Capitolo 1

Introduzione agli algoritmi

- Getto le basi per il resto del libro.
- Il primo algoritmo di ricerca (la ricerca binaria).
- Iniziamo a parlare del tempo di esecuzione di un algoritmo (notazione Big O).
- Una tecnica classica di progettazione di algoritmi (la ricorsività).

In questo capitolo

Primi passi

Un *algoritmo* è un insieme di istruzioni, il cui scopo è eseguire un'attività. Ogni frammento di codice, quindi, potrebbe essere chiamato algoritmo, ma questo libro ne tratta alcuni particolarmente interessanti. In questo libro ho scelto di trattare alcuni algoritmi veloci o che risolvono problemi interessanti o entrambe le cose. Ecco alcuni punti salienti.

- Il Capitolo 1 parla della ricerca binaria e mostra come un algoritmo può velocizzare il codice. In un esempio, il numero di passi necessari passa da 4 miliardi a solo 32.
- Un dispositivo GPS utilizza algoritmi a grafo (trattati nei Capitoli 6, 7 e 8) per calcolare il percorso più breve verso una destinazione.
- È possibile utilizzare la programmazione dinamica (trattata nel Capitolo 9) per scrivere un algoritmo di intelligenza artificiale in grado di giocare alla dama.

In ogni singolo caso, prima descriverò l'algoritmo e poi vi darò un esempio; quindi, parlerò del tempo di esecuzione dell'algoritmo nella notazione Big O;

infine, esplorerò quali altri tipi di problemi potrebbero essere risolti da quello stesso algoritmo.

Che cosa imparerete sulle prestazioni

La buona notizia è che un'implementazione di ogni algoritmo presentato in questo libro è probabilmente disponibile nel vostro linguaggio preferito, quindi non dovrete per forza scrivere ogni algoritmo! Ma quelle implementazioni sono inutili se non capite i compromessi cui andate incontro: i pro e i contro. In questo libro imparerete a confrontare i compromessi tra diversi algoritmi: dovreste usare il merge sort o il quicksort? Dovreste usare un array o una lista? Anche solo utilizzando una struttura dati differente potete ottenere una grande differenza.

Che cosa imparerete sulla risoluzione dei problemi

Imparerete tecniche di risoluzione dei problemi che forse ora non conoscete. Alcuni esempi.

- Se vi piace creare videogiochi, potete scrivere un sistema di intelligenza artificiale che segua le mosse dell'utente utilizzando algoritmi a grafo.
- Imparerete a creare un sistema di suggerimenti usando l'algoritmo KNN.
- Alcuni problemi non sono risolvibili rapidamente! La parte del libro che parla di problemi NP-completi vi mostra come identificarli e trovare un algoritmo che vi dia almeno una risposta approssimativa.

Più in generale, entro la fine di questo libro, conoscerete alcuni degli algoritmi più ampiamente generalizzabili. Potete quindi utilizzare le vostre nuove conoscenze per studiare algoritmi più specifici: per l'intelligenza artificiale, i database e così via. Oppure potrete affrontare sfide più difficili, sul lavoro.

Che cosa dovete sapere, per poter procedere

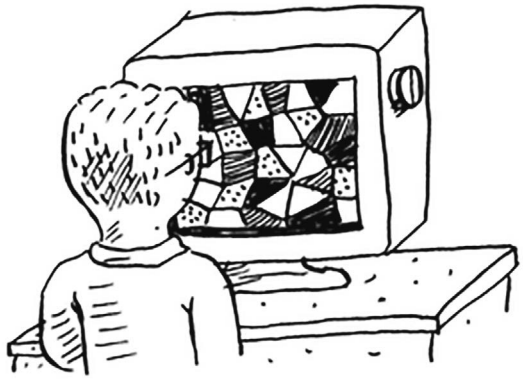
Avrete bisogno di semplici conoscenze di base di algebra. In particolare, data funzione $f(x) = x \times 2$, quanto vale f(5)? Avete risposto 10, benissimo, sapete quel che serve!
Inoltre, questo capitolo (e questo libro) sarà più facile da seguire se avete una certa familiarità con un linguaggio di programmazione. Tutti gli esempi in questo libro sono in Python. Se non conoscete nessun linguaggio di programmazione e volete impararne uno, scegliete Python, che è ottimo per i principianti. Se conoscete un altro linguaggio, come Ruby, tutto bene.

Ricerca binaria

Supponete di cercare una persona nell'elenco telefonico (oddio, che frase antiquata!). Il suo cognome inizia per *K*. Potreste iniziare dall'inizio e continuare a sfogliare le pagine

fino ad arrivare alle *K*. Ma è più probabile che inizierete da una pagina centrale, perché sapete che la *K* è più a metà dell'alfabeto.

Oppure supponete di cercare una parola in un dizionario e che inizi con *O*. Di nuovo, inizierete in prossimità del centro.
Supponete ora di accedere a Facebook. Facebook deve verificare che voi abbiate un account, quindi, deve cercare il vostro nome-utente nel suo database. Supponete che il vostro nome-utente sia `karlmageddon`. Facebook potrebbe iniziare dalla *A* e cercare il vostro nome, ma ha più senso che inizi da qualche parte, nel mezzo.
Questo è un problema di ricerca. E tutti questi casi usano lo stesso algoritmo per risolvere il problema: quello di *ricerca binaria*.



La ricerca binaria è un algoritmo; il suo input è una lista ordinata di elementi (spiegherò in seguito perché deve essere ordinata). Se l'elemento che state cercando è in quella

lista, la ricerca binaria restituisce la sua posizione. In caso contrario, la ricerca binaria restituisce `null`.
Per esempio, osservate la Figura 1.1.

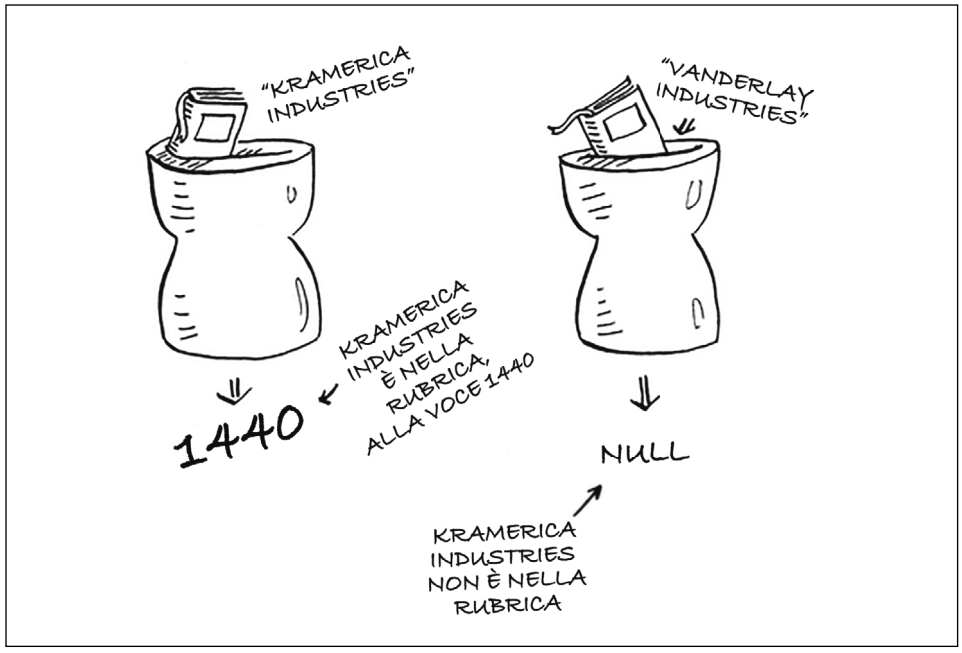


Figura 1.1 Ricerca di aziende in una rubrica telefonica con la ricerca binaria.

Ecco un esempio di come funziona la ricerca binaria. Sto pensando a un numero compreso tra 1 e 100.

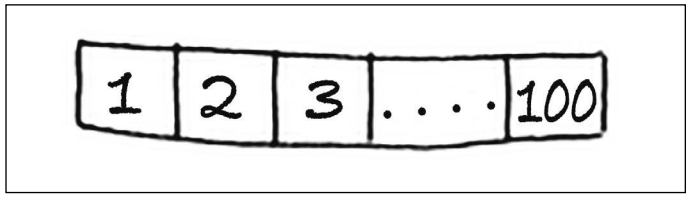


Figura 1.2

Dovete cercare di indovinare il mio numero nel minor numero di tentativi possibile. Io vi dirò se la vostra ipotesi è troppo bassa, troppo alta o corretta.
Supponiamo che voi iniziate a indovinare in questo modo: 1, 2, 3, 4, …. Ecco come andrebbe (Figura 1.3).
Questa è una *ricerca semplice* (ma forse sarebbe meglio chiamarla *ricerca stupida*). A ogni ipotesi, state eliminando un solo numero. Se il mio numero fosse 99, avreste bisogno di 99 tentativi per arrivarci!

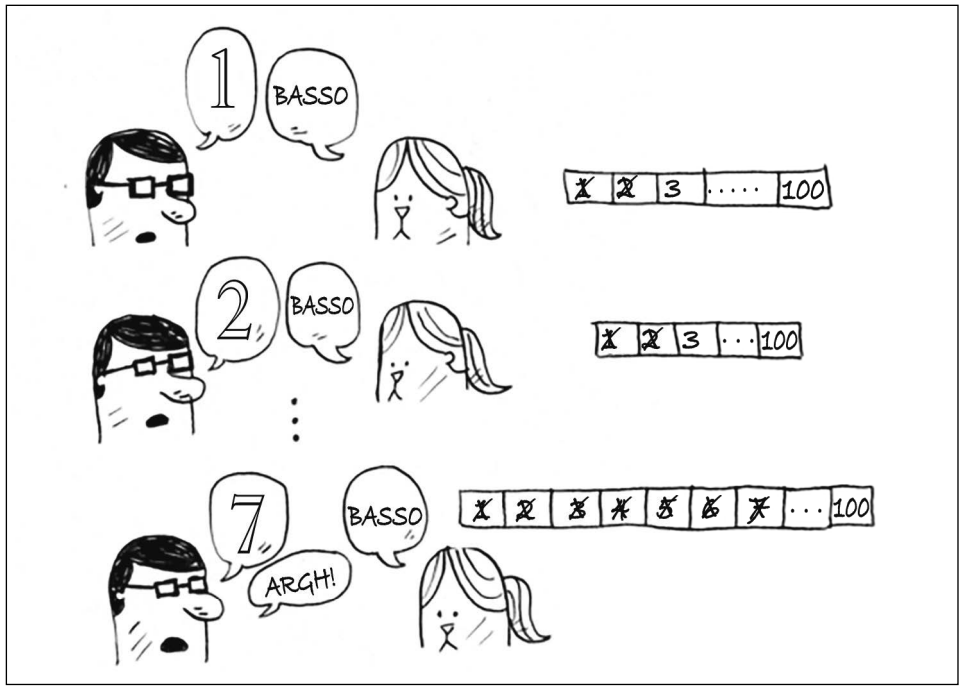


Figura 1.3 Un cattivo approccio per indovinare i numeri.

Un modo migliore per cercare

Ecco una tecnica migliore. Partite da 50.

Figura 1.4

Troppo basso, ma avete appena eliminato *metà* dei numeri! Ora sapete che i numeri da 1 a 50 sono tutti troppo bassi. Prossima ipotesi: 75.

Figura1.5

Troppo alto, ma di nuovo avete ridotto la metà dei numeri rimanenti! *Con la ricerca binaria, proponete il numero centrale ed eliminate ogni volta la metà dei numeri rimanenti*. Il prossimo numero è 63 (a metà tra 50 e 75).

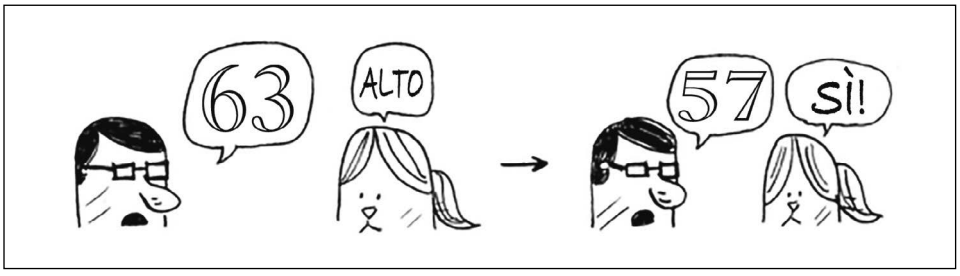


Figura 1.6

Questa è una ricerca binaria. Avete appena imparato il vostro primo algoritmo! Ecco quanti numeri potete eliminare ogni volta (Figura 1.7).

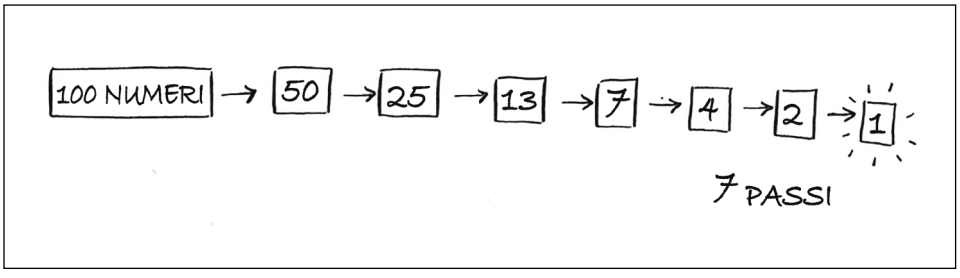


Figura 1.7 Con la ricerca binaria eliminate ogni volta la metà dei numeri.

Qualsiasi sia il numero che ho in mente, dovrete fare al massimo sette ipotesi, perché a ogni ipotesi eliminate così tanti numeri!
Supponiamo che stiate cercando una parola nel dizionario. Il dizionario contiene 240.000 parole. *Nel caso peggiore*, quanti passi pensate che richiederà ogni ricerca?

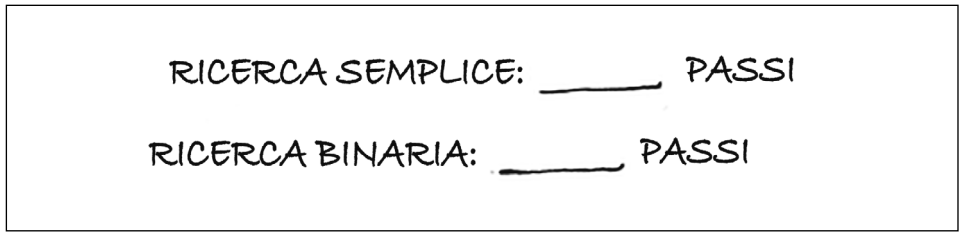


Figura 1.8

La ricerca semplice potrebbe richiedere 240.000 passi, se la parola che state cercando è l'ultima del dizionario. A ogni passo della ricerca binaria, invece, dimezzate il numero di parole rimanenti, finché non ne rimane una sola.

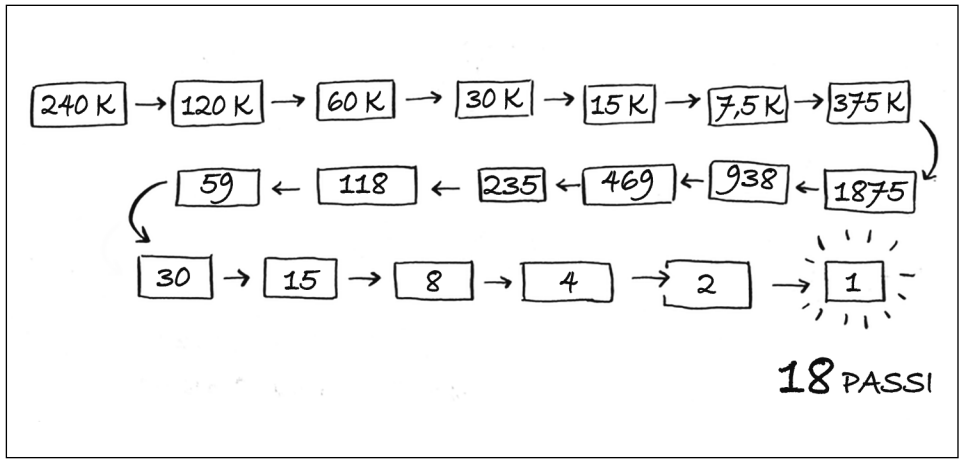


Figura 1.9

Quindi, la ricerca binaria richiederà 18 passi: una grande differenza! In generale, per qualsiasi lista di n elementi, la ricerca binaria richiederà il $\log_2 n$ passi, nel caso peggiore, mentre una ricerca semplice richiederà n passi.

Logaritmi

Forse non vi ricordate più che cosa sono i logaritmi, ma probabilmente conoscete gli esponenziali. $\log_{10} 100$ è come chiedere: "Quante volte dobbiamo moltiplicare 10 per se stesso per ottenere 100?". La risposta è 2: 10×10. Quindi $\log_{10} 100 = 2$. Per calcolare il log basta capovolgere l'esponente.

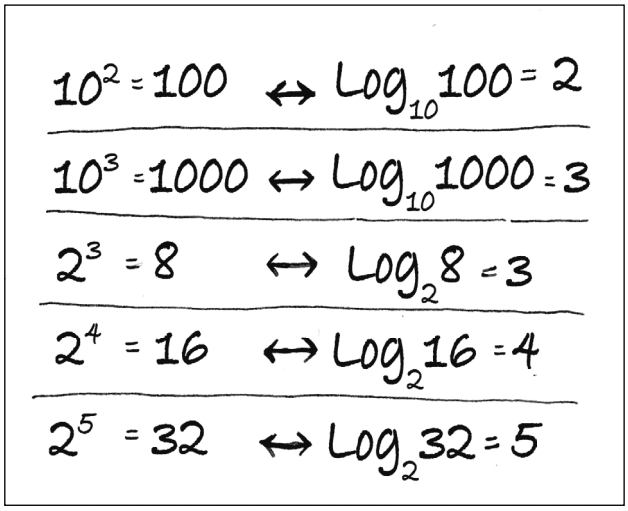


Figura 1.10 I logaritmi sono l'inverso degli elevamenti a potenza.

Quando parlo del tempo di esecuzione nella notazione Big O (ne parleremo più avanti), log significa sempre $\log_2$. Quando cercate un elemento usando la ricerca semplice, nel caso peggiore potreste dover controllare ogni singolo elemento. Quindi, per un elenco di 8 numeri, dovreste controllare anche 8 numeri. Per la ricerca binaria, nel caso peggiore dovete controllare log *n* elementi. Per un elenco di 8 elementi, log 8 == 3, perché $2^3 == 8$. Quindi per un elenco di 8 numeri, dovreste controllare al massimo 4 numeri. Per un elenco di 1024 elementi, log 1024 = 10, perché $2^{10} == 1024$. Quindi, per un elenco di 1024 numeri, dovrete controllare al massimo 11 numeri.

NOTA

Parlerò molto di tempo logaritmico in questo libro, quindi è bene acquisire come si deve il concetto di logaritmo.

NOTA

La ricerca binaria funziona solo quando la lista è già ordinata. Per esempio, in una rubrica i nomi sono in ordine alfabetico, quindi potete usare la ricerca binaria per cercare un nome. Che cosa accadrebbe se i nomi non fossero ordinati?

Vediamo come scrivere una ricerca binaria in Python. Il seguente esempio di codice usa gli array. Se non sapete come funzionano gli array, non preoccupatevi; ne parlerò nel prossimo capitolo. Dovete solo immaginare di memorizzare una sequenza di elementi in una fila di caselle consecutive, chiamata array. Le caselle sono numerate a partire da 0: la prima casella si trova nella posizione 0, la seconda nella posizione 1, la terza nella posizione 2 e così via.

La funzione `binary_search` accetta un array ordinato e un elemento. Se l'elemento è contenuto nell'array, la funzione ne restituisce la posizione. Occorre registrare in quale parte dell'array dovete cercare. All'inizio, si tratta dell'intero array:

```
low = 0
high = len(list) - 1
```

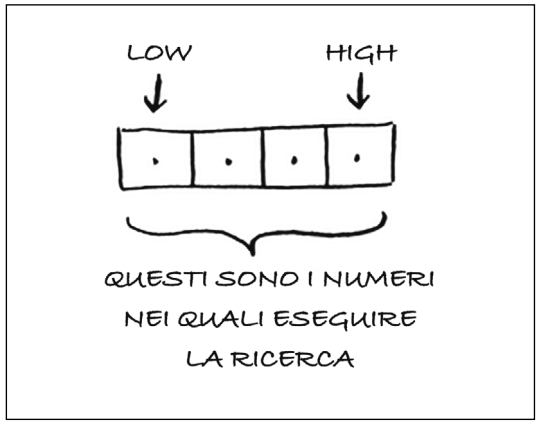


Figura 1.11

Ogni volta, controllate l'elemento centrale:

```
mid = (low + high) // 2  ❶
guess = list[mid]
```

❶ `mid` viene arrotondato automaticamente per difetto da Python se (`low + high`) non è un numero pari.

Se `guess` è troppo basso, aggiornate `low` in modo appropriato:

```
if guess < item:
    low = mid + 1
```

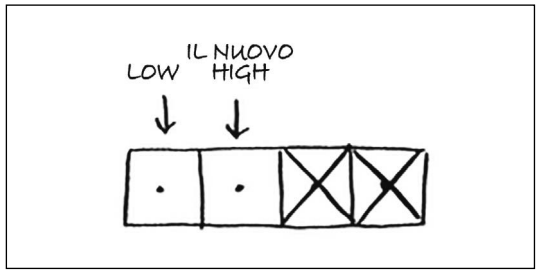


Figura 1.12

E se `guess` è troppo alto, aggiornate `high`. Ecco il codice completo:

```
def binary_search(list, item):
    low = 0  ❶
    high = len(list)-1  ❶

    while low <= high:  ❷
```

```
        mid = (low + high) // 2  ❸
        guess = list[mid]
        if guess == item:  ❹
            return mid
        if guess > item:  ❺
            high = mid - 1
        else:  ❻
            low = mid + 1
    return None  ❼

my_list = [1, 3, 5, 7, 9]  ❽

print binary_search(my_list, 3) # => 1  ❾
print binary_search(my_list, -1) # => None  ❿
```

❶ low e high tengono traccia della parte della lista in cui eseguirete la ricerca.

❷ Finché non avrete ristretto la ricerca a un solo elemento...

❸ ... controllate l'elemento centrale.

❹ Trovato l'elemento.

❺ guess era troppo alto.

❻ guess era troppo basso.

❼ L'elemento non è presente.

❽ Proviamolo!

❾ Ricordate, le liste iniziano da 0. La seconda casella ha indice 1.

❿ `None` significa nulla in Python. Indica che l'elemento non è stato trovato.

Esercizi

1. Supponete di avere una lista ordinata di 128 nomi e di applicarle una ricerca binaria. Qual è il numero massimo di passi necessari?
2. Supponete di raddoppiare le dimensioni della lista. Qual è il numero massimo di passi, adesso?

Tempo di esecuzione

Ogni volta che introduco un algoritmo, parlerò anche del suo tempo di esecuzione. In genere si desidera scegliere l'algoritmo più efficiente, indipendentemente dal fatto che si stia cercando di ottimizzare il tempo di esecuzione o lo spazio occupato.

Torniamo alla ricerca binaria. Quanto tempo risparmiate usandola? Bene, il primo approccio è stato quello di controllare ogni numero, uno per uno. Se questa è una lista di 100 numeri, avrete bisogno di fare fino a 100 ipotesi.
Se la lista contiene 4 miliardi di numeri, dovrete fare fino a 4 miliardi di ipotesi. Quindi il numero massimo di ipotesi corrisponde con le dimensioni della lista. Questo è chiamato *tempo lineare*.
La ricerca binaria è diversa. Se la lista è lunga 100 elementi, avrete bisogno, al massimo, di 7 tentativi. Se la lista è lunga 4 miliardi di elementi, avrete bisogno, al massimo, di 32 tentativi. Potente, vero? La ricerca binaria viene eseguita in un *tempo logaritmico* (o *tempo log*, come dicono gli iniziati). Ecco una tabella che riassume le nostre scoperte di oggi.

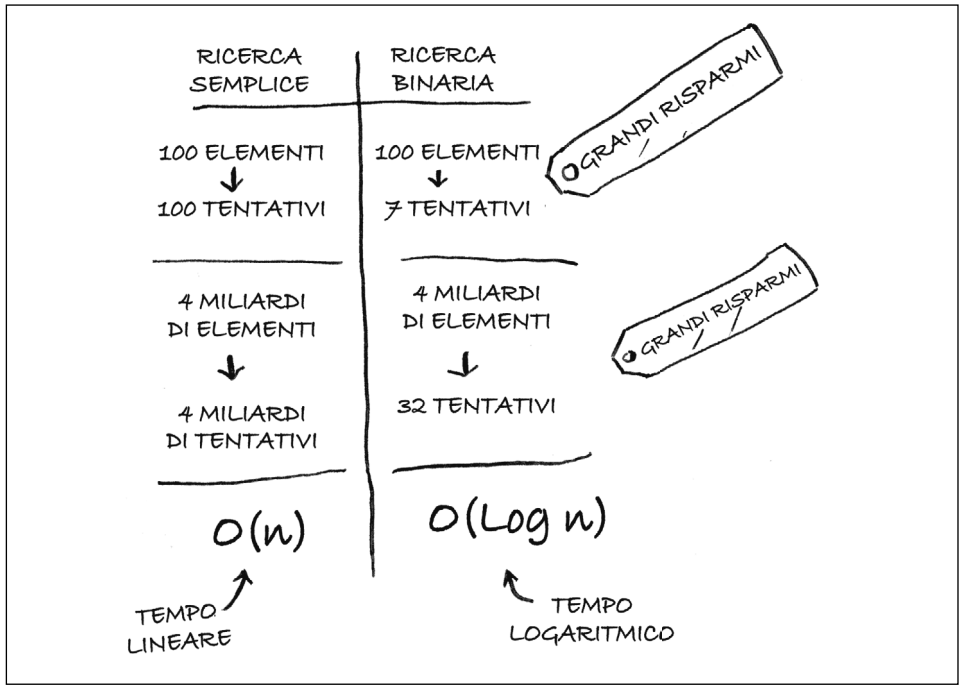


Figura 1.13 Tempi di esecuzione degli algoritmi di ricerca.

Notazione Big O

Big O è una speciale notazione che vi dice quanto è veloce un algoritmo. E chi se ne importa? Vi capiterà spesso di utilizzare algoritmi scritti da altri, ed è bene sapere se sono veloci o lenti. In questo paragrafo, vi spiegherò che cos'è la notazione Big O e vi fornirò un elenco dei tempi di esecuzione più comuni degli algoritmi.

I tempi di esecuzione di un algoritmo crescono a velocità differenti

Bob sta scrivendo un algoritmo di ricerca per la NASA. Il suo algoritmo verrà utilizzato quando la navicella sta per atterrare sulla Luna e aiuterà a calcolare il punto di atterraggio.

Questo è un esempio di come il tempo di esecuzione di due algoritmi può crescere a velocità differenti. Bob sta cercando di decidere tra una ricerca semplice e una ricerca binaria. L'algoritmo deve essere veloce e corretto. Da un lato, la ricerca binaria è più veloce, e Bob ha solo *10 secondi* per calcolare il punto di atterraggio, altrimenti la navicella andrà fuori rotta. D'altra parte, la ricerca semplice è più facile da scrivere, e ci sono meno possibilità che contenga bug. E Bob non vuole *davvero* lasciare ai bug il compito di far atterrare la navicella! Per maggiore sicurezza, Bob decide di cronometrare entrambi gli algoritmi con una lista di 100 elementi.

Supponete che ci voglia 1 millisecondo per controllare un elemento. Con una ricerca semplice, Bob deve controllare 100 elementi, quindi la ricerca impiega 100 ms. Al contrario, con la ricerca binaria deve controllare solo 7 elementi ($\log_2 100$ vale circa 7), e così la ricerca richiede solo 7 ms. Ma realisticamente, la lista conterrà più di un miliardo di elementi. In tal caso, quanto tempo impiegherà la ricerca semplice? E quanto tempo impiegherà la ricerca binaria? Assicuratevi di avere in mente una risposta a queste due domande, prima di continuare a leggere.

Figura 1.14 Tempo di esecuzione per la ricerca semplice e per la ricerca binaria, con una lista di 100 elementi.

Bob esegue la ricerca binaria con un miliardo di elementi, e impiega 30 ms ($\log_2$ 1.000.000.000 è circa 30). "30 ms!", pensa. "La ricerca binaria è circa quindici volte più veloce della ricerca semplice, perché la ricerca semplice ha richiesto 100 ms con 100 elementi e la ricerca binaria ha impiegato 7 ms. Quindi una ricerca semplice su un miliardo di elementi richiederà 30 × 15 = 450 ms, giusto? Molto al di sotto della mia soglia di 10 secondi. Bob decide di adottare una ricerca semplice. È la scelta giusta? No. Bob ha torto. Torto marcio. Il tempo di esecuzione per una ricerca semplice con un miliardo di elementi sarà di un miliardo di ms, ovvero 11 giorni! Il problema è che i tempi di esecuzione per la ricerca binaria e per la ricerca semplice *non crescono alla stessa velocità*.

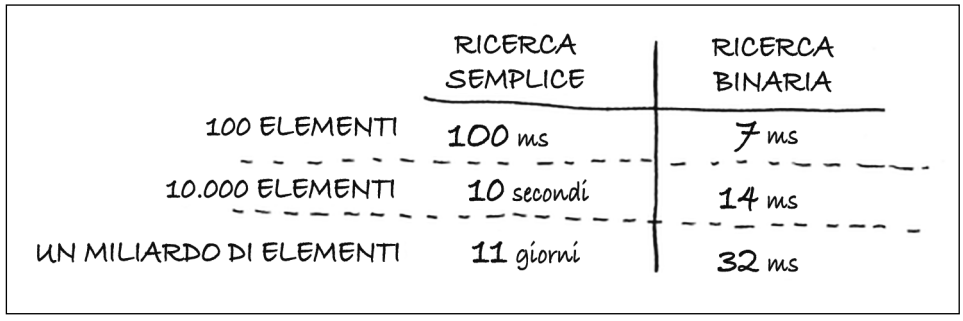


Figura 1.15 I tempi di esecuzione crescono a velocità molto differenti.

Cioè: all'aumentare del numero di elementi, la ricerca binaria richiede solo un po' più di tempo per essere eseguita, mentre la ricerca semplice richiede *molto* più tempo. Quindi, man mano che la lista dei numeri diventa più grande, la ricerca binaria diventa *sempre* più veloce della ricerca semplice. Bob pensava che la ricerca binaria fosse sempre 15 volte più veloce della ricerca semplice, ma non è così. Se la lista contiene un miliardo di elementi, sarà circa 33 milioni di volte più veloce. Ecco perché non è sufficiente conoscere il tempo di esecuzione di un algoritmo: è necessario sapere come aumenta il tempo di esecuzione all'aumentare delle dimensioni della lista. È qui che entra in gioco la notazione Big O.

La notazione Big O vi dice quanto è veloce un algoritmo. Per esempio, supponete di avere una lista di dimensioni *n*. La ricerca semplice deve controllare ogni elemento, quindi ci vorranno *n* operazioni. Il tempo di esecuzione nella notazione Big O è O(*n*). E… dove sono i secondi? Non ce ne sono: Big O non vi dà una velocità in secondi. *La notazione Big O vi consente di confrontare il numero di operazioni*. Vi dice quanto velocemente cresce l'algoritmo.

Ecco un altro esempio. La ricerca binaria richiede operazioni log *n* per controllare una lista di dimensioni *n*. Qual è il tempo di esecuzione nella notazione Big O? È O(log *n*). In generale, la notazione Big O è scritta come segue.

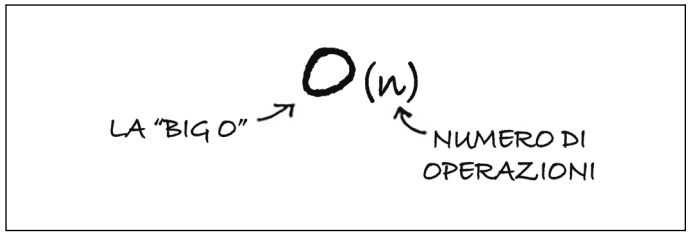


Figura 1.16 L'aspetto della notazione Big O.

Questo vi dice il numero di operazioni che eseguirà un algoritmo. Si chiama notazione Big O ed è rappresentata da una "O" maiuscola.
Ora vediamo alcuni esempi. Cercate di immaginare il tempo di esecuzione di questi algoritmi.

Visualizzazione di diversi tempi di esecuzione di Big O

Ecco un esempio pratico che potete seguire a casa con dei fogli di carta e una matita. Supponete di dover disegnare una griglia di 16 caselle.

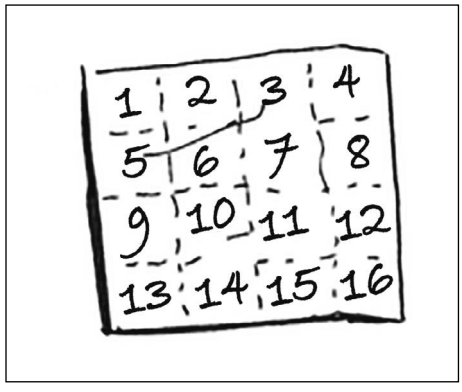


Figura 1.17 Qual è un buon algoritmo per disegnare questa griglia?

Algoritmo 1

Un modo per farlo è disegnare 16 caselle, una alla volta. Ricordate: la notazione Big O conta il numero di operazioni. In questo esempio, un'operazione è "disegnare una casella". Dovete disegnare 16 caselle. Quante operazioni ci vorranno, disegnando una casella per volta?

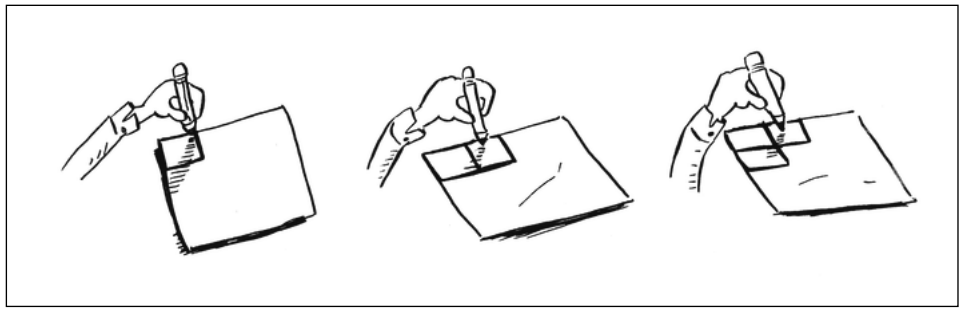


Figura 1.18 Disegnare la griglia una casella alla volta.

Occorrono 16 passi per disegnare 16 caselle. Qual è il tempo di esecuzione di questo algoritmo?

Algoritmo 2

Provate invece questo algoritmo. Piegate la carta.

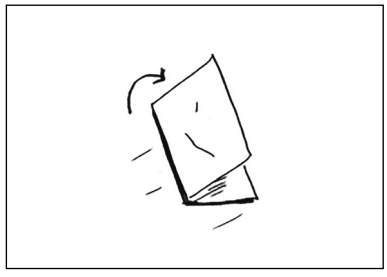


Figura 1.19

In questo esempio, un'operazione è "piegare la carta". Avete appena realizzato due caselle con una sola operazione!
Piegate la carta ancora, e poi ancora e ancora.

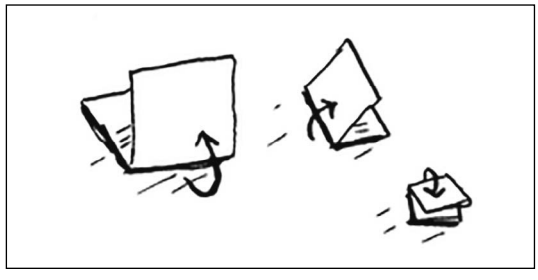


Figura 1.20

Aprite il foglio dopo quattro pieghe e avrete una bellissima griglia! Ogni piega raddoppia il numero di caselle. Avete realizzato 16 caselle con 4 operazioni!

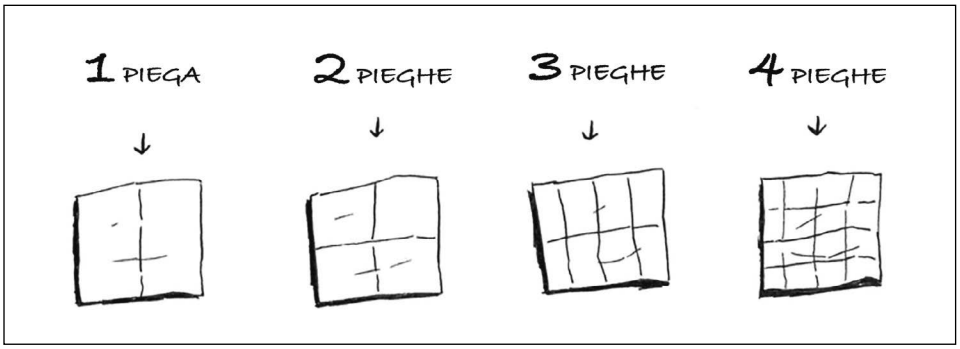


Figura 1.21 Disegnare una griglia in quattro pieghe.

Potete "disegnare" il doppio delle caselle con ogni piega, quindi potete creare 16 caselle in 4 passi. Qual è il tempo di esecuzione di questo algoritmo? Trovate i tempi di esecuzione per entrambi gli algoritmi, prima di procedere.
Risposte: l'algoritmo 1 impiega O(*n*) e l'algoritmo 2 impiega O(log *n*).

La notazione Big O stabilisce un tempo di esecuzione nel caso peggiore

Supponete di utilizzare una ricerca semplice per cercare una persona nella rubrica. Sapete che la ricerca semplice richiede un tempo O(*n*) per essere eseguita, il che significa che nel caso peggiore dovrete esaminare ogni singola voce della rubrica. In questo caso, state cercando Adit. Questo tipo è la prima voce nella vostra rubrica. Quindi non dovrete guardare ogni voce: l'avete trovata al primo tentativo. Questo algoritmo ha impiegato un tempo O(*n*)? O ci è voluto un tempo O(1), perché avete trovato la persona al primo tentativo? La ricerca semplice richiede ancora un tempo O(*n*). In questo caso, avete trovato immediatamente quello che stavate cercando. Questo è lo scenario migliore. Ma la notazione Big O considera lo scenario *peggiore*. Quindi potete dire che, nel *caso peggiore*, dovrete controllare ogni voce della rubrica. È il tempo O(*n*). È anche una rassicurazione: sapete che una ricerca semplice non sarà mai più lenta del tempo O(*n*).

NOTA

Oltre al tempo di esecuzione nel caso peggiore, è anche importante considerare il tempo di esecuzione nel caso medio. Il caso peggiore rispetto al caso medio è discusso nel Capitolo 4.

Alcuni tempi Big O comuni

Ecco cinque tempi di esecuzione Big O che incontrerete molto spesso, ordinati dal più veloce al più lento.

- O(log *n*), noto anche come *tempo logaritmico*. Esempio: la ricerca binaria.
- O(*n*), noto anche come *tempo lineare*. Esempio: la ricerca semplice.
- O(*n* * log *n*). Esempio: un algoritmo di ordinamento veloce, come il quicksort (lo vediamo nel Capitolo 4).

- O(n^2). Esempio: un algoritmo di ordinamento lento, come il selection sort (lo vediamo nel Capitolo 2).
- O(n!). Esempio: un algoritmo davvero lento, come quello del commesso viaggiatore (proprio di seguito).

Supponete di disegnare di nuovo una griglia di 16 caselle e di poter scegliere tra 5 diversi algoritmi per farlo. Se usate il primo algoritmo, ci impiegherà un tempo O(log n) per disegnare la griglia. Potete fare 10 operazioni al secondo. Con un tempo O(log n), ci vorranno 4 operazioni per disegnare una griglia di 16 caselle (log 16 è 4). Quindi ci vorranno 0,4 secondi per disegnare la griglia. E se doveste disegnare 1.024 caselle? Ci vorranno 1.024 = 10 operazioni, o 1 secondo per disegnare una griglia di 1.024 caselle. Questi numeri considerano il primo algoritmo.
Il secondo algoritmo è più lento: impiega un tempo O(n). Ci vorranno 16 operazioni per disegnare 16 caselle e ci vorranno 1.024 operazioni per disegnare 1.024 caselle. Quanto tempo è, in secondi?
Ecco quanto tempo ci vorrebbe per disegnare una griglia per il resto degli algoritmi, dal più veloce al più lento.

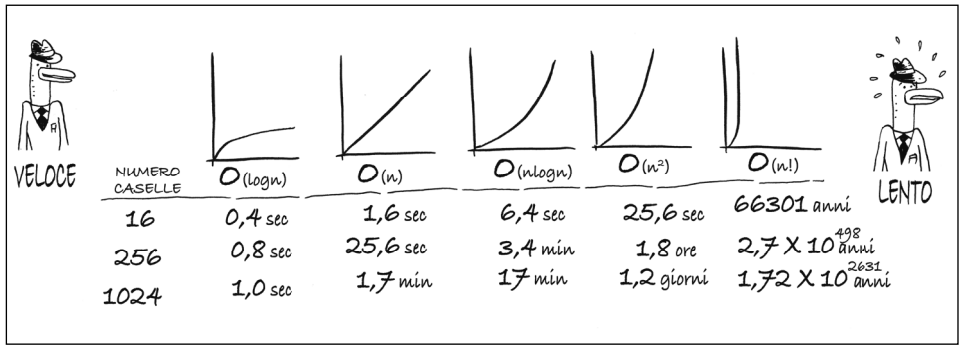


Figura 1.22

Ci sono anche altri tempi di esecuzione, ma questi sono i cinque più comuni.
Questa è una semplificazione. In realtà non è possibile convertire con questa facilità un tempo di esecuzione Big O in un numero di operazioni, ma per i nostri scopi l'approssimazione è sufficiente. Torneremo sulla notazione Big O nel Capitolo 4, dopo aver appreso qualche altro algoritmo. Per ora, i concetti principali sono i seguenti.

- La velocità dell'algoritmo non si misura in secondi, ma in crescita del numero di operazioni.
- Parliamo invece di quanto velocemente il tempo di esecuzione di un algoritmo aumenta all'aumentare della dimensione dell'input.
- Il tempo di esecuzione degli algoritmi è espresso in notazione Big O.
- O(log n) è più veloce di O(n), ma diventa molto più veloce man mano che crescono le dimensioni della lista in cui state eseguendo la ricerca.

Esercizi

Indicate il tempo di esecuzione per ciascuno di questi scenari in termini di Big O.

3. Avete il cognome di una persona e volete trovare il suo numero di telefono nella rubrica.
4. Avete un numero di telefono di una persona e volete trovare il suo cognome nella rubrica. (Suggerimento: dovrete cercare in tutta la rubrica!).
5. Volete leggere i numeri di telefono di ogni persona presente nella rubrica.
6. Volete leggere solo i numeri delle *A*. (Questo è complicato! Coinvolge concetti che saranno trattati più dettagliatamente nel Capitolo 4. Leggete la risposta: potreste essere sorpresi!)

Il commesso viaggiatore

Potreste aver letto il paragrafo precedente e aver pensato: "Assolutamente cercherò di non imbattermi mai in un algoritmo che richiede un tempo O(*n*!)". Bene, lasciatemi provare a dimostrare che avete torto! Ecco un esempio di un algoritmo con un tempo di esecuzione davvero pessimo. Questo è un problema famoso, perché la sua crescita è spaventosa e molte persone molto intelligenti pensano che non possa essere migliorato. Si chiama il problema del *commesso viaggiatore*.
Avete un venditore.

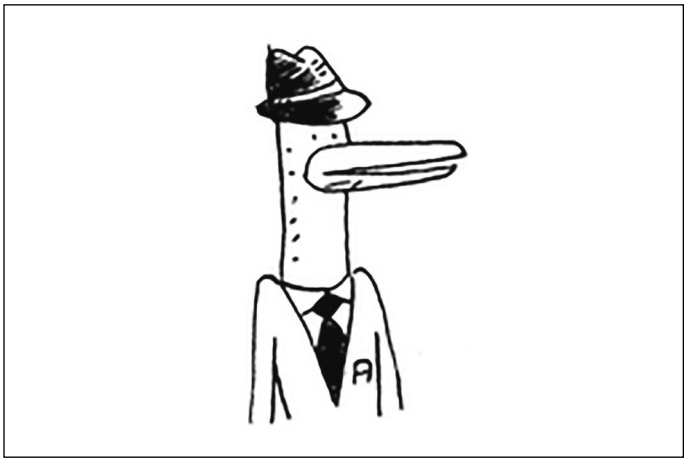


Figura 1.23

Il venditore deve andare in cinque città.

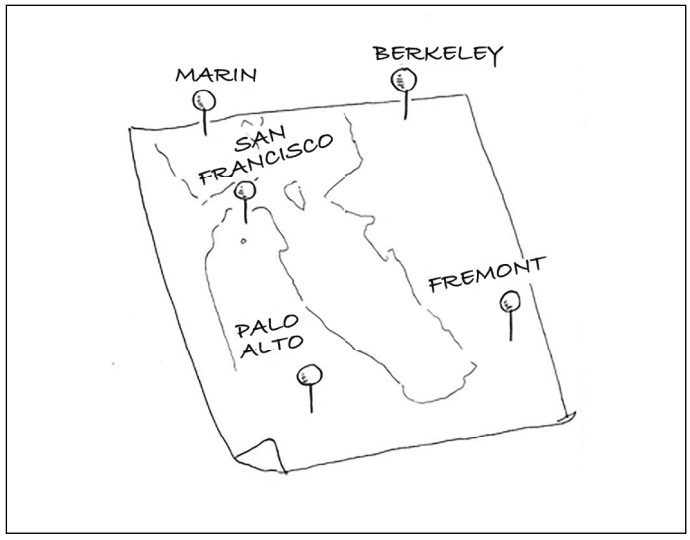


Figura 1.24

Questo venditore, che chiamerò Opus, vuole raggiungere tutte e cinque le città percorrendo la distanza minima. Ecco un modo per farlo: considerare ogni possibile ordine in cui potrebbe visitare le città.

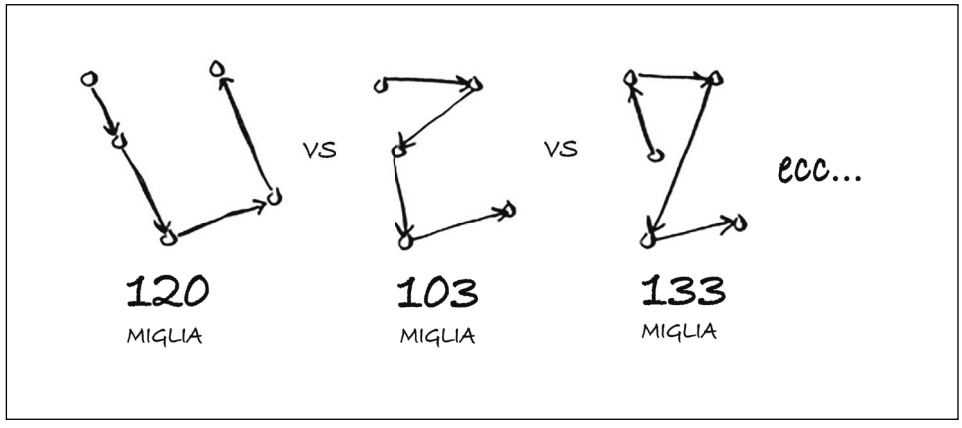


Figura 1.25

Somma le distanze, calcola i totali e poi sceglie il percorso più breve. Cinque città danno 120 permutazioni, quindi ci vorranno 120 operazioni per risolvere il problema, per cinque città. Per sei città, ci vorranno 720 operazioni (per 720 permutazioni). Per sette città, ci vorranno 5.040 operazioni!

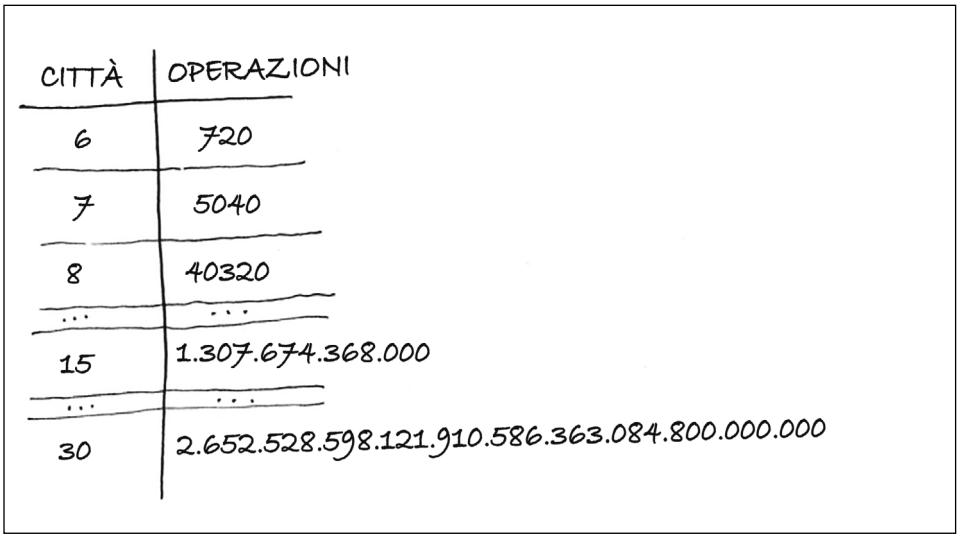


Figura 1.26 Il numero di operazioni aumenta drasticamente.

In generale, per *n* elementi, ci vorranno *n*! (*n* fattoriale) operazioni per calcolare il risultato. Quindi questo è un tempo O(*n*!), o *tempo fattoriale*. Occorrono molte operazioni per qualsiasi calcolo, tranne che per i numeri più piccoli. E se avete a che fare con più di 100 città, diventerà impossibile calcolare la risposta in tempo: farà in tempo a spegnersi il Sole. Questo è un algoritmo terribile! Opus dovrebbe usarne un altro, giusto? Ma non può. Questo è uno dei problemi irrisolti dell'informatica. Non esiste un algoritmo veloce per risolvere questo problema, e molti pensano che sia *impossibile* trovare un algoritmo intelligente per "domare" questo problema. Il meglio che possiamo fare è trovare una soluzione approssimativa, come vedremo nel Capitolo 10. Un'ultima nota: se conoscete l'argomento, provate con gli alberi binari! Ne trovate una breve descrizione nell'ultimo capitolo.

Riepilogo

- La ricerca binaria è molto più veloce della ricerca semplice.
- O(log *n*) è più veloce di O(*n*), ma diventa molto più veloce quando la lista di elementi che state cercando cresce.
- La velocità dell'algoritmo non si misura in secondi.
- I tempi degli algoritmi sono misurati in termini di *crescita* di un algoritmo in base al numero di elementi.
- I tempi degli algoritmi sono scritti in notazione Big O.