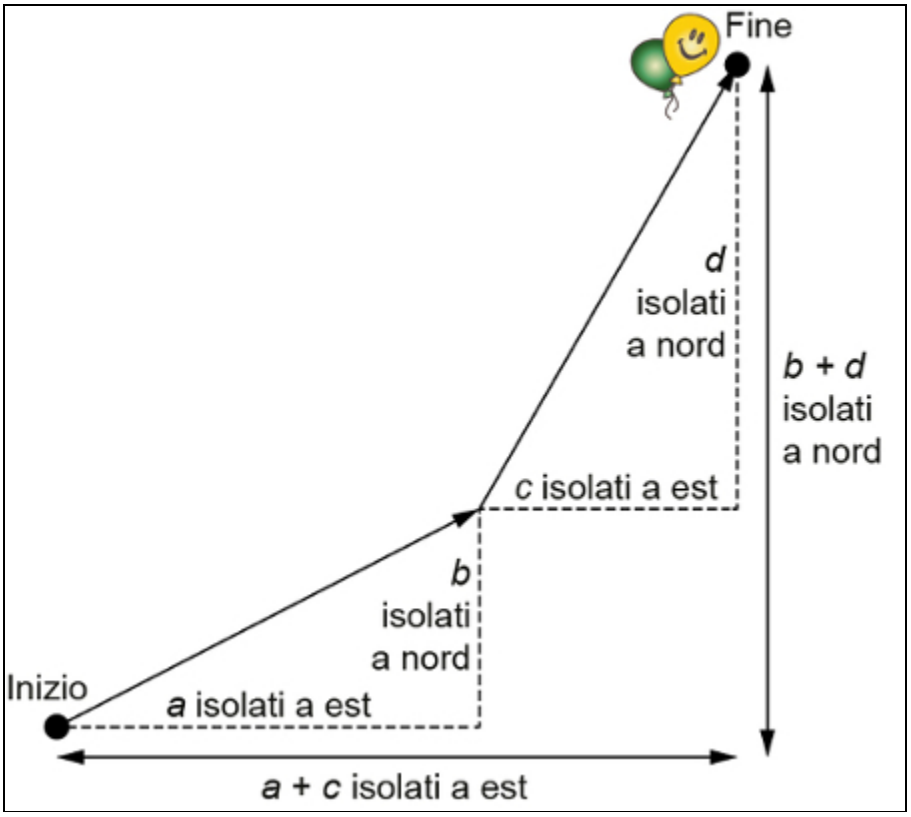


Figura C.3 Somma di vettori per trovare la festa.

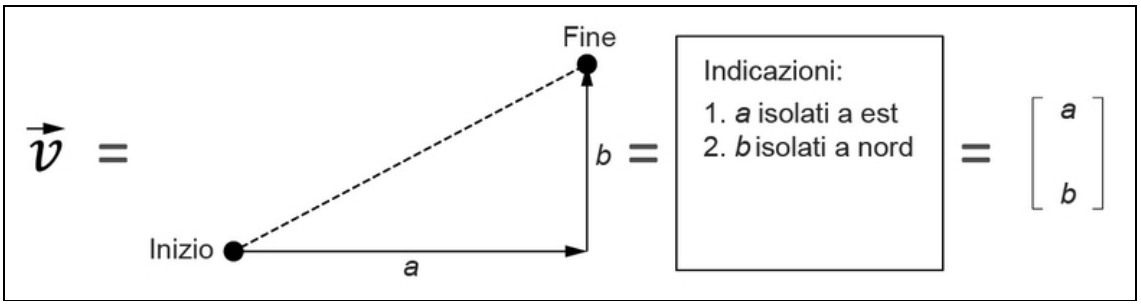


Figura C.4 I vettori tracciati offrono le stesse informazioni di un elenco di direzioni o di una colonna di numeri.

Allo stesso modo, per trasformare i vettori possiamo moltiplicare i vettori per numeri semplici. Per esempio, potremmo esserci persi non in una città qualsiasi, ma in una città che usa miglia e piedi come unità di misura. Per trasformare un vettore in metri in un vettore in piedi,

dobbiamo moltiplicare ogni elemento del nostro vettore per circa 3,28. Possiamo farlo utilizzando la libreria Python NumPy, per aiutarci a gestire il modo in cui rappresentiamo i vettori in un computer.

SUGGERIMENTO

Le istruzioni per l'installazione si trovano nell'Appendice A.

Listato C.1 Rappresentazione di vettori in Python con NumPy.

```
>>> import numpy as np
>>> directions_in_meters = np.array( (1)
...     [[30], [50]]) (2)
>>> directions_in_feet = 3.28 * directions_in_meters (3)
>>> directions_in_feet
array([[ 98.4], (4)
[164. ]])
```

(1) I vettori sono un caso speciale di array NumPy. Creiamo array utilizzando la funzione `array`, passando una lista di righe per il vettore. Ogni riga, poi, è una lista di colonne: per i vettori, abbiamo sempre e solo una colonna per riga, ma nel libro abbiamo esempi più complessi.

(2) Inizia procedendo per 30 metri verso est e poi per 50 metri verso nord.

(3) NumPy rappresenta la moltiplicazione tra numeri scalari e vettori con l'operatore di moltiplicazione Python `*`.

(4) Mostra il risultato della moltiplicazione. Cambiando sistema dobbiamo procedere per 98,4 piedi verso est e poi per 164 piedi verso nord.

ESERCIZIO C.1 CONVERSIONE DI UNITÀ

Come calcolare 25 metri verso ovest e 110 metri verso nord in piedi?

SOLUZIONI DEGLI ESERCIZI

Tutte le soluzioni degli esercizi di questo libro si trovano nel suo archivio di codice: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Aprite la cartella del capitolo in cui vi trovate e poi il notebook Jupyter con il nome che menziona le soluzioni degli esercizi.

Questa struttura aiuta a comunicare le direzioni. Se non usassimo i vettori, ogni scalare avrebbe bisogno della propria direzione e sarebbe fondamentale mantenere insieme le direzioni e i numeri scalari.

Le matrici

Come vedremo tra poco, possiamo descrivere il modo in cui i qubit si trasformano mentre applichiamo loro determinate istruzioni nello stesso modo in cui descriviamo la trasformazione dei vettori, usando un concetto dell'algebra lineare chiamato matrice. Questo diventa particolarmente importante nel momento in cui consideriamo trasformazioni di vettori che sono più complicate dell'addizione o del ridimensionamento.

Per vedere come usare le matrici, torniamo al problema di trovare la festa: quel campanello non suonerà da solo, dopotutto! Finora, abbiamo semplicemente ipotizzato che il primo componente di ogni vettore significhi est e il secondo significhi nord, ma qualcuno avrebbe potuto scegliere un'altra convenzione. Senza un modo per conciliare la trasformazione tra queste due convenzioni, non troveremo mai la festa. Fortunatamente, le matrici non solo ci aiuteranno a modellare i qubit più avanti in questa appendice, ma possono anche aiutarci a trovare la strada per raggiungere i nostri amici.

Fortunatamente, la trasformazione tra elencare prima il nord o prima l'est è semplice da implementare: basta scambiare le coordinate `[[a], [b]]` per ottenere `[[b], [a]]`. Supponiamo che questo scambio sia implementato da una funzione `swap`. Tale funzione opera correttamente con l'addizione vettoriale che abbiamo visto in precedenza, in quanto `swap(v + w)` è la stessa cosa di `swap(v) + swap(w)`. Allo stesso modo, se allunghiamo un vettore e poi lo scambiamo (ovvero eseguiamo una moltiplicazione scalare), è come se lo avessimo prima scambiato e poi

allungato: `swap(a * v) = a * swap(v)`. Qualsiasi funzione che abbia queste due proprietà è una funzione lineare.

Una funzione lineare è una funzione f tale che f(ax + by) = af(x) + bf(y) per tutti i numeri scalari a e b e tutti i vettori x e y.

Le funzioni lineari sono comuni nella computer graphics e nel machine learning, poiché includono vari modi diversi per trasformare i vettori numerici.

Ecco alcuni esempi di funzioni lineari:

- rotazioni;
- ridimensionamento e allungamento;
- riflessioni.

ESERCIZIO C.2 CONTROLLO DELLA LINEARITÀ

Quali delle seguenti funzioni sono lineari?

$f(x) = 2^x$

$f(x) = x^2$

$f(x) = 2x$

Tutte queste funzioni lineari hanno in comune il fatto che possiamo separarle ed esaminarle pezzo per pezzo. Ripensando alla mappa, se stiamo cercando di trovare la strada per la festa (dove speriamo sia rimasto ancora qualcosa) e la mappa che ci è stata data è stata allungata del 10% lungo la direzione nord-sud e capovolta lungo la direzione est-ovest, non sarà troppo difficile da capire. Poiché sia l'allungamento sia la riflessione sono funzioni lineari, qualcuno può metterci sulla giusta strada raccontandoci quello che è successo alla direzione nord-sud e alla direzione est-ovest. In effetti, l'abbiamo appena fatto all'inizio di questo paragrafo.

SUGGERIMENTO

Se doveste imparare anche solo una cosa da questo libro, il singolo argomento più importante è che potete comprendere le funzioni lineari, e quindi le operazioni quantistiche, suddividendole in componenti. Vedremo nel libro che,

poiché le operazioni del quantum computing sono descritte da funzioni lineari, possiamo comprendere gli algoritmi quantistici suddividendoli nello stesso modo in cui abbiamo suddiviso il nostro esempio della mappa. Non preoccupatevi se al momento questo discorso non ha molto senso: è un altro modo di pensare, e ci vuole un po' per abituarsi.

Questo perché, una volta abbiamo capito che cosa accade al vettore nord (chiamiamolo `[[1], [0]]`, come prima) e al vettore ovest (chiamiamolo `[[0], [1]]`), possiamo calcolare che cosa accade a tutti i vettori, usando la proprietà della linearità. Per esempio, se ci viene detto che c'è un angolo davvero pittoresco a tre isolati verso nord e a quattro isolati verso ovest rispetto a noi, e vogliamo capire dove si trova tale punto sulla nostra mappa, possiamo farlo pezzo dopo pezzo:

- dobbiamo allungare il vettore nord del 10% e moltiplicarlo per 3, ottenendo `[[3.3], [0]]`;
- dobbiamo invertire il vettore ovest e moltiplicarlo per 4, ottenendo `[[0], [-4]]`;
- terminiamo sommando ciò che accade a ciascuna direzione, ottenendo `[[3.3], [-4]]`.

Le funzioni lineari sono speciali 💔

Nell'esempio precedente, siamo stati in grado di allungare i nostri vettori utilizzando una funzione lineare. Questo perché le funzioni lineari non sono sensibili alla scala. Lo scambio nord-sud ed est-ovest fa la stessa cosa per i vettori, siano essi rappresentati in passi, isolati, metri, miglia, stadi o parsec. Tuttavia, questo non è ciò che capita alla maggior parte delle funzioni. Considerate una funzione eleva al quadrato il suo input, $f(x) = x^2$. Più grande è x, più viene aumentato.

Le funzioni lineari funzionano allo stesso modo, indipendentemente da quanto grandi o piccoli siano i loro input, che è esattamente ciò che ci consente di scomporle pezzo per pezzo: una volta che sappiamo come funziona una funzione lineare a qualsiasi scala, sappiamo come funziona a tutte le scale.

Quindi, sulla mappa dobbiamo recarci a 3,3 isolati a nord e 4 isolati a est.

Più avanti, vedremo come i bit “0” e “1” possono essere considerati come direzioni o vettori, un concetto non troppo diverso dal nord o dall’est. Allo stesso modo in cui il nord e l’est non sono i migliori vettori per aiutarvi a capire Minneapolis, scopriremo che “0” e “1” non sono sempre i migliori vettori per aiutare a comprendere il quantum computing (Figura C.5).

Figura C.5 Il nord e ovest non sono sempre le migliori direzioni per capire dove stiamo andando. In questa mappa del centro di Minneapolis, un'ampia sezione delle vie del centro è stata ruotata per seguire l'ansa del fiume Mississippi. Foto di davecito.

Questo modo di intendere le funzioni lineari suddividendole pezzo per pezzo funziona anche per le rotazioni. Se la nostra mappa avesse la bussola ruotata di 45° in senso orario, così che il nord punti a nord-est e l'ovest punti a nord-ovest, possiamo ancora raccapezzarci. Impiegando lo

stesso esempio, il vettore nord è ora mappato a circa `[[0.707], [0.707]]` sulla mappa, e il vettore ovest è mappato a `[[-0,707], [0.707]]`.

Quando riassumiamo ciò che accade nell'esempio: otteniamo `3 * [[0.707], [0.707]] + 4 * [[-0.707], [0.707]]`, che è uguale a `(3 + 4) * [[0.707], [0]] + (3 + 4) [[0], [0.707]]`, ovvero `[[-0.707], [4.95]]`. Potrebbe sembrare che questo abbia poco a che fare con la linearità e più con il fatto che nord e l'ovest siano in qualche modo "speciali". Tuttavia, avremmo potuto trattare esattamente lo stesso argomento ma indicando il sud-ovest con `[[1], [0]]` e il nord-ovest come `[[0], [1]]`. Questo funziona perché sud-ovest e nord-ovest sono perpendicolari l'uno all'altro, cosa che ci permette di esprimere qualsiasi altra direzione come una combinazione di nord-ovest e sud-ovest. Oltre alla facilità di lettura della bussola, non c'è nulla che renda davvero speciali il nord o l'ovest. Se immaginate di guidare nel centro di Minneapolis (vedi Figura C.5), diventa subito evidente che nord e ovest non sono sempre il modo migliore per esprimere le direzioni.

Formalmente, qualsiasi set di vettori che ci permettono di esprimere le direzioni suddividendole pezzo per pezzo in questo modo è chiamato base.

NOTA

Tecnicamente, qui ci interessa ciò che i matematici chiamano *base ortonormale*, che molto spesso è utile nel quantum computing. Tutto ciò significa che i vettori espressi in una base sono perpendicolari a tutti gli altri vettori di base, e hanno una lunghezza pari a 1.

Proviamo a scrivere un vettore in termini di base. Il vettore $\vec{v}$ = [[2], [3]] può essere scritto come $2\vec{b}_0 + 3\vec{b}_1$ utilizzando le basi $\vec{b}_0$ = [[1], [0]] e $\vec{b}_1$ = [[0], [1]]].

BASE

Se un vettore $\vec{v}$ in *d* dimensioni può essere scritto come somma di multipli di $\vec{b}_0$, $\vec{b}_1$, …, $\vec{b}_{d-1}$, diciamo che $\vec{b}_0$, $\vec{b}_1$, …, $\vec{b}_{d-1}$ sono una *base*. In due dimensioni, una

base molto comune è quella orizzontale e verticale.

Più in generale, se conosciamo l'output di una funzione f per ogni vettore in una base, possiamo calcolare f per qualsiasi input. La situazione è simile a come usiamo le tabelle di verità per descrivere un'operazione classica, elencando gli output di un'operazione per ogni possibile input.

Risoluzione dei problemi con linearità

Supponiamo che *f* sia una funzione lineare che rappresenta il modo in cui la nostra mappa viene allungata e distorta. Come possiamo trovare dove dobbiamo andare? Vogliamo calcolare il valore `f(np.array([[2], [3]]))` (un valore alquanto arbitrario) data la nostra base `f(np.array([[1], [0]]))` (orizzontale) e `f(np.array([[0], [1]]))` (verticale). Sappiamo anche, osservando le parti della legenda della mappa, che essa deforma la direzione orizzontale in `np.array([[1], [1]])` e la direzione verticale in `np.array([[1], [-1 ]])`.

I passi per calcolare `f(np.array([[2], [3]]))` sono i seguenti.

- Usiamo la nostra base, `np.array([[1], [0]])` e `np.array([[0], [1]])`, per scrivere che `np.array([[2], [3]])` è uguale a `2 * np.array([[1], [0]]) + 3 * np.array([[0], [1]])`.
- Usando questo nuovo modo per scrivere il nostro input per la funzione, vogliamo calcolare `f(2 * np.array([[1], [0]]) + 3 * np.array([[0], [1]]))`.
- Usiamo il fatto che `f` è lineare per scrivere `f(2 * np.array([[1], [0]]) + 3 * np.array([[0], [1]]))` come `2 * f(np.array([[1], [0]])) + 3 * f(np.array([[0], [1]]))`:

```
>>> import numpy as np
>>> horizontal = np.array([[1], [0]]) (1)
>>> vertical = np.array([[0], [1]])
>>> vec = 2 * horizontal + 3 * vertical (2)
>>> vec
array([[2],
       [3]])
>>> f_horizontal = np.array([[1], [1]]) (3)
>>> f_vertical = np.array([[1], [-1]])
>>> 2 * f_horizontal + 3 * f_vertical (4)
array([[ 5],
       [-1]])
```

(1) Definisce le variabili `horizontal` e `vertical` per rappresentare la base che useremo per rappresentare `[[2], [3]]`.

(2) Possiamo scrivere `[[2], [3]]` sommando multipli di `horizontal` e `vertical`.

(3) Definisce il modo in cui `f` agisce su `horizontal` e `vertical` introducendo le nuove variabili `f_horizontal` e `f_vertical` per rappresentare, rispettivamente, `f(horizontal)` e `f(vertical)`.

(4) Poiché `f` è lineare, possiamo definire come funziona per `[[2], [3]]` sostituendo a `horizontal` e `vertical` gli output `f_horizontal` e `f_vertical`.

ESERCIZIO C.3 CALCOLO DI FUNZIONI LINEARI

Supponete di avere una funzione lineare g tale che g([[1], [0]]) = [[2.3], [-3.1]] e g([[0], [1]]) = [[-5,2], [0,7]]. Calcolate g([[2], [-2]]).

Usando questa conoscenza, possiamo creare una tabella del modo in cui una funzione lineare trasforma ciascuno dei suoi input. Queste tabelle sono chiamate matrici e rappresentano descrizioni complete delle funzioni lineari. Conoscendo la matrice di una funzione lineare, possiamo calcolare tale funzione per qualsiasi vettore. Per esempio, la trasformazione dalla convenzione nord/est alla convenzione est/nord per le direzioni della mappa trasforma l'istruzione "vai di un'unità verso nord" da `[[1], [0]]` a `[[0], [1]]`. Allo stesso modo, l'istruzione "vai di un'unità verso est" è trasformata da `[[0], [1]]` a `[[1], [0]]`. Se raggruppiamo gli output di entrambi i set di istruzioni, otteniamo la seguente matrice:

```
>>> swap_north_east = np.array([[0, 1], [1, 0]])
>>> swap_north_east
array([[0, 1],
       [1, 0]])
```

SUGGERIMENTO

Anche questa è una matrice molto importante nel quantum computing! La reincontreremo un po' in tutto il libro.

Per applicare a un determinato vettore la funzione lineare rappresentata da una certa matrice, moltiplichiamo la matrice e il vettore, come illustrato nella Figura C.6.

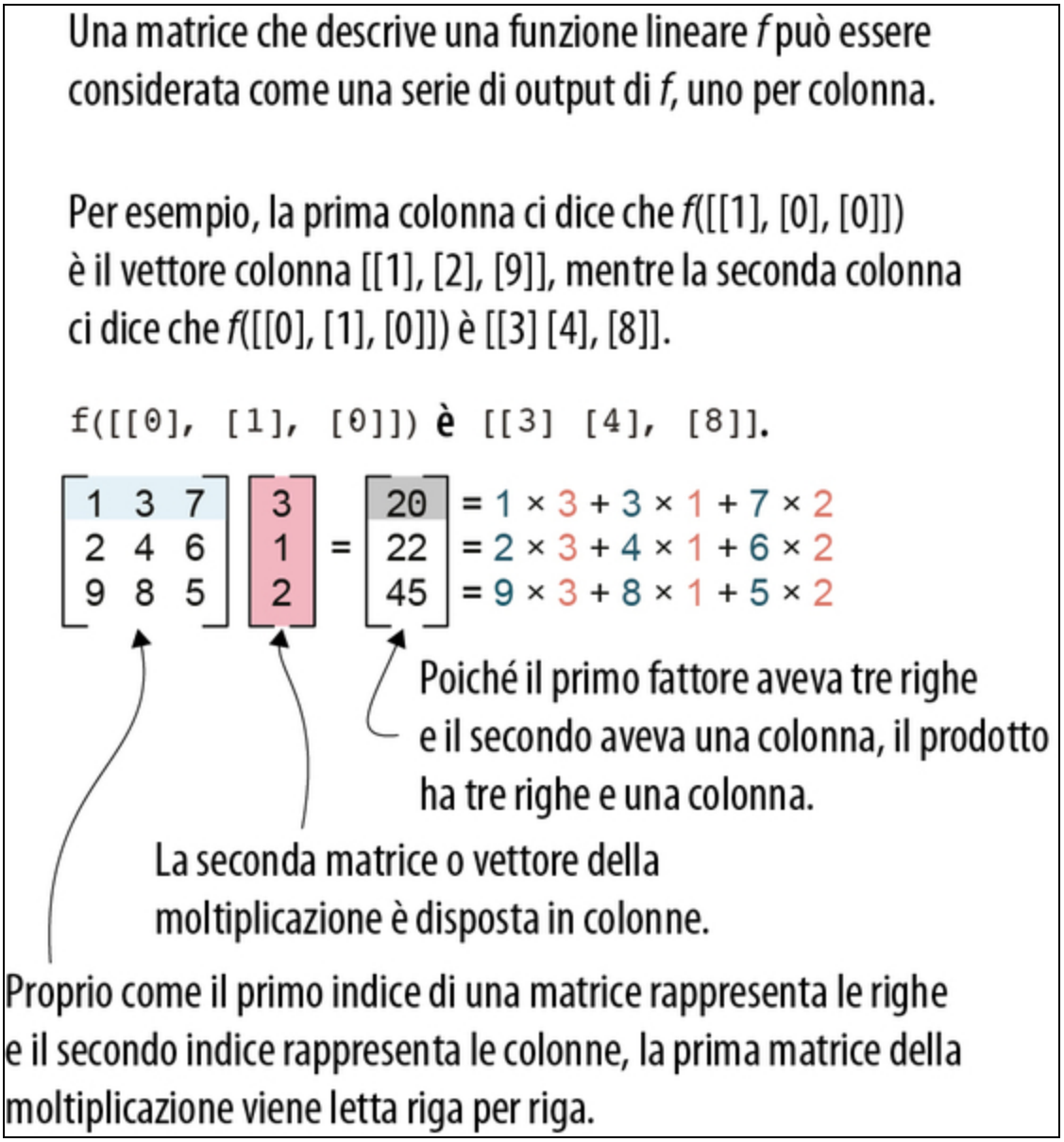


Figura C.6 Come moltiplicare una matrice per un vettore. In questo esempio, la matrice per f ci dice che f([[1], [0], [0]]) è uguale a [[1], [2], [9]].

ATTENZIONE

Sebbene l'ordine in cui sommiamo i vettori non sia importante, l'ordine in cui moltiplichiamo le matrici è molto importante. Se giriamo la mappa di 90° e poi la ribaltiamo allo specchio, otteniamo un'immagine molto diversa rispetto a una mappa prima ribaltata e poi girata. Sia girare la mappa, sia ribaltarla a specchio sono funzioni lineari, quindi possiamo scrivere una matrice per ciascuna di esse,

chiamiamole, rispettivamente, G e R. Se ribaltiamo un vettore $\vec{x}$, otteniamo $R\vec{x}$. Girando tale output otteniamo $GR\vec{x}$, un vettore molto diverso rispetto a quello che otterremmo se prima girassimo la mappa, $RG\vec{x}$.

La moltiplicazione di matrici formalizza il modo in cui abbiamo calcolato f dati i suoi output per un particolare insieme di input "impilando" gli output di f per vettori come [[1], [0], [0]] e [[0], [1], [0]], come illustrato nella Figura C.6. Sebbene le dimensioni effettive delle matrici e dei vettori possano cambiare, l'idea che una matrice possa descrivere una trasformazione lineare rimane la stessa. Per il resto di questa appendice, osserveremo le trasformazioni lineari su vettori di lunghezza 2. Possiamo immaginare ogni riga (l'indice più esterno, in NumPy) di una matrice come al modo in cui la funzione agisce su un particolare input.

Approfondimento: perché moltiplichiamo le funzioni?

Quando moltiplichiamo una matrice per un vettore (o anche una matrice per una matrice), stiamo facendo qualcosa che all'inizio ci può sembrare un po' strano. Dopotutto, le matrici sono un altro modo per rappresentare delle funzioni lineari, quindi che cosa significa moltiplicare una funzione per il suo input, per non parlare di moltiplicarla per un'altra funzione?

Per rispondere a questa domanda, è utile tornare per un momento all'algebra ordinaria, dove per qualsiasi variabile a, b, e c, abbiamo che $a(b + c) = ab + ac$. Questa proprietà, nota come *proprietà distributiva*, è fondamentale per le interazioni fra moltiplicazione e somma. In effetti, è così fondamentale che la proprietà distributiva è uno dei modi chiave in cui definiamo la moltiplicazione: nella teoria dei numeri e in altre parti più astratte della matematica, i ricercatori spesso lavorano con oggetti noti come *anelli*, dove tutto ciò che sappiamo veramente sulla moltiplicazione è che è distributiva sull'addizione. Sebbene applicato su un concetto astratto, lo studio degli anelli e di altri oggetti algebrici simili ha ampie applicazioni, specialmente nella crittografia e nella correzione degli errori.

La proprietà distributiva sembra molto simile alla proprietà di linearità: $f(x + y) = f(x) + f(y)$. Se pensiamo a f come membro di un anello, la proprietà distributiva è identica alla proprietà di linearità.

In altre parole, proprio come ai programmatori piace riutilizzare il codice, ai matematici piace riutilizzare i concetti. Pensare di moltiplicare le matrici fra loro ci consente di trattare le funzioni lineari in molti dei modi cui siamo ormai abituati dall'algebra.

Pertanto, se vogliamo conoscere l'i-esimo elemento di un vettore $\vec{x}$ che è stato ruotato di una matrice M, possiamo trovare l'output di M per ogni elemento di $\vec{x}$, sommare i vettori risultanti e prendere l'i-esimo elemento. In NumPy, la moltiplicazione di matrici è rappresentata dall'operatore `@`.

NOTA

L'esempio di codice seguente funziona solo in Python 3.5 o versione successiva.

Listato C.2 Moltiplicazione di matrici con l'operatore @.

```
>>> M = np.array([
...      [1, 1],
...      [1, -1]
... ], dtype=complex)
>>> M @ np.array([[2], [3]], dtype=complex)
array([[ 5.+0.j],
       [-1.+0.j]])
```

ESERCIZIO C.4 MOLTIPLICAZIONE DI MATRICI

Sia *X* la matrice [[0, 1], [1, 0]] e sia $\vec{y}$ il vettore [[2], [3]]. Utilizzando NumPy, calcolate $X\vec{y}$ e *XX*.

Perché NumPy?

Avremmo potuto scrivere tutta la moltiplicazione di matrici precedente a mano, ma ci sono alcuni motivi per cui è molto meglio lavorare con NumPy. La maggior parte del nucleo centrale di NumPy utilizza l'indicizzazione a tempo costante ed è implementata in codice nativo, per sfruttare le istruzioni del microprocessore, così da accelerare le istruzioni di algebra lineare. Pertanto, NumPy è spesso molto, molto più veloce della manipolazione manuale delle liste. Nel Listato C.3, vediamo un esempio in cui NumPy può accelerare di un fattore 10× la moltiplicazione anche di matrici molto piccole. Quando opereremo su matrici più grandi, nei capitoli dal 4 in poi, l'uso di NumPy rispetto alle operazioni manuali si dimostrerà ancora più vantaggioso.

Listato C.3 Valutazione prestazionale di NumPy nella moltiplicazione di matrici.

```
$ ipython (1)
In [1]: def matmul(A, B):
   ...:     n_rows_A = len(A) (2)
   ...:     n_cols_A = len(A[0])
   ...:     n_rows_B = len(B)
   ...:     n_cols_B = len(B[0])
   ...:     assert n_cols_A == n_rows_B (3)
   ...:     return [ (4)
   ...:         [
   ...:             sum( (5)
   ...:                 A[idx_row][idx_inner] * B[idx_inner][idx_col]
   ...:                 for idx_inner in range(n_cols_A)
   ...:             )
   ...:             for idx_col in range(n_cols_B)
   ...:         ]
   ...:         for idx_row in range(n_rows_A)
   ...:     ]
   ...:
In [2]: import numpy as np (6)
In [3]: X = np.array([[0+0j, 1+0j], [1+0j, 0+0j]]) (7)
In [4]: Z = np.array([[1+0j, 0+0j], [0+0j, -1+0j]])
In [5]: matmul(X, Z)
Out[5]: [[0j, (-1+0j)], [(1+0j), 0j]]
In [6]: X @ Z (8)
Out[6]:
array([[ 0.+0.j, -1.+0.j],
       [ 1.+0.j, 0.+0.j]])
In [7]: %timeit matmul(X, Z) (9)
10.3 µs ± 176 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
In [8]: %timeit X @ Z
926 ns ± 4.42 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

(1) Questa volta, utilizziamo l'interprete IPython per Python, poiché fornisce alcuni strumenti aggiuntivi utili in questo esempio. Vedi l'Appendice A per le istruzioni su come installare IPython.

(2) Trova le dimensioni di ciascuna matrice che dobbiamo moltiplicare. Se rappresentiamo le matrici come liste di liste, ogni elemento della lista esterna è una riga. Cioè, una matrice n × m ha n righe e m colonne, quando viene scritta in questo modo.

(3) Le dimensioni interne di entrambe le matrici devono concordare affinché la moltiplicazione di matrici abbia un senso. Pensando a ciascuna matrice come rappresentazione di una funzione lineare, il primo indice (il numero di righe) ci dice quanto è grande ogni output, mentre il secondo indice (il numero di colonne) ci dice quanto è grande ogni input. Così

abbiamo bisogno che gli output della prima funzione da applicare (quella a destra) siano della stessa dimensione degli input della seconda funzione. Questa riga verifica tale condizione.

(4) Per calcolare effettivamente il prodotto fra le matrici `A` e `B`, dobbiamo calcolare ogni elemento e poi formare una lista di liste.

(5) Possiamo trovare ogni elemento sommando l'output di `B` passato come input ad `A`, un po' come abbiamo rappresentato il prodotto di una matrice con un vettore nella Figura C.6.

(6) Per confronto, possiamo importare NumPy, che ci fornisce un'implementazione della moltiplicazione di matrici che utilizza le istruzioni interne del microprocessore per accelerare il calcolo.

(7) Inizializza due matrici come array NumPy, da usare come casi di test. Reincontreremo queste due particolari matrici un po' in tutto il libro.

(8) La moltiplicazione di matrici in NumPy è rappresentata dall'operatore `@` (Python versioni 3.5 e successive).

(9) Il "comando magico" `%timeit` dice a IPython di ripetere più volte un piccolo frammento di codice Python e di riportare il tempo medio impiegato.

Il prodotto interno

C'è un'ultima cosa di cui dobbiamo occuparci per trovare la famosa festa. In precedenza, abbiamo ignorato il fatto che magari esiste una strada che ci permetterebbe di andare più direttamente nella direzione desiderata, ma questa è davvero una pessima idea quando girovaghiamo per una città sconosciuta. Per orientarci, abbiamo bisogno di un modo per valutare fino a che punto dovremmo camminare lungo una determinata strada per arrivare dove stiamo andando. Fortunatamente, l'algebra lineare ci fornisce uno strumento per fare proprio questo: il prodotto

interno (Figura C.7). I prodotti interni sono un modo per proiettare un vettore $\vec{v}$ su un altro vettore $\vec{w}$: ci dice quanta "ombra" $\vec{v}$ proietta su $\vec{w}$.

Possiamo calcolare il prodotto interno di due vettori moltiplicando i rispettivi elementi e sommando il risultato. Notate che questa "ricetta" (moltiplicazione e somma) è la stessa che seguiamo nella moltiplicazione di matrici. Moltiplicare una matrice che ha una singola riga per una matrice che ha una singola colonna fa esattamente ciò di cui abbiamo bisogno. Quindi, per trovare la proiezione di $\vec{v}$ su $\vec{w}$, dobbiamo trasformare $\vec{v}$ in un vettore riga prendendo la sua trasposta, scritta $\vec{v}^T$.

ESEMPIO

La trasposta di $\vec{w} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$ è $\vec{w}^T = (2\ 3)$

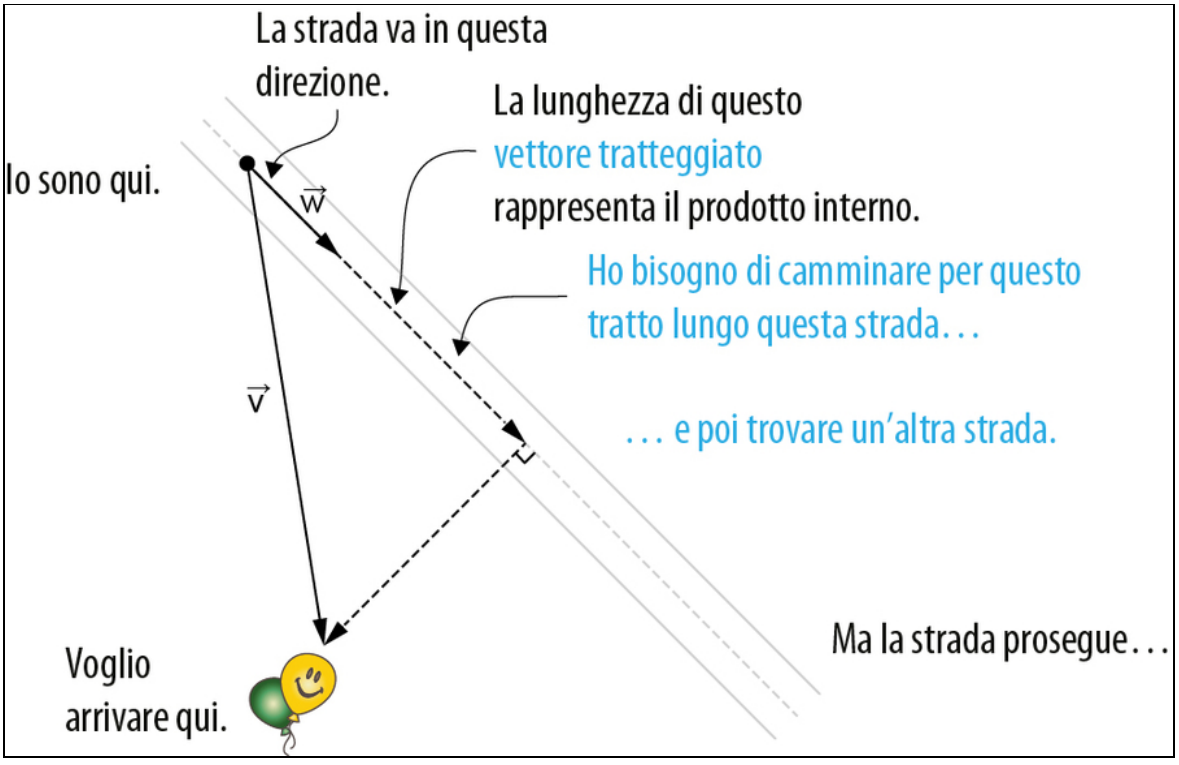


Figura C.7 Come trovare la festa con prodotti interni.

NOTA

Nel Capitolo 3, vedremo che abbiamo anche bisogno di prendere il coniugato complesso di ogni elemento, ma per il momento non ce ne occupiamo.

In particolare, il prodotto fra le matrici $\vec{v}^T$ (la trasposta di $\vec{v}$) e $\vec{w}$ ci dà una matrice 1×1 contenente il prodotto interno che vogliamo. Supponiamo di dover andare due isolati a sud e tre isolati a est, ma possiamo solo seguire una strada che punta più a sud-sud-est. Dato che dobbiamo ancora procedere verso sud, questa strada ci aiuta ad arrivare là dove dobbiamo andare. Ma quanto dovremmo camminare prima che questa strada smetta di essere utile?

Listato C.4 Calcolo di prodotti scalari di vettori con NumPy.

```
>>> import numpy as np
>>> v = np.array([[-2], [-3]]) (1)
>>> south_east = np.array([[1], [-1]]) (2)
>>> np.linalg.norm(south_east) (3)
1.4142135623730951 (4)
>>> w = np.array([[1], [-1]]) / np.sqrt(2) (5)
>>> np.linalg.norm(w) (6)
0.9999999999999999
>>> v.transpose() (7)
array([[-2, -3]])
>>> v.transpose() @ w (8)
array([[ 0.70710678]]) (9)
```

(1) In questo caso, $\vec{v}$ è il vettore che descrive dove dobbiamo andare: due isolati a nord e tre isolati a est.

(2) Se la strada punta a sud-est, allora va di un isolato a sud per ogni isolato a est.

(3) Possiamo trovare la lunghezza di un vettore usando il teorema di Pitagora, prendendo la somma dei valori assoluti di ogni elemento e poi applicando la radice quadrata. In NumPy, questo viene fatto con la funzione `np.linalg.norm` (la lunghezza di un vettore è talvolta chiamata anche norma).

(4) La lunghezza di [[1], [–1]] è quindi $\sqrt{(+1^2 + (-1^2} = \sqrt{2} \approx 1{,}4142$.

(5) Quando definiamo $\vec{w}$ come la direzione sud-est, dobbiamo dividere per $\sqrt{2}$.

(6) Controllando, vediamo che la lunghezza di $\vec{w}$ è ora di circa 1.
(7) La trasposizione trasforma $\vec{v}$ = [[-2], [–3]] nella "riga" [[-2, -3]].
(8) Quindi possiamo moltiplicare la trasposta di $\vec{v}$ per $\vec{w}$ nello stesso modo in cui abbiamo moltiplicato le matrici con i vettori.
(9) Abbiamo bisogno di camminare $1 / \sqrt{2} \approx 0{,}707$ isolati lungo questa strada prima che smetta di aiutarci ad avvicinarci alla festa.

ESERCIZIO C.5 NORMALIZZAZIONE VETTORIALE

Dato un vettore [[2], [3]], trovate un vettore che punti nella stessa direzione, ma con lunghezza 1.

Suggerimento: potete farlo utilizzando il prodotto interno o la funzione `np.linalg.norm`.

Finalmente siamo arrivati alla festa (appena in ritardo) e siamo pronti per provare il nuovo campanello!

Radici quadrate e lunghezze

La radice quadrata di un numero x è un numero $y = \sqrt{x}$ tale che otteniamo di nuovo x elevando al quadrato y, $y^2 = x$. Usiamo molto le radici quadrate in tutto il libro, poiché sono essenziali per trovare la lunghezza dei vettori. Per esempio, nella computer graphics, trovare rapidamente le lunghezze dei vettori è essenziale per far funzionare i giochi (vedi https://en.wikipedia.org/wiki/Fast_inverse_square_root per qualche storia divertente sull'uso delle radici quadrate nei giochi).

Indipendentemente dal fatto che i vettori descrivano come arrivare a una festa o le informazioni rappresentate da un bit quantistico rappresenta, per valutare le lunghezze servono le radici quadrate.

L’algoritmo di Deutsch-Jozsa tramite esempi

In questa appendice, compiamo un’immersione profonda nell’algoritmo di Deutsch-Jozsa per mostrare come funziona e come possiamo usare le competenze e gli strumenti che abbiamo sviluppato nel Capitolo 8 per verificare la nostra comprensione. Abbiamo implementato l’algoritmo di Deutsch-Jozsa nel Capitolo 7, facendo un uso massiccio di QuTiP per controllare i passaggi matematici.

Usare le nostre conoscenze per dimostrare le cose

Nei Capitoli 2 e 5 abbiamo imparato a usare NumPy e QuTiP per simulare come vengono trasformati gli stati dei qubit quando inviamo istruzioni a un computer quantistico. Possiamo usare efficacemente questi package per svolgere calcoli e scoprire che cosa accade ai nostri stati quantistici. Questo è un approccio alla “fai fare i calcoli al computer”, rappresentato nella Figura D.1.

Quando programmiamo algoritmi più grandi in Q#, possiamo usare sia l’approccio “fai fare i calcoli al computer” sia un po’ dell’approccio “premi i tasti” per aiutarci a prevedere il risultato di una determinata operazione. I tre approcci mostrati nella Figura D.1, usati insieme, sono potenti strumenti di risoluzione dei problemi durante

l'apprendimento della programmazione quantistica. Se rimaniamo bloccati usando un approccio, possiamo sempre provarne un altro, per vedere se aiuta.

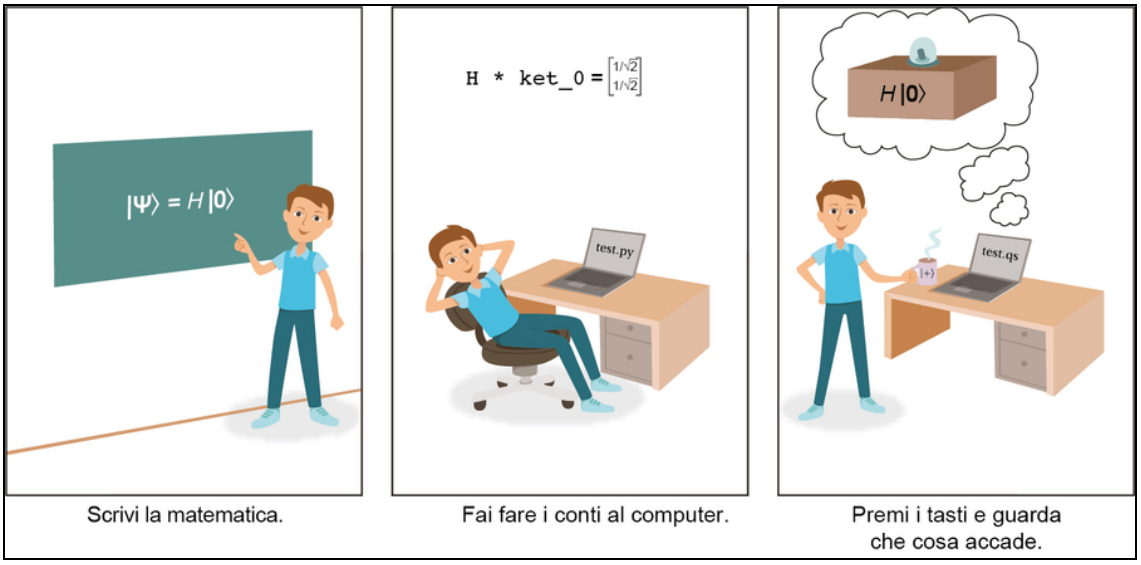


Figura D.1 Tre differenti approcci per l'apprendimento del funzionamento di un programma o algoritmo quantistico.

Proviamo ad applicare questo approccio combinato all'algoritmo di Deutsch-Jozsa del Capitolo 8. Il seguente listato mostra i quattro passi dell'algoritmo.

Listato D.1 I quattro passi dell'algoritmo di Deutsch-Jozsa.

```
H(control); (1)
X(target); (1)
H(target); (1)
oracle(control, target); (2)
H(target); (3)
X(target); (3)
set result = MResetX(control); (4)
```

(1) Prepara lo stato di input |+-⟩.

(2) Applica l'oracolo.

(3) Annulla la preparazione sul qubit `target` dello stato di input |+-⟩.

(4) Infine, misura in base X.

La chiave per capire come funziona l'algoritmo di Deutsch-Jozsa è capire il passaggio in cui richiamiamo l'oracolo, `oracle(control, target)`. Ma prima di poterlo fare, dobbiamo capire il Passo 1, dove prepariamo il nostro input per l'oracolo.

Passo 1: preparazione dello stato di input per Deutsch-Jozsa

Usiamo Python per cercare di capire che cosa succede quando prepariamo il nostro stato |+-⟩. Le operazioni che utilizziamo per preparare lo stato di input in Q# sono le seguenti:

```
H(control);
X(target);

H(target);
```

Ogni operazione applicata è un gate mono-qubit, quindi possiamo considerare che cosa accade a ciascun qubit indipendentemente. Diamo un'occhiata a quello che accade al qubit `control` dopo l'operazione di Hadamard. Usiamo QuTiP per modellare la preparazione degli stati del qubit control:

```
>>> import qutip as qt
>>> from qutip.qip.operations import hadamard_transform
>>> H = hadamard_transform() (1)
>>> H
Quantum object: dims = [[2], [2]], shape = (2, 2), type = oper, isherm =
True
Qobj data =
[[ 0.70710678 0.70710678] (2)
 [ 0.70710678 -0.70710678]]
>>> control_state = H * qt.basis(2, 0) (3)
>>> control_state
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket (4)
Qobj data =
[[0.70710678]

 [0.70710678]]
```

(1) Mentre `H` è un'istruzione Q#, `hadamard_transform` in QuTiP ci fornisce una matrice unitaria che possiamo usare per simulare come

l'istruzione `H` trasforma gli stati.

(2) $1 / \sqrt{2} \approx 0.707$, quindi questo output ci dice che H è uguale a:

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} / \sqrt{2}$$

.

(3) In QuTiP, possiamo ottenere il vettore per lo stato $|0\rangle$ richiamando `basis(2, 0)`. Il `2` dice a QuTiP che vogliamo un qubit (dimensione necessaria di $|0\rangle$), mentre lo `0` dice che vogliamo che lo stato abbia il valore $|0\rangle$. Poiché $|+\rangle = H\,|0\rangle$, questo imposta `control_state` a $|+\rangle$.

(4) Usando che $1 / \sqrt{2} \approx 0{,}707$, leggiamo questa riga come se dicesse $|+\rangle = (|0\rangle + |1\rangle) / \sqrt{2}$.

È tutto abbastanza semplice: il qubit `control` ora è nello stato $|+\rangle$. Ora diamo un'occhiata alla preparazione del qubit `target` nel prossimo frammento di codice.

```
>>> target_state = H * (qt.sigmax() * qt.basis(2, 0)) (1)
>>> target_state
Quantum object: dims = [[2], [1]], shape = (2, 1), type = ket (2)
Qobj data =
[[ 0.70710678]

 [-0.70710678]]
```

(1) Ripete la stessa operazione `H` di prima, ma questa volta su $X|0\rangle = |1\rangle$.

(2) QuTiP ci dice che $|-\rangle = (|0\rangle - |1\rangle) / \sqrt{2}$: come $|+\rangle$, ma con il segno di $|1\rangle$ capovolto.

Ora che abbiamo visto come preparare ogni qubit, facciamo in modo che QuTiP ci aiuti a scrivere lo stato del nostro registro di input.

```
>>> register_state = qt.tensor(control_state, target_state) (1)
>>> register_state
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket (2)
Qobj data =
[[ 0.5]
 [-0.5]
 [ 0.5]

 [-0.5]]
```

(1) Come nel Capitolo 4, combiniamo gli stati di più qubit per ottenere lo stato di un intero registro di qubit usando la funzione `tensor`.

(2) QuTiP ci dice che $|+\rangle \otimes |-\rangle = |+-\rangle = (|00\rangle - |01\rangle + |10\rangle - |11\rangle) / 2$. Cioè, abbiamo una sovrapposizione uguale su tutti e quattro i possibili stati computazionali di base, con un segno meno davanti a quegli stati in cui il qubit `target` è nello stato $|1\rangle$.

> **NOTA**
>
> Come si vede nel Capitolo 4, quando si scrive lo stato di un sistema multi-qubit, i prodotti tensoriali possono diventare un po' complessi. Pertanto, spesso scriviamo gli stati multi-qubit come $|0\rangle \otimes |1\rangle$ concatenando le loro etichette all'interno di un singolo ket, come in $|01\rangle$. Allo stesso modo, $|+-\rangle$ è uguale a $|+\rangle \otimes |-\rangle$.

Passo 2: applicazione dell'oracolo

Dopo aver preparato il nostro input, torniamo al nucleo dell'algoritmo di Deutsch-Jozsa, dove richiamiamo il nostro oracolo:

```
oracle(control, target);
```

Proprio come possiamo comprendere operazioni come `H(control)` scrivendo lo stato del qubit `control` e applicando l'operatore unitario `H` a tale stato, possiamo comprendere che cosa fa l'oracolo Uf analizzando la sua azione sullo stato che gli passiamo.

Ricordiamo la configurazione di gioco del Capitolo 8, dove Viviana e Merlino stanno giocando a Kingmaker. Il nostro oracolo quantistico opera su due qubit, il che solleva la questione di come dovremmo interpretare ciascuno di essi. Nel caso classico, l'interpretazione dei bit classici di input e output da f era chiara: Viviana ha posto una domanda di un bit e ha ottenuto una risposta di un bit.

Per capire che cosa fa ogni qubit, ricordiamo che quando usiamo una funzione classica reversibile, abbiamo anche bisogno di due input: il primo si comporta come la domanda che poniamo nel caso

irreversibile; il secondo input ci dà un posto dove collocare la risposta (vedi la Figura D.2, per un promemoria).

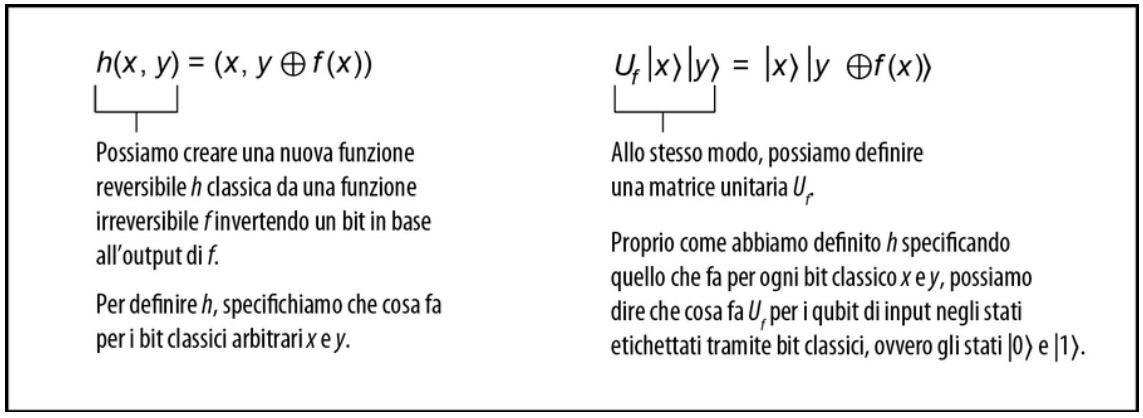


Figura D.2 Costruzione di funzioni classiche reversibili e matrici unitarie a partire da funzioni classiche irreversibili.

Possiamo considerare l'oracolo più o meno allo stesso modo: il primo qubit (`control`, nel frammento di codice precedente) rappresenta la nostra domanda, mentre il secondo qubit (`target`) ci dà un posto in cui Merlino può inserire la sua risposta. Questa interpretazione ha senso quando `control` inizia nello stato $|0\rangle$ o $|1\rangle$, ma come possiamo interpretare questo caso in cui passiamo all'oracolo un qubit nello stato $|+-\rangle$? Il nostro qubit `control` inizia nello stato $|+\rangle$, ma f(+) non ha alcun senso. Poiché f è una funzione classica, il suo input deve essere 0 o 1: non possiamo passare + alla funzione classica f. Può sembrare che ci troviamo in un vicolo cieco, ma fortunatamente c'è un modo per procedere.

La meccanica quantistica è lineare, il che significa che possiamo sempre capire che cosa fa un'operazione quantistica scomponendola nella sua azione su un insieme rappresentativo di stati.

SUGGERIMENTO

Come abbiamo visto nel Capitolo 2, un insieme di stati che possono essere usati in questo modo è chiamato *base*.

Per capire ciò che fa il nostro oracolo quando il qubit `control` è nello stato $|+\rangle$, possiamo utilizzare il fatto che $|+\rangle = (|0\rangle + |1\rangle) / \sqrt{2}$ a suddividere l'azione dell'oracolo in quello che fa a $|0\rangle$ più in quello che fa a $|1\rangle$ e poi sommare le due azioni (assicurandosi di dividere per $\sqrt{2}$ alla fine). Questo ci aiuta, perché invece di cercare di capire che cosa significa "f(+)", possiamo ridurre l'azione di Uf a casi che sappiamo calcolare, come f(0) e f(1).

Stati computazionali di base

Espandere l'azione di un'operazione quantistica in termini di come essa agisce su $|0\rangle$ e $|1\rangle$ è molto comune nella programmazione quantistica. Data la sua utilità, diamo un nome speciale a questi due stati di input e chiamiamo $|0\rangle$ e $|1\rangle$ la *base computazionale* per distinguerla da altre basi che potremmo usare, come $|+\rangle$ e $|-\rangle$.

L'uso della linearità per comprendere le operazioni quantistiche non si limita a un singolo qubit, come vedremo nel resto dell'appendice. Per due qubit, per esempio, la base computazionale è costituita dagli stati $|00\rangle$, $|01\rangle$, $|10\rangle$ e $|11\rangle$.

Se abbiamo ancora più qubit (diciamo cinque), possiamo scrivere allo stesso modo stati come $|1\rangle \otimes |0\rangle \otimes |0\rangle \otimes |1\rangle \otimes |0\rangle$ come stringhe, ottenendo $|10010\rangle$. Possiamo quindi scrivere la base computazionale per cinque qubit come $\{|00000\rangle, |00001\rangle, |00010\rangle, \ldots, |11110\rangle, |11111\rangle\}$.

Più in generale, se abbiamo n qubit, la base computazionale è costituita da tutte le stringhe di n bit classici, ciascuna come etichetta di un ket. In altre parole, la base computazionale per un sistema multi-qubit è costituita da tutti i prodotti tensoriali di $|0\rangle$ e $|1\rangle$: cioè, tutti gli stati etichettati da una stringa di bit classici.

Con questo approccio che ci consente di capire come funziona l'oracolo, vediamo alcuni esempi degli oracoli che abbiamo implementato nel Capitolo 8.

Esempio 1: l'oracolo "id"

Supponiamo che ci venga dato un oracolo che implementa la strategia in cui Merlino sceglie Artù come re. Ricordiamo che la classica funzione a un bit che rappresentaenta questa strategia è `id`. Dalla Tabella D.1,

sappiamo che questo significa che Uf è implementata dall'istruzione `CNOT`, quindi cerchiamo di vedere quello che fa a `register_state`.

Tabella D.1 Rappresentazione delle funzioni a un bit come oracoli a due qubit.

Nome funzione	Funzione	Output dell'oracolo	Operazione Q#
`id`	f(x) = x	$\|x\rangle\|y \oplus x\rangle$	`CNOT(control, target)`

SUGGERIMENTO

Ricordate che l'istruzione Controlled-NOT inverte il suo secondo qubit se il primo qubit è $|1\rangle$.

Listato D.2 Come l'oracolo id trasforma il suo stato di input.

```
>>> cnot = qt.cnot(2, 0, 1) (1)
>>> cnot
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper,
isherm = True
Qobj data = (2)
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 1.] (3)
 [0. 0. 1. 0.]]
>>> register_state = cnot * register_state
>>> register_state (4)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.5]
 [-0.5]
 [-0.5]

 [ 0.5]]
```

(1) Chiede a QuTiP una matrice che ci consenta di simulare l'istruzione `CNOT` utilizzando la funzione `cnot`. Qui, il `2` indica che vogliamo simulare CNOT su un registro di due qubit, lo `0` indica che lo "zeresimo" qubit è il nostro `control` e l'`1` indica che il primo qubit è il nostro `target`.

(2) Ricordate che gli operatori unitari sono per il quantum computing ciò che le tabelle di verità sono per la logica classica. Ogni riga di questa tabella ci dice che cosa accade a uno stato computazionale di base.

(3) Per esempio, la riga all'indice 2 (con indicizzazione a base 0) può essere scritta come 10 in binario. Così, questa riga è il vettore che

otterremmo se il nostro input fosse |10⟩, e ci dice che l'istruzione `CNOT` lascia i nostri qubit a |11⟩ (3 in decimale, perché c'è un 1 nella terza colonna).

(4) QuTiP ci dice che il registro con i nostri qubit `control` e `target` è ora nello stato (|00⟩ - |01⟩ - |10⟩ + |11⟩) / 2.

Ora che abbiamo elaborato l'azione dell'oracolo `id`, diamo un'occhiata a quello che fa l'oracolo `not` al nostro stato di input.

Esempio 2: l'oracolo "not"

Ripetiamo l'analisi utilizzando l'oracolo `not`, l'altra funzione bilanciata. L'oracolo che rappresenta Merlino che sceglie Mordred è implementato con una serie di operazioni `X` e `CNOT` come nella Tabella D.2.

Tabella D.2 La funzione a un bit not come un oracolo a due qubit.

Nome funzione	Funzione	Output dell'oracolo	Operazione Q#
`not`	f(x) = ¬x	|x⟩|y ⊕ ¬x⟩	`X(control); CNOT(control, target); X(control);`

Passiamo al codice Python per vedere come scomporre il funzionamento dell'oracolo `not`.

Listato D.3 Utilizzo di QuTiP, ora con l'oracolo not.

```
>>> control_state = H * qt.basis(2, 0) (1)
>>> target_state = H * qt.basis(2, 1)
>>> register_state = qt.tensor(control_state, target_state)
>>> I = qt.qeye(2) (2)
>>> X = qt.sigmax()
>>> oracle = qt.tensor(X, I) * qt.cnot(2, 0, 1) *
... qt.tensor(X, I) (3)
>>> oracle
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper,
isherm = True
Qobj data = (4)
[[0. 1. 0. 0.] (5)
 [1. 0. 0. 0.]
 [0. 0. 1. 0.] (6)
 [0. 0. 0. 1.]]
>>> register_state = oracle * register_state
```

```
>>> register_state
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket (7)
Qobj data =
[[-0.5]
 [ 0.5]
 [ 0.5]
 [-0.5]]
```

(1) Prepara i qubit `control` e `target` nello stato |+-⟩, esattamente come prima.

(2) Come nel Capitolo 5, è utile definire l'abbreviazione delle variabili `I` e `X`, rispettivamente per la matrice identità (`qt.qeye`) e la matrice che rappresenta l'operazione X.

(3) Questa volta il nostro oracolo è `not`, che implementiamo con la sequenza di istruzioni `X (control); CNOT(control, target); X (control);` come dalla Tabella D.2.

(4) L'operatore unitario per l'operazione dell'oracolo ha un aspetto leggermente diverso questa volta: inverte il qubit `target` quando il qubit `control` è |0⟩.

(5) Per esempio, la riga 0 (00 in binario) ci dice che |00⟩ viene trasformato in |01⟩.

(6) Allo stesso modo, la riga 2 (10 in binario) ci dice che |10⟩ viene trasformato in |10⟩; l'oracolo lascia immutato questo input.

(7) Lo stato dopo aver applicato l'oracolo è (-|00⟩ + |01⟩ + |10⟩ - |11⟩) / 2 = (-1) |+-⟩, esattamente lo stesso di prima, a parte una fase globale di - 1.

Osservando questi due esempi, abbiamo ottenuto lo stesso stato di output, tranne per il fatto che i segni sono invertiti. Ciò significa che se moltiplichiamo uno dei vettori di stato per -1, sarebbero entrambi uguali. La moltiplicazione di un intero vettore per una costante è chiamata aggiunta di una fase globale. Poiché le fasi globali non possono essere osservate tramite misurazioni, abbiamo ottenuto esattamente le stesse informazioni applicando gli oracoli `id` e `not`. Non abbiamo imparato nulla

sul fatto che abbiamo applicato `id` o `not`; se potessimo confrontare i vettori, ora sapremmo solo che abbiamo applicato un oracolo bilanciato.

Per confronto, cerchiamo di vedere l'aspetto del registro dopo aver applicato un oracolo che rappresenta una funzione costante.

Esempio 3: l'oracolo "zero"

Ancora una volta, usiamo Python per scoprire come funziona un oracolo che rappresenta la funzione costante `zero`. Vogliamo usare l'oracolo `zero` per mostrare che cosa accade di diverso quando applichiamo un oracolo che rappresenta una funzione costante. Questo oracolo è particolarmente facile da applicare, poiché consiste nel non applicare alcuna istruzione. Potete vedere tutti i modi per rappresentarlo nella Tabella D.3.

Tabella D.3 La funzione a un bit zero come oracoli a due qubit.

Nome funzione	Funzione	Output dell'oracolo	Operazione Q#
`zero`	f(x) = 0	$\|x\rangle\|y \oplus 0\rangle = \|x\rangle\|y\rangle$	(nulla)

Nel Listato D.4, possiamo vedere che il non fare nulla sul qubit `control` e sul qubit `target` può essere simulato senza fare nulla sull'intero registro. Quindi, l'oracolo che creiamo è la matrice identità a due qubit $\otimes$ per l'oracolo `zero`.

Listato D.4 Calcolo della trasformazione dell'oracolo zero.

```
>>> control_state = H * qt.basis(2, 0)
>>> target_state = H * qt.basis(2, 1)
>>> register_state = qt.tensor(control_state, target_state)
>>> X = qt.sigmax()
>>> oracle = qt.tensor(I, I)
>>> oracle
Quantum object: dims = [[2, 2], [2, 2]], shape = (4, 4), type = oper,
isherm = True
Qobj data =
[[1. 0. 0. 0.] (1)
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

```
>>> register_state = oracle * register_state
>>> register_state (2)
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket
Qobj data =
[[ 0.5]
 [-0.5]
 [ 0.5]
 [-0.5]]
```

(1) Il fatto di non fare nulla sul qubit `control` e sul qubit `target` può essere simulato senza fare nulla sull'intero registro.
(2) Lo stato di output differisce di più di una fase globale rispetto all'oracolo `id`. Non esiste uno scalare che possiamo moltiplicare per una fase globale per trasformare l'output di `id` nell'output di `zero`.

Qui vediamo la nostra prima differenza dal listato precedente: i segni meno si trovano in punti diversi sul nostro vettore di stato. In precedenza, abbiamo utilizzato l'oracolo `id` e abbiamo ottenuto che lo stato dell'output era (|00⟩ - |01⟩ - |10⟩ + |11⟩) / 2. Usando l'oracolo zero, lo stato dell'output è (|00⟩ - |01⟩ + |10⟩ - |11⟩) / 2. Non esiste un numero per il quale possiamo moltiplicare l'intero vettore per cambiarlo in `[[0.5], [- 0.5], [- 0.5], [0.5]]` o `[[-0,5], [0,5], [0,5], [-0,5]]`. Per vedere come questa differenza porta a dirci con certezza se abbiamo un oracolo equilibrato o costante, continuiamo alla fase successiva dell'algoritmo di Deutsch-Jozsa.

ESERCIZIO D.1 PROVATE L'ORACOLO ONE

Scoprite se potete usare i trucchi Python che abbiamo usato in precedenza per capire come cambiano gli stati dei qubit `target` e `control` quando viene applicato l'oracolo `one`.

SOLUZIONI DEGLI ESERCIZI

Tutte le soluzioni degli esercizi di questo libro si trovano nel suo archivio di codice: https://github.com/crazy4pi314/learn-qc-with-python-and-qsharp. Basta andare nella cartella dell'appendice in cui siamo e aprire il notebook Jupyter con il nome che menziona le soluzioni degli esercizi.

Passi 3 e 4: annullamento della preparazione sul qubit target e misurazione

A questo punto, è molto più facile dare un senso all'output se annulliamo i passaggi che abbiamo usato per preparare |+-⟩ in modo che tutto ritorni nella base computazionale (|00⟩ … |11⟩). Per ricapitolare, la Tabella D.4 mostra i vettori di stato di tutti e quattro gli oracoli (tre dei quali abbiamo elaborato in precedenza).

Ora vogliamo annullare i nostri passaggi di preparazione sul qubit `target`.

Tabella D.4 Lo stato del registro dopo l'applicazione degli oracoli.

Nome della funzione	Stato del registro dopo l'applicazione dell'oracolo
`zero`	[[0,5], [-0,5], [0,5], [-0,5]]
`one`	[[-0,5], [0,5], [-0,5], [0,5]]
`id`	[[0,5], [-0,5], [-0,5], [0,5]]
`not`	[[-0,5], [0,5], [0,5], [-0,5]]

Perché de-prepariamo il qubit target?
Nel Capitolo 7, abbiamo visto che abbiamo bisogno di ripristinare i nostri qubit allo stato |0⟩ prima di restituirli alla macchina target. A questo punto, il nostro qubit `target` è sempre nello stato |-⟩, indipendentemente dall'oracolo che usiamo. Ciò significa che dopo aver applicato l'oracolo, sappiamo esattamente come rimetterlo a |0⟩. Come nel Capitolo 7, questo ci aiuta a evitare una misurazione aggiuntiva, che può essere costosa su alcuni dispositivi quantistici.
Notate che possiamo tranquillamente riportare il qubit `target` a |-⟩ senza influenzare i risultati della misurazione del qubit `control`, poiché la chiamata all'oracolo è l'unica operazione a due qubit nell'algoritmo di Deutsch-Jozsa. Come abbiamo visto nel Capitolo 5, eseguire operazioni mono-qubit su un qubit non può influenzare i risultati di un altro qubit; altrimenti, saremmo in grado di inviare informazioni più velocemente della luce.

Proviamo a de-preparare il registro dall'oracolo che rappresenta la funzione `id`.

Listato D.5 L'output dell'oracolo id nella base computazionale.

```
>>> I = qt.qeye(2) (1)
>>> register_state_id = qt.cnot(2,0,1) * (2)
... (qt.tensor(H * qt.basis(2, 0), H * (qt.sigmax() * qt.basis(2, 0))))
...
>>> register_state_id = qt.tensor(I, H) * register_state_id (3)
>>> register_state_id
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket (4)
Qobj data =
[[ 0. ]
 [ 0.70710678]
 [ 0. ]
 [-0.70710678]]
>>> register_state_id = qt.tensor(I, qt.sigmax()) * register_state_id (5)
>>> register_state_id
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket (6)
Qobj data =
[[ 0.70710678]
 [ 0. ]
 [-0.70710678]
 [ 0. ]]
>>> qt.tensor(H * qt.basis(2, 1), qt.basis(2, 0)) == register_state_id (7)
True
>>> register_state_id = qt.tensor(H, I) * register_state_id (8)
>>> register_state_id
Quantum object: dims = [[2, 2], [1, 1]], shape = (4, 1), type = ket (9)
Qobj data =
[[0.]
 [0.]
 [1.]
 [0.]]
```

(1) È utile definire una scorciatoia per la matrice identità $\mathbb{1}$, che usiamo per rappresentare ciò che accade a un qubit quando non gli applichiamo alcuna istruzione.

(2) Riproduce il registro dell'oracolo che rappresenta la funzione `id`, subito dopo l'applicazione dell'oracolo.

(3) Dato che stiamo trasformando uno stato di due qubit, dobbiamo dire che cosa accade a ciascun qubit per ottenere la matrice. Lo facciamo usando di nuovo la funzione `tensor`.

(4) L'output è molto più facile da leggere: il registro è nello stato (|01⟩ - |11⟩) / √2.

(5) Nel nostro programma Q#, abbiamo usato l'istruzione `X` per riportare il qubit `target` a |0⟩ prima di rilasciarlo, operazione simulata dalla funzione QuTiP `sigmax` ().

(6) Poiché l'istruzione `X` inverte il suo argomento, l'applicazione della matrice X ci fornisce lo stato (|00⟩ - |10⟩) / √2.

(7) Possiamo usare QuTiP per confermare che un altro modo per scrivere (|00⟩ - |10⟩) / √2 è (H|1⟩) ⊗ |0⟩ = |-0⟩.

(8) Possiamo simulare l'operazione `MResetX` applicando `H` e quindi misurando in base Z.

(9) Lo stato del nostro registro, immediatamente prima della misurazione, ha un 1 nella riga 2 (10 in binario), quindi lo stato del nostro registro è |10⟩, e misureremo `One` con certezza.

> **SUGGERIMENTO**
>
> Usiamo l'operazione `MResetX` delle librerie standard Q# per misurare in base *X*. Una misurazione in base *X* restituisce `Zero` quando il suo argomento è nello stato |+⟩ e restituisce `One` quando il suo argomento è nello stato |-⟩. Pertanto, possiamo simulare l'operazione `MResetX` applicando `H` e poi eseguendo la misurazione in base *Z*.

Osservando il vettore finale del Listato D.5, possiamo vedere che rappresenta lo stato |10⟩. Se dovessimo misurare il qubit `control` da quello stato, otterremmo il bit classico `One` il 100% delle volte. Nel file `Algorithm.qs` che presentiamo nel Capitolo 8, diciamo che un oracolo è bilanciato se abbiamo misurato `One` sul qubit `control`, quindi concludiamo correttamente che `id` è un oracolo bilanciato. Il fatto che misureremo ogni volta `One` sul bit `control` è davvero interessante.

> **NOTA**

Sebbene alcuni algoritmi quantistici siano casuali, come l'esempio del generatore quantistico di numeri casuali del Capitolo 2 o il gioco di Morgana e Lancillotto del Capitolo 7, non è necessario che lo siano. In effetti, l'algoritmo di Deutsch-Jozsa è deterministico: otteniamo sempre la stessa risposta a ogni esecuzione.

Da questi esempi, traiamo un'osservazione importante (vedi Tabella D.5): l'applicazione di un oracolo ai qubit `control` e `target` può influenzare lo stato del qubit `control`.

Tabella D.5 I vettori che rappresentano lo stato del registro dopo l'applicazione dei vari oracoli.

Nome funzione	Stato del registro appena prima della misurazione	Risultato della misurazione del qubit control lungo Z
`zero`	[[1], [0], [0], [0]]	`Zero`
`one`	[[−1], [0], [0], [0]]	`Zero`
`id`	[[0], [0], [1], [0]]	`One`
`not`	[[0], [0], [−1], [0]]	`One`

Prefazione

Premessa

Ringraziamenti

- Sarah Kaiser
- Christopher Granade

Gli autori

Introduzione

- A chi è rivolto questo libro
- Come è organizzato questo libro: una tabella di marcia
- Il codice
- Altre risorse online
- Per approfondire

Parte I - Primi passi con il quantum computing

Capitolo 1 - Introduzione al quantum computing

- Perché il quantum computing è importante?
- Che cos'è un computer quantistico?
- Come utilizzeremo i computer quantistici?
- Che cos'è un programma?
- Riepilogo

Capitolo 2 - I qubit: gli elementi costitutivi

Perché abbiamo bisogno dei numeri casuali?
Che cosa sono i bit classici?
Qubit: stati e operazioni
Programmazione di un generatore quantistico di numeri casuali (QRNG)
Riepilogo

Capitolo 3 - Condivisione di segreti con la distribuzione quantistica delle chiavi

Tutto è lecito in amore e crittografia
La storia di due basi
Distribuzione quantistica delle chiavi: BB84
Utilizzo di una chiave segreta per inviare messaggi segreti
Riepilogo

Capitolo 4 - Giochi non locali: gestione di più qubit

Giochi non locali
Lavorare con più stati di qubit
Riepilogo

Capitolo 5 - Giochi non locali: implementazione di un simulatore multi-qubit

Oggetti quantistici in QuTiP
CHSH: strategia quantistica
Riepilogo

Capitolo 6 - Teletrasporto ed entanglement: trasferimento di dati quantistici

Trasferimento di dati quantistici
Tutte le rotazioni mono-qubit
Teletrasporto
Riepilogo
Parte I: conclusioni

Parte II - Programmazione di algoritmi quantistici in Q#

Capitolo 7 - Cambiare le probabilità: un'introduzione a Q#

Presentazione del Quantum Development Kit

Funzioni e operazioni in Q#

Passaggio di operazioni come argomenti

Il gioco di Morgana in Q#

Riepilogo

Capitolo 8 - Che cos'è un algoritmo quantistico?

Algoritmi classici e quantistici

L'algoritmo di Deutsch-Jozsa: piccoli miglioramenti per la ricerca

Gli oracoli: come rappresentare le funzioni classiche negli algoritmi quantistici

Simulazione dell'algoritmo di Deutsch-Jozsa in Q#

Le tecniche degli algoritmi quantistici

Phase kickback: la chiave del nostro successo

Riepilogo

Capitolo 9 - Sensori quantistici: non solo una fase

Stima della fase: utilizzo di proprietà utili dei qubit per le misurazioni

Tipi definiti dall'utente

Esecuzione di codice Q# da Python

Autostati e fasi locali

Applicazione controllata: trasformazione di fasi globali in fasi locali

Implementazione della migliore strategia di Lancillotto per il gioco della stima della fase

Riepilogo

Parte II: conclusioni

Parte III - Quantum computing applicato

Capitolo 10 - Soluzione di problemi di chimica con computer quantistici

Applicazioni di chimica per il quantum computing

I tanti percorsi che portano alla meccanica quantistica

Uso di matrici hamiltoniane per descrivere l'evoluzione dei sistemi quantistici nel corso del tempo

Rotazione attorno ad assi arbitrari con le operazioni di Pauli

Apportare i cambiamenti che vogliamo vedere nel sistema

Svolgere alcuni (piccolissimi) cambiamenti

Conclusione

Riepilogo

Capitolo 11 - Ricerca con computer quantistici

Ricerca di dati non strutturati

Riflessioni sugli stati

Implementazione dell'algoritmo di ricerca di Grover

Stima delle risorse

Riepilogo

Capitolo 12 - Aritmetica con computer quantistici

Fattorizzazione: quantum computing e sicurezza

Il legame fra matematica modulare e fattorizzazione

Algebra classica e fattorizzazione

Aritmetica quantistica

Soluzione completa

Riepilogo

Conclusioni

Appendice A - Installazione del software

Esecuzione degli esempi online

Installazione con Anaconda

Installazione del Quantum Development Kit

Appendice B - Glossario e riferimenti rapidi

Glossario

Notazione di Dirac

Operazioni quantistiche

Linguaggio Q#

Appendice C - Ripasso di algebra lineare

I vettori

Le matrici

Appendice D - L'algoritmo di Deutsch-Jozsa tramite esempi

Usare le nostre conoscenze per dimostrare le cose

Passo 1: preparazione dello stato di input per Deutsch-Jozsa

Passo 2: applicazione dell'oracolo

Passi 3 e 4: annullamento della preparazione sul qubit target e misurazione