Capitolo 1

Perché le strutture dati sono così importanti

In questo capitolo

- **Strutture dati**
- **L'array: la struttura dati fondamentale**
- **Misurazione della velocità**
- **Lettura**
- **Ricerca**
- **Inserimento**
- **Eliminazione**
- **Il set: come una singola regola può influire sull'efficienza**
- **Conclusioni**
- **Esercizi**

Quando si impara a programmare, l'obiettivo è, e *dovrebbe essere*, far funzionare correttamente il codice. Il codice prodotto viene però misurato utilizzando una metrica semplice: funziona?

A mano a mano che gli sviluppatori acquisiscono maggiore esperienza, iniziano a scoprire ulteriori livelli e sfumature relativi alla *qualità* del loro codice. Scoprono che due frammenti di codice possono anche svolgere lo stesso identico compito, ma che uno è *migliore* dell'altro.

Esistono molti sistemi per misurare la qualità del codice. Una misura importante è la manutenibilità, che coinvolge aspetti come la leggibilità, l'organizzazione e la modularità del codice.

Tuttavia, un altro aspetto del codice di qualità è l'*efficienza*. Per esempio, potete avere due frammenti di codice che raggiungono entrambi lo stesso obiettivo, ma *uno* è *più veloce dell'altro*.

Osservate queste due funzioni. Entrambe mostrano tutti i numeri pari da 2 a 100:

```
def print_numbers_version_one():
    number = 2

    while number <= 100:
        # Se il numero è pari, mostralo:
        if number % 2 == 0:
            print(number)

        number += 1
```

```
def print_numbers_version_two():
    number = 2

    while number <= 100:
        print(number)

        # Aumenta il numero di 2, che, per definizione,
        # è il numero pari successivo:
        number += 2
```

Quale di queste funzioni pensate che funzioni più velocemente?
Se avete risposto la Versione 2, avete ragione. Questo perché la Versione 1 finisce per eseguire il ciclo cento volte, mentre la Versione 2 esegue il ciclo solo cinquanta volte. La prima versione, quindi, richiede il doppio dei passaggi della seconda.
Questo libro tratta proprio l'argomento della scrittura di *codice efficiente*. Avere la capacità di scrivere codice che venga eseguito rapidamente è un aspetto importante per diventare sviluppatori di software.
Il primo passo per scrivere codice veloce consiste nel capire che cosa sono le strutture dati e in che modo le diverse strutture dati possono influenzare la velocità del nostro codice. Quindi… tuffiamoci.

Strutture dati

Parliamo dei dati.
Dati è un termine generico, che si riferisce a tutti i tipi di informazioni, fin dai numeri e dalle stringhe più elementari. Nel semplice ma classico programma "Hello World!", la stringa `"Hello World!"` è un dato. In effetti, anche i dati più complessi solitamente si scompongono in numeri e stringhe.
Le *strutture dati* si riferiscono al modo in cui i dati sono *organizzati*. Come scoprirete, gli stessi dati possono essere organizzati in modi differenti.
Osserviamo il seguente codice:

```
x = "Hello! "
y = "How are you "
z = "today?"

print(x + y + z)
```

Questo semplice programma gestisce tre diversi dati, producendo tre stringhe per produrre un messaggio coerente. Se dovessimo descrivere come sono organizzati i dati in questo programma, diremmo che abbiamo tre stringhe indipendenti, ciascuna delle quali è contenuta in una singola variabile.
Tuttavia, questi stessi dati possono anche essere memorizzati in un array:

```
array = ["Hello! ", "How are you ", "today?"]

print(array[0] + array[1] + array[2])
```

In questo libro scoprirete che l'organizzazione dei dati non è importante solo per il bene dell'azienda per cui lavorate, ma che può avere un impatto significativo anche sulla *velocità di esecuzione del codice*. A seconda di come scegliete di organizzare i vostri dati, il vostro programma può funzionare più o meno velocemente anche di parecchi ordini di grandezza. E se state creando un programma che deve gestire molti dati o un'app web utilizzata da migliaia di persone contemporaneamente, le strutture dati che scegliete di usare potrebbero influire sul fatto che il vostro software funzioni o si blocchi perché non riesce a gestire il carico.

Quando avrete una conoscenza più approfondita delle implicazioni prestazionali delle strutture dati sul software che state creando, avrete le chiavi per scrivere codice veloce ed elegante e la vostra esperienza come sviluppatori sarà notevolmente migliorata.

In questo capitolo inizieremo la nostra analisi da due strutture dati: gli array e gli insiemi. Sebbene le due strutture possano sembrare quasi identiche, imparerete gli strumenti per analizzare le implicazioni sulle prestazioni di ciascuna di esse.

L'array: la struttura dati fondamentale

L'array è una delle strutture dati più importanti della computer science. In Python, la struttura dati built-in più simile a un array è chiamata *lista*, ma mi riferirò a essa parlando di array, rimanendo in linea con il termine tecnico più generale. Presumo che abbiate già lavorato con gli array, quindi che siate consapevoli del fatto che un array è un elenco di elementi di dati. L'array è versatile e può rivelarsi uno strumento utile in molte situazioni, ma esaminiamo un rapido esempio.

Se state esaminando il codice sorgente di un'applicazione che consente agli utenti di creare e utilizzare liste della spesa per un negozio di alimentari, potreste trovare codice come questo:

```
array = ["apples", "bananas", "cucumbers", "dates", "elderberries"]
```

Questo array contiene cinque stringhe, ciascuna delle quali rappresenta qualcosa da comprare al supermercato.

Gli array hanno un loro gergo tecnico. Le *dimensioni* di un array indicano il numero di elementi contenuti nell'array. L'array della nostra lista della spesa ha dimensioni pari a 5, poiché contiene cinque valori. L'*indice* di un array è il numero che identifica la posizione di un dato all'interno dell'array. Nella maggior parte dei linguaggi di programmazione, l'indice parte da `0`. Quindi, per il nostro array di esempio, `"apples"` è all'indice `0` e `"elderberries"` è all'indice `4`, come possiamo vedere nella Figura 1.1.

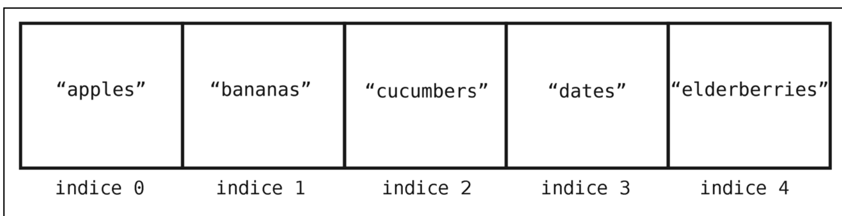


Figura 1.1

Operazioni su una struttura dati

Per comprendere le prestazioni di una struttura dati, come l'array, dobbiamo analizzare i modi in cui il nostro codice può interagire con quella struttura dati.
Molte strutture dati vengono utilizzate in quattro modi fondamentali, che chiamiamo *operazioni*.

- *Lettura*: si riferisce alla ricerca di qualcosa in un determinato punto della struttura dati. Con un array, ciò significa cercare il valore situato a un determinato indice. Per esempio, cercare l'articolo della spesa che si trova all'indice `2` è un'operazione di *lettura* dall'array.
- *Ricerca*: si riferisce alla ricerca di un valore all'interno di una struttura dati. Nel caso di un array, significa verificare se un determinato valore esiste all'interno dell'array e, in tal caso, a quale indice. Per esempio, cercare l'indice di `"dates"` nella nostra lista della spesa è una *ricerca* nell'array.
- *Inserimento*: si riferisce all'aggiunta di un nuovo valore alla struttura dati. Nel caso di un array, significa aggiungere un nuovo valore in una posizione all'interno dell'array. Se alla nostra lista della spesa aggiungessimo `"figs"`, *inseriremmo* un nuovo valore nell'array.
- *Eliminazione*: si riferisce alla rimozione di un valore dalla struttura dati. Nel caso di un array, significa rimuovere dall'array uno dei valori. Per esempio, se rimuovessimo `"bananas"` dalla lista della spesa, tale valore verrebbe *eliminato* dall'array.

In questo capitolo analizzeremo la velocità di ciascuna di queste operazioni, applicata a un array.

Misurazione della velocità

Ma come si misura la velocità di un'operazione?
Se da questo libro volete togliere un argomento, può essere questo: quando misuriamo la velocità di un'operazione, non ci riferiamo a quanto *tempo* impiega l'operazione, ma quanti *passaggi* richiede.
In realtà lo abbiamo già visto in precedenza nel contesto della visualizzazione dei numeri pari da 2 a 100. La seconda versione di quella funzione era più veloce perché richiedeva la metà dei passaggi rispetto alla prima.
Perché misuriamo la velocità del codice in termini di passaggi?
Lo facciamo perché non possiamo mai dire con certezza che un'operazione richiede, diciamo, cinque secondi. Mentre un frammento di codice può richiedere cinque secondi su un certo computer, lo stesso frammento di codice potrebbe richiedere più tempo su un hardware più vecchio. Del resto, lo stesso codice potrebbe funzionare molto più velocemente sui computer di domani. Misurare la velocità di un'operazione in termini di tempo è poco utile, poiché il tempo cambierà sempre a seconda dell'hardware sul quale viene eseguita.
Al contrario, possiamo misurare la velocità di un'operazione in termini di numero di *passaggi computazionali* necessari. Se l'operazione A richiede 5 passaggi e l'operazione B richiede 500 passaggi, possiamo supporre che l'operazione A sarà sempre più veloce dell'operazione B indipendentemente dal componente hardware impiegato. Misurare il numero di passaggi è quindi la chiave per analizzare la velocità di un'operazione.

La misurazione della velocità di un'operazione è anche chiamata misurazione della sua *complessità temporale*. In questo libro utilizzerò i termini *velocità*, *complessità temporale*, *efficienza*, *prestazioni* e *runtime* in modo un po' intercambiabile. Si riferiscono tutti al numero di passaggi necessari per svolgere una determinata operazione.
Passiamo alle quattro operazioni eseguibili su un array e determiniamo quanti passaggi richiede ciascuna di esse.

Lettura

La prima operazione che considereremo è la *lettura*, che cerca quale valore è contenuto a un determinato indice all'interno dell'array.
Un computer può leggere un dato da un array in un solo passaggio. Questo perché il computer ha la capacità di saltare a qualsiasi indice dell'array. Nel nostro esempio `["apples, "bananas", "cucumbers", "dates", "elderberries"]`, se fossimo interessati a leggere l'elemento all'indice 2, il computer salterebbe direttamente a quell'indice e segnalerebbe che contiene il valore `"cucumbers"`.
In che modo il computer è in grado di cercare l'indice di un array in un solo passaggio? Vediamo come fa.
La memoria di un computer può essere considerata come un gigantesco insieme di celle. Nella Figura 1.2 potete vedere una griglia di celle: alcune sono vuote e altre contengono dei dati.

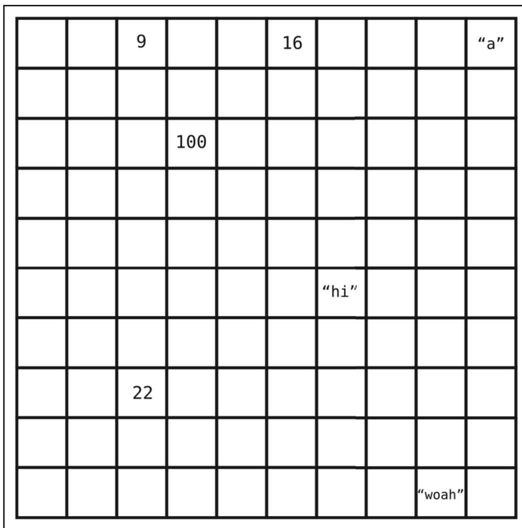


Figura 1.2

Sebbene questa immagine sia una semplificazione del funzionamento della memoria del computer, ne rappresenta l'idea essenziale.
Quando un programma dichiara un array, alloca un insieme contiguo di celle vuote, che intende utilizzare. Quindi, se state creando un array destinato a contenere cinque elementi,

il vostro computer cercherà in memoria un gruppo di cinque celle vuote contigue e lo designerebbe per funzionare come un array (Figura 1.3).

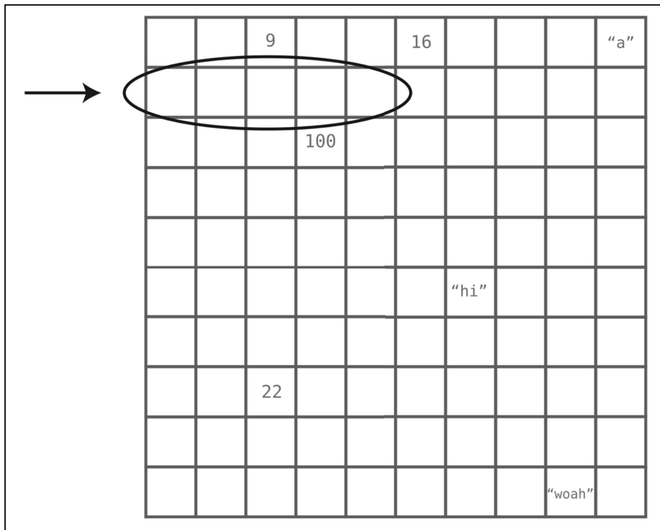


Figura 1.3

Ora, ogni cella della memoria di un computer ha un suo indirizzo. È un po' come un indirizzo stradale, tranne per il fatto che è rappresentato da un numero. L'indirizzo di memoria di ciascuna cella è maggiore di un'unità dell'indirizzo della cella precedente. La Figura 1.4 mostra l'indirizzo di memoria di ciascuna cella.

Nella Figura 1.5, potete vedere l'array della nostra lista della spesa con i suoi indici e indirizzi di memoria.

Quando il computer legge il valore ospitato a un determinato indice di un array, può saltare direttamente a quell'indice a causa della combinazione dei seguenti fatti.

1. Un computer può passare a qualsiasi *indirizzo di memoria* in un solo passaggio. Per esempio, se chiedete a un computer di leggere ciò che si trova all'indirizzo di memoria 1063, può accedervi senza alcun processo di ricerca. Per analogia, se io vi chiedessi di alzare il mignolo destro, non dovreste cercare tra tutte le dita per scoprire qual è il mignolo destro. Sareste in grado di identificarlo immediatamente.
2. Ogni volta che un computer alloca un array, prende nota anche dell'indirizzo di memoria in cui *inizia*. Quindi, se chiedessimo al computer di trovare il primo elemento dell'array, sarebbe in grado di accedere immediatamente all'indirizzo di memoria appropriato per trovarlo.

Ora, questi fatti spiegano come il computer possa trovare il *primo* valore di un array in un unico passaggio. Tuttavia, un computer può anche trovare il valore contenuto in *qualsiasi* indice, eseguendo una semplice addizione. Se chiedessimo al computer di trovare il valore contenuto nell'indice 3, il computer prenderebbe semplicemente l'indirizzo di memoria nell'indice 0 e gli aggiungerebbe 3 (dopotutto, gli indirizzi di memoria sono sequenziali).

1000	1001	1002	1003	1004	1005	1006	1007	1008	1009
1010	1011	1012	1013	1014	1015	1016	1017	1018	1019
1020	1021	1022	1023	1024	1025	1026	1027	1028	1029
1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
1040	1041	1042	1043	1044	1045	1046	1047	1048	1049
1050	1051	1052	1053	1054	1055	1056	1057	1058	1059
1060	1061	1062	1063	1064	1065	1066	1067	1068	1069
1070	1071	1072	1073	1074	1075	1076	1077	1078	1079
1080	1081	1082	1083	1084	1085	1086	1087	1088	1089
1090	1091	1092	1093	1094	1095	1096	1097	1098	1099

Figura 1.4

	"apples"	"bananas"	"cucumbers"	"dates"	"elderberries"
memory address:	1010	1011	1012	1013	1014
index:	0	1	2	3	4

Figura 1.5

Applichiamo questo ragionamento all'array della nostra lista della spesa. Il nostro array di esempio inizia dall'indirizzo di memoria 1010. Quindi, se chiedessimo al computer di leggere il valore dell'indice `3`, seguirebbe il seguente processo.

1. L'array inizia con l'indice `0`, che si trova all'indirizzo di memoria 1010.
2. L'indice `3` sarà esattamente tre posizioni dopo l'indice `0`.
3. Di conseguenza, l'indice `3` si troverebbe nell'indirizzo di memoria 1013, poiché 1010 + 3 fa 1013.

Una volta che il computer sa che l'indice `3` si trova all'indirizzo di memoria 1013, può saltare direttamente lì e scoprire che contiene il valore `"dates"`.

Leggere da un array, quindi, è un'operazione efficiente, poiché il computer può leggere qualsiasi indice saltando a qualsiasi indirizzo di memoria in un unico passaggio. Anche se ho descritto il processo del computer suddividendolo in tre parti, attualmente ci stiamo concentrando sul fatto che il computer può saltare a un determinato indirizzo di memoria. Nei prossimi capitoli scopriremo quali sono i passaggi sui quali vale la pena di concentrarsi.

Naturalmente, un'operazione che richiede un solo passaggio è molto veloce. Oltre a essere una struttura dati fondamentale, gli array sono anche molto potenti, perché possiamo leggerli a grande velocità.
Ora, che cosa accadrebbe se, invece di chiedere al computer quale valore è contenuto nell'indice 3, capovolgessimo la domanda e chiedessimo a quale indice possiamo trovare `"dates"`? Questa è l'operazione di ricerca che esploreremo ora.

Ricerca

Come ho affermato in precedenza, *cercare* in un array significa verificare se un determinato valore esiste all'interno di un array e, in tal caso, a quale indice si trova.
In un certo senso, la ricerca è il contrario della lettura. Leggere significa fornire al computer un *indice* e chiedergli di restituire il valore in esso contenuto. Cercare, al contrario, significa fornire al computer un *valore* e chiedergli di restituire l'indice della posizione in cui si trova quel valore.
Sebbene queste due operazioni sembrino simili, c'è un'enorme differenza tra loro, in termini di efficienza. La lettura da un indice è veloce, poiché il computer può saltare immediatamente a qualsiasi indice e scoprire il valore in esso contenuto. La ricerca, al contrario, è noiosa, poiché il computer non ha modo di passare a un determinato valore.
Questo è un fatto importante: un computer ha accesso immediato a tutti i suoi indirizzi di memoria, ma non ha idea di quali *valori* siano contenuti in ciascun indirizzo di memoria.
Prendiamo per esempio il nostro array di frutta e verdura. Il computer non può vedere immediatamente il contenuto effettivo di ciascuna cella. Per il computer, l'array somiglia a quanto rappresentato nella Figura 1.6.

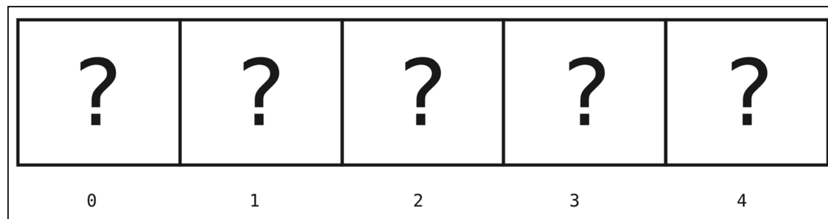


Figura 1.6

Per cercare un determinato frutto all'interno dell'array, il computer non ha altra scelta che ispezionare ogni cella, una alla volta.
Le Figure da 1.7 a 1.10 mostrano il processo che il computer utilizzerebbe per cercare `"dates"` all'interno del nostro array.
Innanzitutto, il computer controlla all'indice `0`:

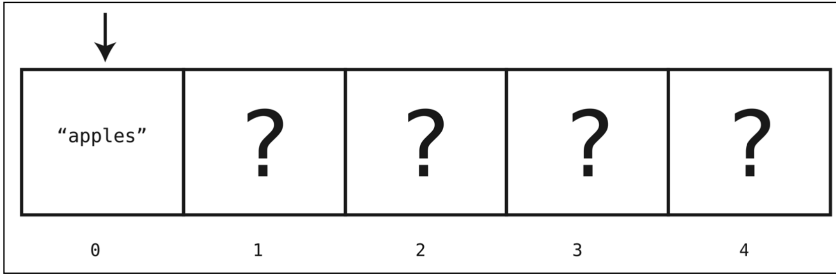


Figura 1.7

Poiché il valore dell'indice `0` è `"apples"` e non `"dates"` (che stiamo cercando), il computer passa all'indice successivo, come mostrato nella Figura 1.8.

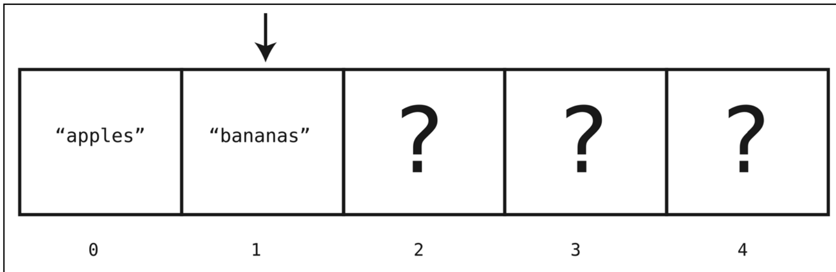


Figura 1.8

Poiché neanche l'indice `1` contiene `"dates"`, il computer passa all'indice `2` (Figura 1.9).

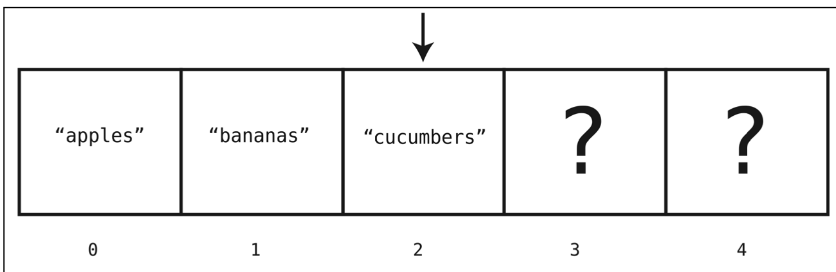


Figura 1.9

Ancora una volta siamo sfortunati, quindi il computer passa alla cella successiva (Figura 1.10).

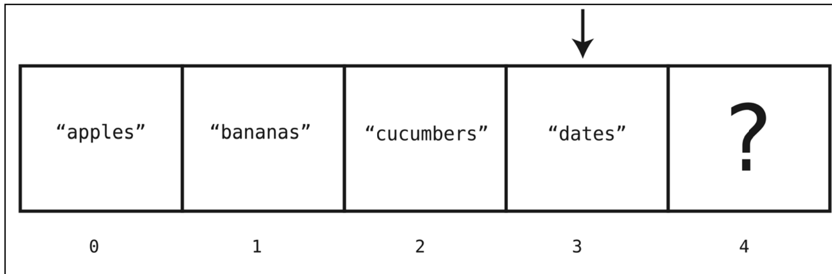


Figura 1.10

Aha! Abbiamo trovato `"dates"` e ora sappiamo che si trova all'indice `3`. A questo punto, il computer non ha bisogno di passare alla cella successiva dell'array, poiché ha trovato ciò che gli abbiamo chiesto di cercare.

In questo esempio, poiché il computer ha dovuto controllare quattro celle prima di trovare il valore che stavamo cercando, diciamo che questa particolare operazione ha richiesto un totale di quattro passaggi.

Nel Capitolo 2 imparerete un altro modo per eseguire la ricerca in un array, ma questa operazione di ricerca di base, in cui il computer controlla ogni cella, una alla volta, è nota come *ricerca lineare*.

Ora, qual è il numero *massimo* di passaggi che un computer potrebbe dover eseguire per condurre una ricerca lineare su un array?

Se il valore che stiamo cercando si trova nella cella finale dell'array (come `"elderberries"`), il computer finirebbe per esaminare *ogni* cella dell'array fino a trovare, finalmente, il valore che sta cercando. E se il valore che stiamo cercando non si trova affatto nell'array, anche in questo caso il computer dovrebbe cercare in ogni cella, in modo da assicurarsi che il valore non esista all'interno dell'array.

Il risultato è che per un array di 5 celle, il numero massimo di passaggi necessari per condurre una ricerca lineare è 5. Per un array di 500 celle, il numero massimo di passaggi necessari per la ricerca lineare è 500.

Un altro modo per dirlo è che per N celle in un array, la ricerca lineare richiederebbe un massimo di N passaggi. In questo contesto, N è semplicemente una variabile che può essere sostituita da qualsiasi numero.

In ogni caso, è evidente che la ricerca è meno efficiente della lettura, poiché la ricerca può richiedere molti passaggi, mentre la lettura richiede sempre un solo passaggio, indipendentemente dalle dimensioni dell'array.

Ora analizzeremo l'operazione di inserimento.

Inserimento

L'efficienza nell'inserimento di un nuovo dato in un array dipende da *dove* lo state inserendo all'interno dell'array.

Diciamo che vogliamo aggiungere `"figs"` alla fine della nostra lista della spesa. Tale inserimento richiede un solo passaggio.

Ciò è vero per un altro fatto relativo ai computer: quando alloca un array, il computer tiene sempre traccia delle dimensioni dell'array stesso.

Se a questo aggiungiamo il fatto che il computer sa anche a quale indirizzo di memoria parte l'array, calcolare l'indirizzo di memoria dell'ultimo elemento dell'array è un gioco da ragazzi: se l'array inizia all'indirizzo di memoria 1010 e le sue dimensioni sono di 5 elementi, questo significa che il suo indirizzo di memoria finale è 1014. Quindi inserire un elemento oltre questo significherebbe aggiungerlo all'indirizzo di memoria successivo, ovvero 1015.
Una volta che il computer ha calcolato in quale indirizzo di memoria inserire il nuovo valore, può farlo in un solo passaggio.
La Figura 1.11 mostra l'inserimento di `"figs"` alla fine dell'array.

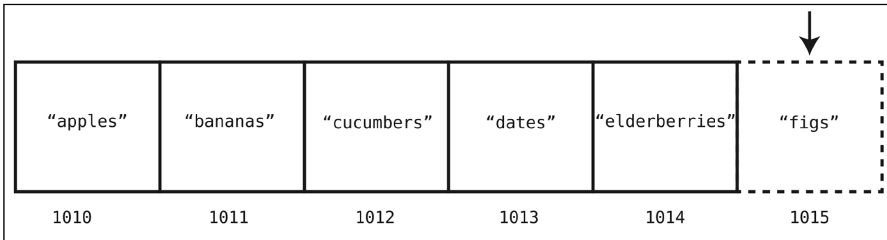


Figura 1.11

Ma c'è un problema. Poiché inizialmente il computer ha allocato solo cinque celle in memoria per l'array e ora stiamo aggiungendo un sesto elemento, il computer potrebbe dover allocare celle aggiuntive per questo array. In molti linguaggi di programmazione, questa operazione è automatica, ma ogni linguaggio la gestisce in modo diverso, quindi non entrerò nei dettagli.
Abbiamo trattato gli inserimenti alla fine di un array, ma inserire un nuovo dato *all'inizio* o *nel bel mezzo* di un array è tutta un'altra faccenda. In questi casi, dobbiamo *spostare* i dati per fare spazio a ciò che stiamo inserendo, e ciò porta a passaggi aggiuntivi.
Per esempio, supponiamo di voler aggiungere `"figs"` all'indice `2` all'interno dell'array. Osservate la Figura 1.12.

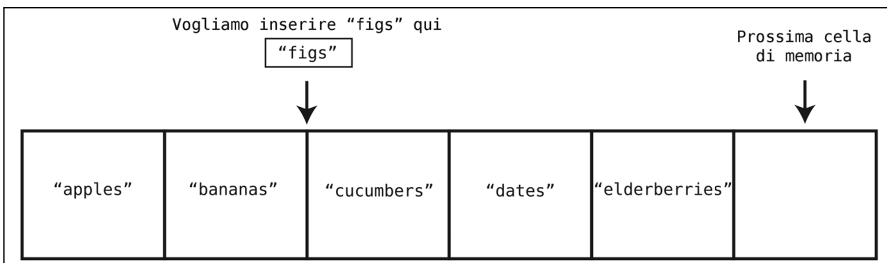
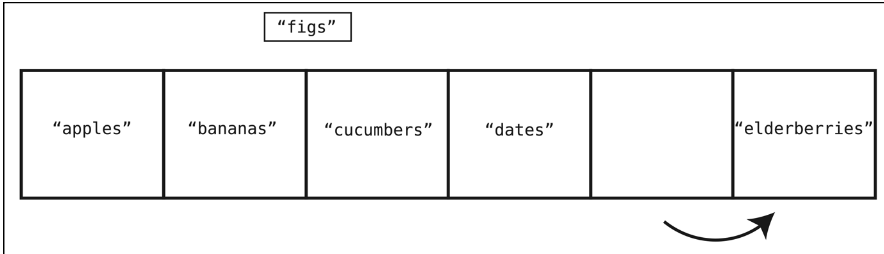


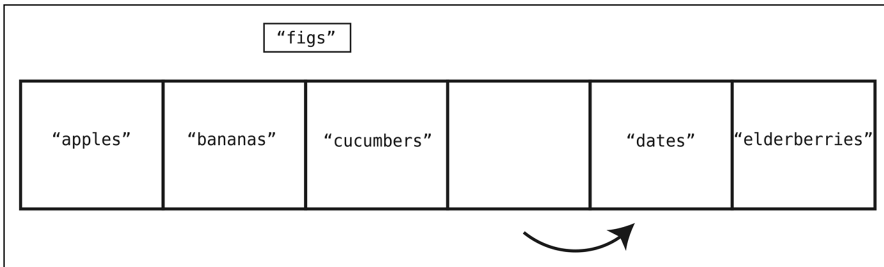
Figura 1.12

Per farlo, dobbiamo spostare `"cucumbers"`, `"dates"` ed `"elderberries"` verso destra per fare spazio a `"figs"`. Ciò richiede più passaggi, poiché dobbiamo prima spostare `"elderberries"` di una cella a destra per fare spazio per spostare `"dates"`. Poi bisogna spostare `"dates"` per fare spazio a `"cucumbers"`. Proviamo a svolgere queste operazioni.

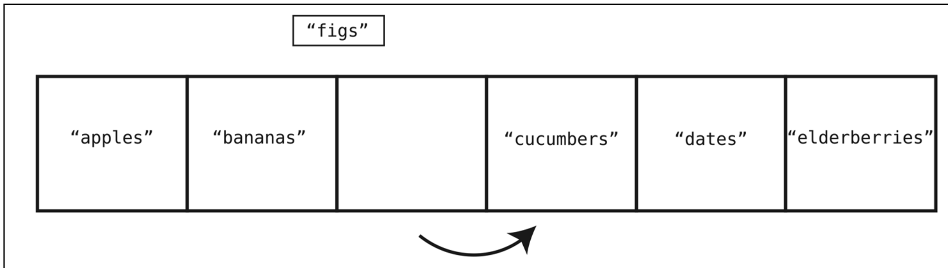
Passaggio 1: spostiamo `"elderberries"` a destra:



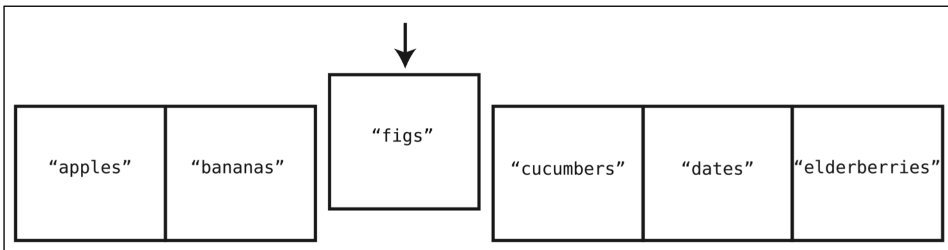
Passaggio 2: ora spostiamo `"dates"` a destra:



Passaggio 3: ora spostiamo `"cucumbers"` a destra:



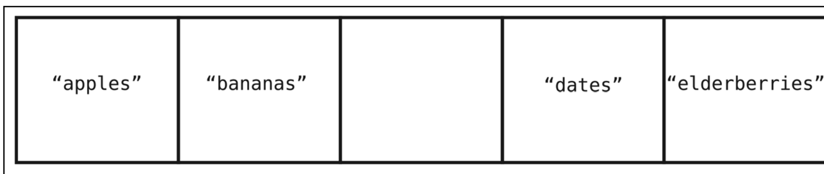
Passaggio 4 – Infine, possiamo inserire `"figs"` nell'indice 2:



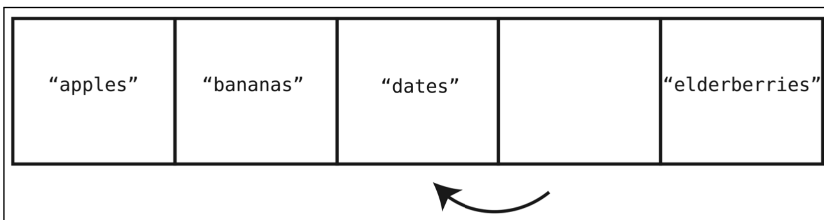
Notate che nell'esempio precedente l'inserimento ha richiesto quattro passaggi. Tre di essi prevedevano lo spostamento dei dati a destra, mentre uno solo prevedeva l'effettivo inserimento del nuovo valore.
Lo scenario peggiore per l'inserimento in un array, ovvero lo scenario in cui l'inserimento richiede il maggior numero di passaggi, è quando inseriamo i dati *all'inizio* dell'array. Questo perché, quando inseriamo un dato all'inizio dell'array, dobbiamo spostare *tutti* gli altri valori di una cella a destra.
Possiamo dire che l'inserimento nello scenario peggiore può richiedere N + 1 *passaggi* per un array contenente N elementi. Questo perché dobbiamo spostare tutti gli N elementi e infine eseguire il passaggio di inserimento vero e proprio.
Ora che abbiamo trattato l'inserimento, passiamo all'ultima operazione riguardante gli array: l'eliminazione.

Eliminazione

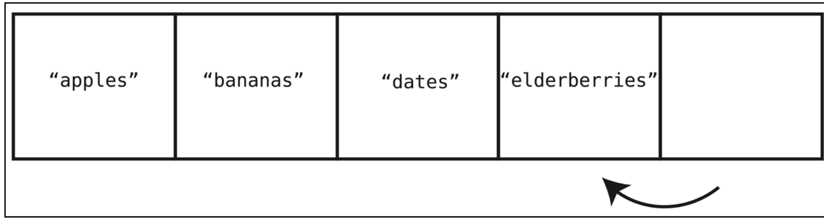
L'eliminazione da un array consiste nella cancellazione del valore situato a un determinato indice.
Torniamo al nostro array di esempio ed eliminiamo il valore nell'indice 2. Nel nostro esempio, questo valore è `"cucumbers"`.
Passaggio 1: eliminiamo `"cucumbers"` dall'array:



Anche se tecnicamente l'effettiva eliminazione di `"cucumbers"` ha richiesto un solo passaggio, ora abbiamo un problema: abbiamo una cella vuota proprio in mezzo al nostro array. Un array non è efficace se ospita spazi vuoti, quindi per risolvere questo problema dobbiamo spostare `"dates"` ed `"elderberries"` a sinistra. Ciò significa che il nostro processo di eliminazione richiede alcuni passaggi aggiuntivi.
Passaggio 2 – Spostiamo `"dates"` a sinistra:



Passaggio 3 – Spostiamo `"elderberries"` a sinistra:



Il risultato è che per questa eliminazione l'intera operazione ha richiesto tre passaggi. Il primo prevedeva l'effettiva eliminazione, mentre gli altri due prevedevano lo spostamento dei dati per chiudere lo spazio vuoto.
Come l'inserimento, lo scenario peggiore dell'eliminazione di un elemento consiste nell'eliminazione del primo elemento dell'array. Questo perché l'indice 0 diventerebbe vuoto e dovremmo spostare *tutti* gli elementi rimanenti a sinistra, per colmare il vuoto. Per un array di 5 elementi, spenderemmo 1 passaggio per eliminare il primo elemento e 4 per spostare i 4 elementi rimanenti. Per un array di 500 elementi, spenderemmo 1 passaggio per eliminare il primo elemento e 499 per spostare i dati rimanenti. Possiamo quindi dire che per un array contenente N elementi, il numero massimo di passaggi necessari per l'eliminazione è N passaggi.
E così abbiamo analizzato la complessità, in termini di tempi, della nostra prima struttura dati. Ora che avete imparato ad analizzare l'efficienza di una struttura dati, potrete scoprire che strutture dati differenti hanno efficienze differenti. Questo concetto è fondamentale, perché la scelta della struttura dati corretta per il vostro codice può avere gravi conseguenze sulle prestazioni del vostro software.
La struttura dati successiva, il *set* (l'insieme) a prima vista sembra molto simile all'array. Tuttavia, vedrete che le operazioni eseguite su array e su set hanno efficienze molto differenti.

Il set: come una singola regola può influire sull'efficienza

Esploriamo quest'altra struttura di dati: il *set*. Un set è una struttura dati che *non consente* di contenere valori duplicati.
Vi sono vari tipi di set, ma in questa discussione parlerò di un *set basato su array*. Questo set funziona proprio come un array: è un semplice elenco di valori. L'unica differenza tra questo set e un array classico è che il set non consente mai l'inserimento di valori duplicati.
Per esempio, se avessi il set `["a", "b", "c"]` e provassi ad aggiungere un'altra `"b"`, il computer non lo consentirebbe, poiché una `"b"` esiste già all'interno del set.
I set sono utili quando è necessario assicurarsi di non avere dati duplicati.
Per esempio, se state creando una rubrica online, non volete che lo stesso numero di telefono venga visualizzato due volte. In effetti, attualmente ho un problema con l'elenco telefonico locale: il numero di telefono di casa mia non è elencato solo per me, ma anche (erroneamente) per la famiglia Zirkind (sì, questa è una storia vera). È fastidioso

ricevere telefonate e messaggi da persone che cercano gli Zirkin. Del resto, sono sicuro che anche gli Zirkin si stiano chiedendo perché nessuno li chiami mai. E se provassi a chiamare gli Zirkin per informarli dell'errore, risponderebbe mia moglie, perché ho chiamato sempre il mio numero (ok, quest'ultima parte non è mai accaduta). Se solo il programma che ha prodotto l'elenco avesse utilizzato un set...

In ogni caso, un set basato su array è sempre un array ma con un ulteriore vincolo di divieto di duplicati. Sebbene non consentire la presenza di duplicati sia una funzionalità utile, questo semplice vincolo fa sì che il set vanti *un'efficienza differente* per una delle quattro operazioni principali.

Analizziamo le operazioni di lettura, ricerca, inserimento ed eliminazione nel contesto di un set basato su array.

Leggere da un set è esattamente come leggere da un array: al computer basta un solo passaggio per cercare ciò che è contenuto a un determinato indice. Come ho descritto in precedenza, ciò accade perché il computer può passare a qualsiasi indice del set, poiché può facilmente calcolare e passare al suo indirizzo di memoria.

Inoltre, la ricerca in un set non differisce dalla ricerca in un array: sono necessari fino a N passaggi per cercare un valore all'interno di un set. Anche l'eliminazione è identica tra un set e un array: sono necessari fino a N passaggi per eliminare un valore e spostare i dati a sinistra per colmare il vuoto.

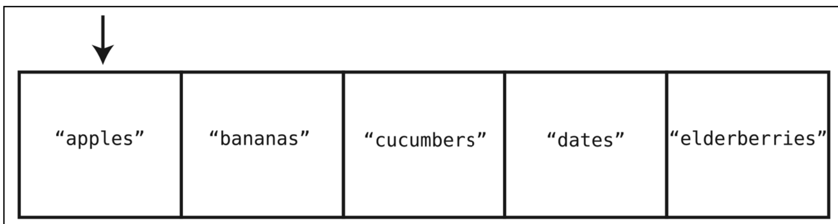
L'inserimento, invece, è il punto in cui array e set divergono. Esploriamo innanzitutto l'inserimento di un valore *alla fine* di un set, che era lo scenario migliore per un array. Abbiamo visto che con un array il computer può inserire un valore alla sua fine in un unico passaggio.

Con un set, al contrario, il computer deve prima determinare che questo valore non esista già nel set, perché è questo che fanno i set: impediscono l'inserimento di dati duplicati. Ora, come farà il computer a garantire che il nuovo dato non sia già contenuto nel set? Un computer non sa immediatamente quali valori sono contenuti nelle celle di un array o set. Per questo motivo, prima dovrà *cercare* nel set per vedere se il valore che vogliamo inserire è già presente. Solo se il set non contiene ancora il nostro nuovo valore il computer consentirà l'inserimento.

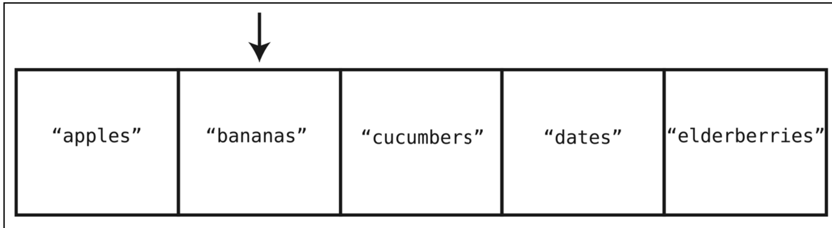
Quindi ogni inserimento in un set richiede *prima una ricerca*.

Vediamo il processo in azione con un esempio. Immaginate che la nostra lista della spesa di prima fosse archiviata come un set, il che sarebbe una scelta corretta, dal momento che, dopotutto, non vogliamo comprare la stessa cosa due volte. Se il nostro set attuale è `["apples, "bananas", "cucumbers", "dates", "elderberries"]` e vogliamo inserire anche `"figs"`, il computer deve eseguire i seguenti passaggi, iniziando a cercare `"figs"`.

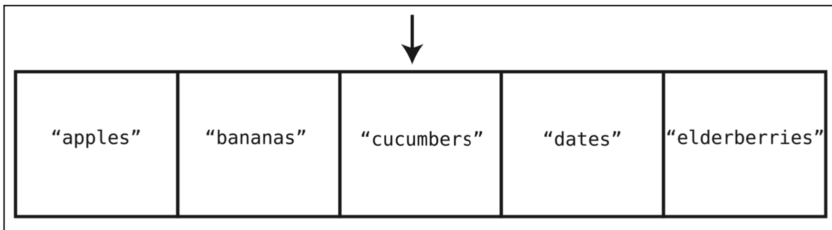
Passaggio 1 – Cercare `"figs"` all'indice 0:



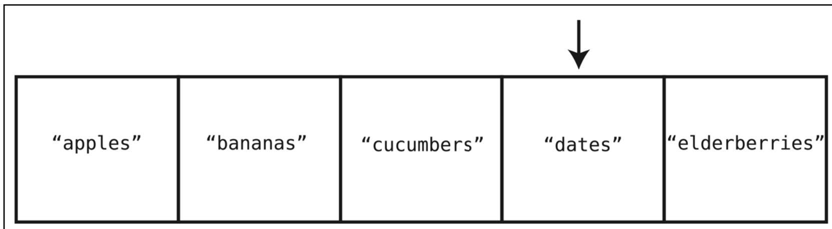
Non c'è, ma potrebbe essere da qualche altra parte nel set. Dobbiamo assicurarci che `"figs"` non esista da nessuna parte prima di poterlo inserire.
Passaggio 2 – Indice di ricerca 1:



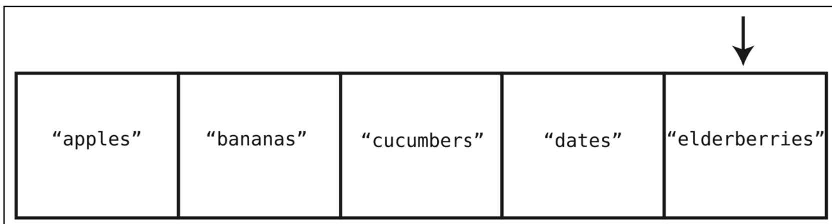
Passaggio 3: indice di ricerca 2:



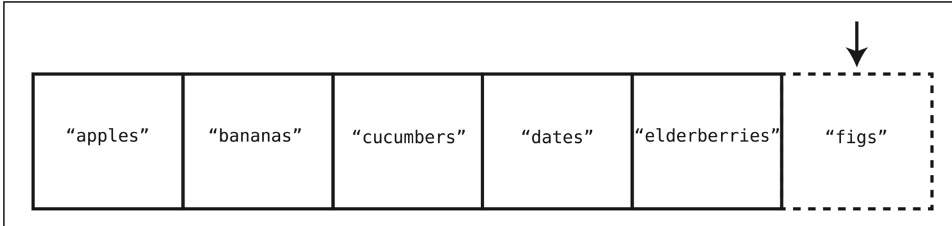
Passaggio 4 – Indice di ricerca 3:



Passaggio 5 – Indice di ricerca 4:



Ora che abbiamo cercato nell'intero set, sappiamo con certezza che non contiene `"figs"`. A questo punto è possibile completare l'inserimento. E questo ci porta al nostro passaggio finale:
Passaggio 6: inserire `"figs"` alla fine del set:



L'inserimento di un valore alla fine di un set è lo scenario migliore, ma abbiamo comunque dovuto eseguire sei passaggi per un set di cinque elementi. Cioè, abbiamo dovuto cercare tutti e cinque gli elementi prima di eseguire il passaggio finale di inserimento.
In altre parole: l'inserimento alla fine di un set richiede fino a N + 1 passaggi per N elementi. Questo perché sono necessari N passaggi di ricerca, per assicurarsi che il valore non esista già all'interno del set, e poi un passaggio per l'effettivo inserimento. In un normale array, tale inserimento richiedeva un totale complessivo di un solo passaggio.
Nello scenario peggiore, quando stiamo inserendo un valore *all'inizio* di un set, il computer deve cercare N celle per assicurarsi che il set non contenga già quel valore, altri N passaggi per spostare tutti i dati a destra e un altro passaggio finale per inserire il nuovo valore. Si tratta di un totale di 2N + 1 passaggi. Al contrario, l'inserimento all'inizio di un normale array richiede solo N + 1 passaggi.
Ora, questo significa che dovreste evitare i set per il solo fatto che l'inserimento è più lento per i set rispetto agli array? Assolutamente no. I set sono importanti quando è necessario garantire che non siano presenti dati duplicati (e, se tutto va bene, un giorno l'elenco telefonico verrà sistemato). Ma quando questa necessità non esiste, un array potrebbe essere preferibile, poiché gli inserimenti negli array sono più efficienti degli inserimenti nei set. È necessario analizzare le esigenze della propria applicazione per decidere quale struttura dati è più adatta.

Conclusioni

L'analisi del numero di passaggi necessari per svolgere un'operazione è fondamentale per comprendere le prestazioni delle strutture dati. Scegliere la giusta struttura dati per il vostro programma può fare la differenza tra la capacità di sopportare un carico pesante e crollare sotto di esso. In questo capitolo avete imparato a utilizzare questa analisi per valutare se un array o un set potrebbe essere la scelta più appropriata per una determinata applicazione.
Ora che avete iniziato a imparare a pensare alla complessità temporale delle strutture dati, possiamo utilizzare la stessa analisi per confrontare algoritmi differenti (anche all'interno della *stessa* struttura dati) per garantire al nostro codice la massima velocità e le migliori prestazioni. Ed è proprio di questo che parlerà il prossimo capitolo.

Esercizi

Gli esercizi seguenti offrono l'opportunità di esercitarsi con gli array. Le soluzioni di questi esercizi si trovano nell'Appendice online.

1. Per un array contenente 100 elementi, specificate il numero di passaggi necessari per le seguenti operazioni.
   a. Lettura.
   b. Ricerca di un valore non contenuto nell'array.
   c. Inserimento all'inizio dell'array.
   d. Inserimento alla fine dell'array.
   e. Eliminazione all'inizio dell'array.
   f. Eliminazione alla fine dell'array.
2. Per un set basato su array contenente 100 elementi, specificate il numero di passaggi necessari per le seguenti operazioni.
   a. Lettura.
   b. Ricerca di un valore non contenuto nel set.
   c. Inserimento di un nuovo valore all'inizio del set.
   d. Inserimento di un nuovo valore alla fine del set.
   e. Eliminazione all'inizio del set.
   f. Eliminazione alla fine del set.
3. Normalmente l'operazione di ricerca in un array cerca la prima istanza di un dato valore. Ma a volte potremmo voler cercare ogni istanza di un dato valore. Per esempio, supponiamo di voler contare quante volte compare il valore `"apple"` si trova all'interno di un array. Quanti passaggi sarebbero necessari per trovare tutti gli `"apple"`? Date la risposta in termini di N.

Capitolo 2

Perché gli algoritmi sono così importanti

In questo capitolo

- **Array ordinati**
- **Ricerca in un array ordinato**
- **Ricerca binaria**
- **Ricerca binaria vs. ricerca lineare**
- **Conclusioni**
- **Esercizi**

Nel capitolo precedente abbiamo esaminato le strutture dati di base e abbiamo visto come la scelta della struttura dati più adatta può influenzare le prestazioni del codice. Perfino due strutture dati che sembrano così simili, come l'array e il set, possono avere livelli di efficienza molto differenti.

In questo capitolo scopriremo che, anche se decidiamo di usare una determinata struttura dati, la più adatta, vi è un altro fattore importante che può influenzare l'efficienza del nostro codice: la scelta corretta dell'*algoritmo* da utilizzare.

La parola *algoritmo* fa pensare a qualcosa di complesso, ma in realtà non è così. Un algoritmo è semplicemente *un insieme di istruzioni da svolgere per completare un determinato compito*.

Anche un processo semplice come preparare una tazza di cereali è, tecnicamente, un algoritmo, poiché chiede di seguire una serie ben definita di passaggi per svolgere il compito. L'algoritmo di preparazione dei cereali segue questi quattro passaggi (almeno per me).

1. Prendi una ciotola.
2. Versa i cereali nella ciotola.
3. Versa il latte nella ciotola.
4. Immergi un cucchiaio nella ciotola.

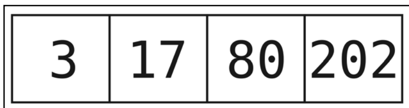
Seguendo questi passaggi in questo preciso ordine, possiamo goderci la nostra colazione.

Tornando al mondo dei computer, un algoritmo si riferisce all'insieme di istruzioni fornite a un computer per portare a termine un determinato compito.

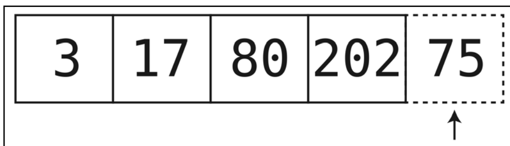
Quando scriviamo del codice, quindi, stiamo creando algoritmi che il computer può leggere ed eseguire.
Possiamo esprimere gli algoritmi anche utilizzando il linguaggio naturale, per definire i dettagli delle istruzioni che intendiamo fornire al computer. In questo libro utilizzerò sia il linguaggio naturale sia il codice per mostrare come funzionano i vari algoritmi.
A volte vi sono due algoritmi differenti che svolgono lo stesso compito. Ne abbiamo visto un esempio all'inizio del Capitolo 1, dove avevamo due approcci differenti per stampare i numeri pari. In quel caso, un algoritmo prevedeva un numero di passaggi doppio rispetto all'altro.
In questo capitolo incontreremo altri due algoritmi che risolvono lo stesso problema. In questo caso, però, un algoritmo sarà più veloce dell'altro per diversi *ordini di grandezza*.
Per esplorare questi nuovi algoritmi, dovremo considerare una nuova struttura dati.

Array ordinati

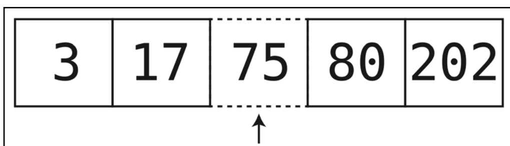
Un *array ordinato* è quasi identico a un array classico (che abbiamo visto nel capitolo precedente); l'unica differenza è che gli array ordinati richiedono che i valori siano sempre mantenuti, come avrete certamente indovinato, *in ordine*: ogni volta che viene aggiunto un nuovo valore, deve essere inserito nella cella corretta, in modo che i valori contenuti nell'array rimangano ordinati.
Per esempio, prendiamo l'array `[3, 17, 80, 202]`:



Supponiamo di voler inserire nell'array il valore `75`. Se questo fosse un array classico, potremmo inserire il `75` alla fine, così:



Come abbiamo visto nel capitolo precedente, il computer può realizzare l'operazione in un unico passaggio.
Ma trattandosi di un *array ordinato*, non avremmo altra scelta che inserire il `75` nel posto giusto, in modo che i valori rimangano in ordine crescente:



Ora, questo è più facile a dirsi che a farsi. Il computer non può semplicemente inserire il `75` nel punto giusto in un unico passaggio, perché deve prima *trovarlo*, il punto giusto, e poi spostare tutti gli altri valori per fargli spazio. Analizziamo questo processo passaggio per passaggio.

Ricominciamo con il nostro array ordinato originale:

3	17	80	202

Passaggio 1 – Controlliamo il valore all'indice 0 (zero), per determinare se il valore che vogliamo inserire, `75`, deve andare alla sua sinistra o alla sua destra:

3	17	80	202
↑			

Poiché `75` è maggiore di `3`, sappiamo che `75` verrà inserito da qualche parte alla sua destra. Tuttavia, non sappiamo ancora esattamente in quale cella inserirlo, quindi dobbiamo controllare la cella successiva.

Chiameremo questo tipo di passaggio *confronto*: confrontiamo il valore che stiamo inserendo con un numero già presente nell'array ordinato.

Passaggio 2 – Controlliamo il valore contenuto nella cella successiva:

3	17	80	202
	↑		

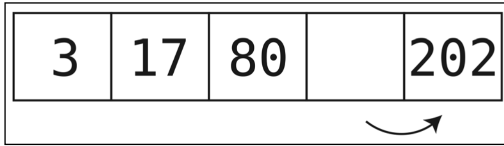
Poiché `75` è maggiore di `17`, dobbiamo andare avanti.

Passaggio 3 – Controlliamo il valore contenuto nella cella successiva:

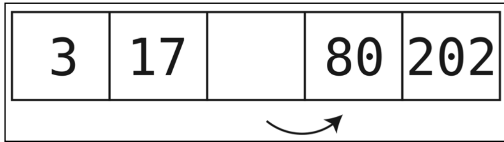
3	17	80	202
		↑	

Abbiamo incontrato il valore `80`, che è *maggiore* del `75` che vogliamo inserire. Poiché abbiamo raggiunto il primo valore maggiore di `75`, possiamo concludere che il `75` deve essere posizionato immediatamente a sinistra di questo `80` per mantenere l'ordine di questo array ordinato. Per farlo, dobbiamo spostare i dati rimanenti per fare spazio al `75`.

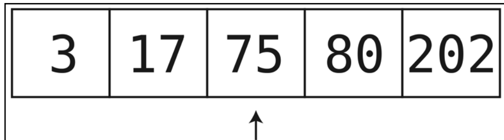
Passaggio 4 – Spostiamo il valore finale a destra:



Passaggio 5 – Spostiamo il penultimo valore a destra:



Passaggio 6 – Finalmente possiamo inserire il 75 nella posizione corretta:



Scopriamo, quindi, che, quando eseguiamo un inserimento in un array ordinato, dobbiamo innanzitutto effettuare sempre una ricerca per determinare il punto corretto per l'inserimento. Questa è una differenza prestazionale tra un array classico e un array ordinato. In questo esempio possiamo vedere che inizialmente c'erano quattro elementi e che l'inserimento ha richiesto sei passaggi. In termini di N, diremmo che per N elementi in un array ordinato, l'inserimento ha richiesto N + 2 passaggi in totale.

È interessante notare che il numero di passaggi per l'inserimento rimane simile, indipendentemente dal punto in cui finisce il nostro nuovo valore nell'array ordinato. Se il nostro valore finisce verso l'inizio dell'array ordinato, abbiamo meno confronti e più spostamenti. Se il nostro valore va inserito verso la fine, avremo più confronti ma meno spostamenti. Il minor numero di passaggi si verifica quando il nuovo valore va proprio alla fine, poiché non sono necessari spostamenti. In questo caso, eseguiamo N passaggi per confrontare il nuovo valore con tutti gli N valori esistenti, più un passaggio per l'inserimento stesso, per un totale di N + 1 passaggi.

Mentre l'inserimento è meno efficiente per un array ordinato che per un array classico, l'array ordinato ha un superpotere segreto quando si parla di ricerche.

Ricerca in un array ordinato

Nel capitolo precedente ho descritto il processo di ricerca di un determinato valore all'interno di un array classico: controlliamo ogni cella, una alla volta, da sinistra a destra, finché non troviamo il valore che stiamo cercando. Come ho scritto, questo processo viene definito ricerca lineare.

Vediamo come differisce la ricerca lineare tra un array classico e uno ordinato.
Supponiamo di avere un array regolare [17, 3, 75, 202, 80]. Se dovessimo cercare il valore 22, inesistente in questo array, dovremmo esaminare ogni singolo elemento, perché il 22 potrebbe potenzialmente trovarsi ovunque nell'array. L'unico caso in cui potremmo interrompere la ricerca prima di raggiungere la fine dell'array è se troviamo il valore che stiamo cercando prima di raggiungere la fine.
Con un array ordinato, al contrario, possiamo interrompere anticipatamente la ricerca anche se il valore non è contenuto nell'array. Diciamo che stiamo cercando il 22 all'interno dell'array ordinato [3, 17, 75, 80, 202]. Possiamo interrompere la ricerca non appena raggiungiamo il 75, poiché è impossibile che il 22 si trovi alla sua destra.
Ecco un'implementazione Python della ricerca lineare su un array ordinato:

```
def linear_search(array, search_value):
    for index, element in enumerate(array):

        if element == search_value:
            return index
        elif element > search_value:
            break

    return None
```

Questo metodo accetta due argomenti: array è l'array ordinato sul quale stiamo eseguendo la ricerca e search_value è il valore che stiamo cercando.
Ecco come utilizzare questa funzione per trovare il 22 nel nostro array di esempio:

```
print(linear_search([3, 17, 75, 80, 202], 22))
```

Come potete vedere, questo metodo linear_search() esegue un'iterazione su ogni elemento dell'array, cercando search_value. La ricerca si interrompe non appena l'elemento sul quale sta iterando risulta essere maggiore di search_value, poiché per definizione sappiamo che in un array ordinato search_value non verrà trovato dopo quella posizione dell'array.
Alla luce di ciò, la ricerca lineare in un array ordinato può compiere alcuni passaggi in meno rispetto a un array classico, in determinate situazioni. Detto questo, se stiamo cercando un valore che, si dà il caso, è il valore finale o che non è presente nell'array, finiremo comunque per controllare ogni singola cella.
A prima vista, quindi, sembra che gli array standard e gli array ordinati non presentino enormi differenze in termini di efficienza o, almeno, non negli scenari peggiori. Per entrambi i tipi di array, se contengono N elementi, la ricerca lineare può richiedere fino a N passaggi.
Ma stiamo per lanciare un algoritmo così potente da far "mangiare la polvere" alla ricerca lineare.
Finora abbiamo dato per scontato che l'unico modo per cercare un valore all'interno di un array ordinato fosse la ricerca lineare. La verità, però, è che la ricerca lineare è solo *uno dei possibili algoritmi* per la ricerca di un valore, non l'unico.
Il grande vantaggio di un array ordinato rispetto a un array classico è che un array ordinato consente di impiegare un algoritmo di ricerca alternativo. Questo algoritmo è chiamato *ricerca binaria* ed è molto, molto più veloce della ricerca lineare.

Ricerca binaria

Probabilmente da bambino avete giocato a questo gioco a indovinello: io penso a un numero compreso tra 1 e 100 e tu devi indovinare il numero in base alle mie indicazioni "Più alto" e "Più basso". Sapete intuitivamente come giocare a questo gioco. Non inizieresti dal numero 1. Probabilmente inizieresti da 50, proprio nel mezzo. Perché? Perché con il 50, vi direi "Più alto" o "Più basso" e voi scartereste automaticamente la metà dei numeri possibili.

Se dite 50 e vi dico "Più alto", poi direte 75, per eliminare la metà dei numeri *rimanenti*. Se dopo il 75 vi dicessi "Più basso", direste 62 o 63. Continuereste a scegliere un punto a metà, per continuare a eliminare la metà dei numeri rimanenti.

Rappresentiamo questo processo per i numeri compreso tra 1 e 10, come mostrato nella Figura 2.1. Questa, in poche parole, è la ricerca binaria.

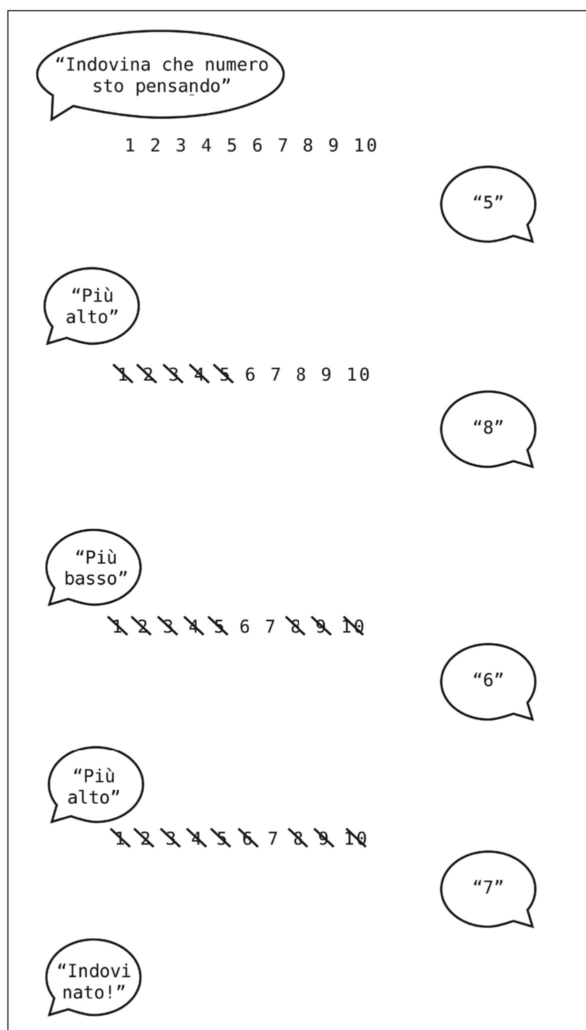
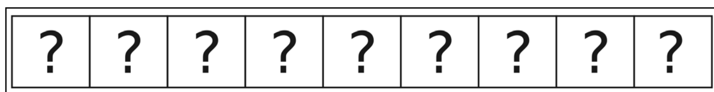


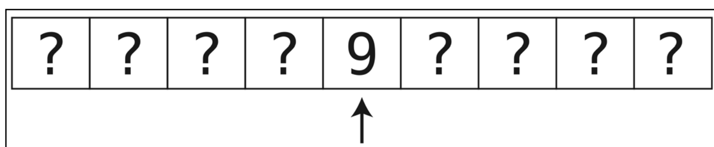
Figura 2.1

Vediamo come applicare la ricerca binaria a un array ordinato. Supponiamo di avere un array ordinato contenente nove elementi. Il computer non sa quale valore contiene ciascuna cella, quindi rappresenteremo l'array in questo modo:

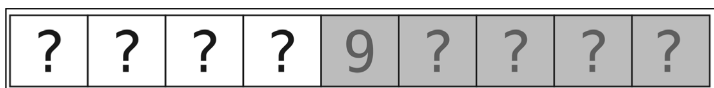


Supponiamo di voler cercare il valore 7 all'interno di questo array ordinato. Ecco come funzionerebbe la ricerca binaria.

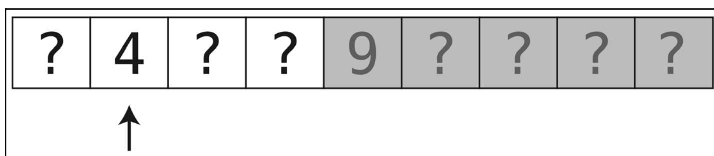
Passaggio 1 – Iniziamo la nostra ricerca dalla cella centrale. Possiamo determinare immediatamente questa cella, poiché possiamo calcolare il suo indice prendendo la lunghezza dell'array e dividendola per 2. Controlliamo il valore presente in questa cella:



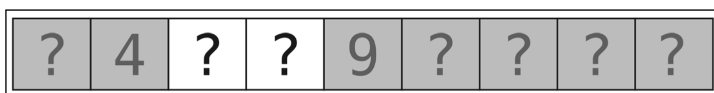
Poiché troviamo il valore 9, possiamo concludere che il 7 è da qualche parte alla sua sinistra. Abbiamo appena eliminato con successo metà delle celle dell'array, ovvero tutte le celle a destra del 9 (e il 9 stesso):



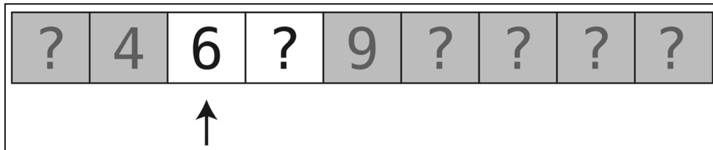
Passaggio 2 – Tra le celle a sinistra del 9, controlliamo il valore più centrale. Ci sono due valori centrali, quindi scegliamo arbitrariamente quello sinistro:



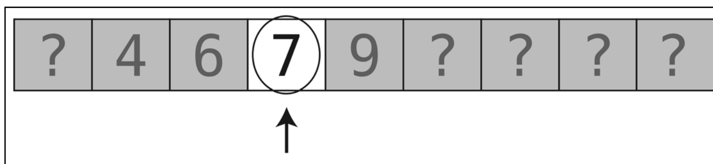
È un 4, quindi il 7 deve essere da qualche parte alla sua destra. Possiamo eliminare il 4 e le celle (una sola) alla sua sinistra:



Passaggio 3 – Ci sono altre due celle in cui può trovarsi il 7. Scegliamo arbitrariamente quella sinistra:



Passaggio 4 – Ispezioniamo l'ultima cella rimanente. E se non c'è, significa che il 7 non è presente all'interno di questo array ordinato.



Abbiamo trovato il 7 in quattro passaggi. In questo esempio si tratta dello stesso numero di passaggi che avrebbe richiesto la ricerca lineare, ma a breve vedremo un altro esempio che ci permetterà di scoprire la potenza della ricerca binaria.
Tenete presente che la ricerca binaria è possibile solo nel caso di un array ordinato. Con un array classico, i valori potrebbero essere in qualsiasi ordine e non sapremmo mai se guardare a sinistra o a destra di un dato valore. Questo è uno dei vantaggi degli array ordinati: abbiamo la possibilità di eseguire una ricerca binaria.

Implementazione della ricerca binaria

Ecco un'implementazione della ricerca binaria in Python:

```
def binary_search(array, search_value):
    lower_bound = 0
    upper_bound = len(array) - 1

    while lower_bound <= upper_bound:
        midpoint = (upper_bound + lower_bound) // 2
        value_at_midpoint = array[midpoint]

        if search_value == value_at_midpoint:
            return midpoint
        elif search_value < value_at_midpoint:
            upper_bound = midpoint - 1
        elif search_value > value_at_midpoint:
            lower_bound = midpoint + 1

    return None
```

Analizziamola. Come il metodo linear_search(), anche binary_search() accetta come argomenti array e search_value.
Ecco un esempio di come richiamare questo metodo:

```
print(binary_search([3, 17, 75, 80, 202], 22))
```

Il metodo stabilisce innanzitutto l'intervallo di indici in cui potrebbe trovarsi search_value. Lo facciamo con il seguente codice:

```
lower_bound = 0
upper_bound = len(array) - 1
```

Poiché quando si avvia la ricerca, search_value potrebbe trovarsi ovunque all'interno dell'intero array, stabiliamo lower_bound come primo indice e upper_bound come ultimo indice.
L'essenza della ricerca si svolge all'interno del ciclo while:

```
while lower_bound <= upper_bound:
```

Questo ciclo continua a essere eseguito finché abbiamo ancora elementi in cui può trovarsi search_value. Come vedremo tra breve, il nostro algoritmo continuerà a restringere questo intervallo man mano che procediamo. La clausola lower_bound <= upper_bound non sarà più valida una volta esaurito l'intervallo e allora possiamo concludere che search_value non è presente nell'array.
All'interno del ciclo, il nostro codice controlla il valore presente nel punto medio dell'intervallo. Il seguente codice realizza ciò:

```
midpoint = (upper_bound + lower_bound) // 2
value_at_midpoint = array[midpoint]
```

value_at_midpoint è l'elemento trovato al centro dell'intervallo.
Ora, se value_at_midpoint è proprio il search_value che stiamo cercando, abbiamo finito la ricerca, e possiamo restituire l'indice in cui si trova search_value:

```
if search_value == value_at_midpoint:
    return midpoint
```

Se invece search_value è inferiore a value_at_midpoint, significa che search_value può trovarsi da qualche parte *prima* nell'array. Possiamo quindi restringere l'intervallo della ricerca ricollocando upper_bound sull'indice a sinistra del punto medio, poiché search_value non può trovarsi altrove:

```
elif search_value < value_at_midpoint:
    upper_bound = midpoint - 1
```

Al contrario, se search_value è maggiore di value_at_midpoint, significa che search_value può trovarsi solo da qualche parte *a destra* del punto medio, quindi aumentiamo il lower_bound in modo appropriato:

```
elif search_value > value_at_midpoint:
    lower_bound = midpoint + 1
```

Restituiamo `None` anche una volta che l'intervallo si è ristretto a 0 elementi, poiché sappiamo con certezza che `search_value` non è presente all'interno dell'array.

Ricerca binaria vs. ricerca lineare

Con array ordinati di piccole dimensioni, l'algoritmo della ricerca binaria non sembra presentare particolari vantaggi rispetto alla ricerca lineare. Ma vediamo che cosa succede con array più grandi.

Con un array contenente 100 valori, ecco il numero massimo di passaggi necessari per ciascun tipo di ricerca,

- Ricerca lineare: 100 passaggi.
- Ricerca binaria: 7 passaggi.

Con la ricerca lineare, se il valore che stiamo cercando è contenuto nella cella finale o è maggiore del valore contenuto nella cella finale, dobbiamo comunque ispezionare ogni singolo elemento. Per un array di dimensione 100, sarebbero necessari 100 passaggi.

Se utilizziamo la ricerca binaria, invece, ogni ipotesi che effettuiamo elimina la metà delle possibili celle che dovremmo ispezionare. Nella nostra primissima ipotesi, riusciremmo a eliminare ben cinquanta celle.

Consideriamo la cosa in un altro modo e vedremo emergere uno schema.

Con un array di dimensione 3, la ricerca binaria richiederebbe un massimo di 2 passaggi. Se raddoppiamo il numero di celle contenute nell'array (e ne aggiungiamo un'altra per mantenere il numero dispari per semplicità), avremo 7 celle. Per un array di questo tipo, il numero massimo di passaggi per trovare un numero utilizzando la ricerca binaria è 3. Se raddoppiamo nuovamente le celle (e ne aggiungiamo una) in modo che l'array ordinato contenga quindici 15, il numero massimo di passaggi per la ricerca binaria è 4.

Lo schema che emerge è che ogni volta che raddoppiamo le dimensioni dell'array ordinato, il numero di passaggi necessari per la ricerca binaria aumenta solo di un'unità. Ciò ha senso, poiché ogni passaggio elimina dalla ricerca la metà degli elementi.

Questo schema è particolarmente efficiente: ogni volta che raddoppiamo i dati, l'algoritmo di ricerca binaria aggiunge *solo un ulteriore passaggio*.

Vediamo che cosa accadrebbe con la ricerca lineare. Se aveste 3 elementi, avreste bisogno di un massimo di 3 passaggi. Per 7 elementi, avrete bisogno di un massimo di 7 passaggi. Per 100 valori, sarebbero necessari fino a 100 passaggi. Con la ricerca lineare, quindi, ci sono *tanti passaggi quanti sono gli elementi*. Quindi, per la ricerca lineare, ogni volta che *raddoppiamo* le dimensioni dell'array, *raddoppia* anche il numero di passaggi della ricerca. Per la ricerca binaria, al contrario, ogni volta che *raddoppiamo* le dimensioni dell'array, dobbiamo aggiungere *un solo ulteriore passaggio*.

Vediamo come funziona per array più grandi. Con un array di 10.000 elementi, la ricerca lineare può richiedere fino a 10.000 passaggi, mentre la ricerca binaria richiede fino a un massimo di soli 13 passaggi. Per un array di dimensioni pari a un milione di elementi, la ricerca lineare richiederebbe fino a un milione di passaggi, mentre la ricerca binaria richiederebbe solo *20 passaggi*.

Possiamo rappresentare la differenza prestazionale tra la ricerca lineare e quella binaria con il grafico rappresentato nella Figura 2.2.

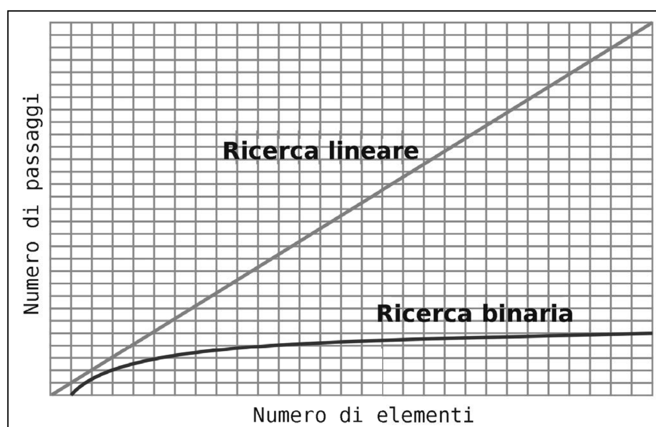


Figura 2.2

Analizzeremo vari altri grafici simili a questo, quindi prendiamoci un momento per capire che cosa significano. L'asse x rappresenta il numero di elementi presenti nell'array. Cioè, mentre ci spostiamo da sinistra a destra, abbiamo a che fare con una quantità crescente di dati.

L'asse y rappresenta il numero di passaggi eseguiti dall'algoritmo. Man mano che saliamo nel grafico, osserviamo un numero maggiore di passaggi.

Se osservate la linea che rappresenta la ricerca lineare, vedrete che, mentre cresce il numero di elementi presenti nell'array, la ricerca lineare richiede un numero di passaggi proporzionalmente crescente. In sostanza, per ogni elemento aggiuntivo nell'array, la ricerca lineare richiede un passaggio aggiuntivo. Questo produce una linea retta diagonale.

Con la ricerca binaria, invece, notiamo che all'aumentare dei dati, i passaggi dell'algoritmo aumentano solo di poco. Ciò ha perfettamente senso, dato quello che sappiamo: ogni raddoppio della quantità di dati aggiungere solo un ulteriore passaggio con la ricerca binaria.

Tenete presente che gli array ordinati non sono più veloci sotto ogni aspetto. Come abbiamo visto, l'inserimento negli array ordinati è più lento rispetto a quello negli array standard. Ma ecco il compromesso: utilizzando un array ordinato, l'inserimento è leggermente più lento ma la ricerca è molto più veloce. Ancora una volta, dovete sempre analizzare l'applicazione che state sviluppando per vedere quale è la soluzione migliore. Il vostro software prevede molti inserimenti? La ricerca sarà una caratteristica significativa dell'app che state creando?

Un piccolo quiz

Trovo che la seguente domanda costringa davvero a comprendere l'efficienza della ricerca binaria. Coprite la risposta, per vedere se avete capito bene.

La domanda: abbiamo detto che per un array ordinato di 100 elementi, la ricerca binaria richiede 7 passaggi. Quanti passaggi richiederebbe la ricerca binaria su un array ordinato di 200 elementi?

La risposta: 8 passaggi.
La risposta intuitiva che sento spesso dire è 14, ma non è corretta. Il bello della ricerca binaria è che ogni controllo elimina la metà degli elementi rimanenti. Pertanto, ogni volta che *raddoppiamo* la quantità di dati, aggiungiamo *un solo* passaggio. Pensatela così: il raddoppio dei dati viene completamente eliminato già con la prima ispezione!
Ma ora che abbiamo aggiunto la ricerca binaria al nostro toolkit, anche l'operazione di inserimento all'interno di un array ordinato può diventare più veloce. L'inserimento richiede una ricerca prima dell'inserimento vero e proprio, ma ora sappiamo che per tale ricerca possiamo impiegare una ricerca binaria. Tuttavia, l'inserimento all'interno di un array ordinato rimane ancora più lento che all'interno di un array regolare, il quale non richiede alcuna ricerca.

Conclusioni

Spesso in programmazione esiste più di un modo per raggiungere un determinato obiettivo, e l'algoritmo scelto può influire notevolmente sulla velocità del codice.
È anche importante rendersi conto che di solito non esiste un'unica struttura dati o algoritmo perfetto per ogni situazione. Per esempio, solo perché gli array ordinati consentono di eseguire ricerche binarie non significa che dovreste sempre utilizzarli. Nelle situazioni in cui non si prevede la necessità di eseguire molte ricerche ma soprattutto di aggiungere sempre nuovi dati, gli array standard potrebbero essere una scelta migliore, perché con loro l'inserimento è più veloce.
Come abbiamo visto, il modo per analizzare due algoritmi alternativi consiste nel contare il numero di passaggi compiuti da ciascuno. Nel prossimo capitolo esamineremo un modo formalizzato per esprimere la complessità temporale delle strutture dati e degli algoritmi. Questo linguaggio comune ci fornirà informazioni più chiare, che ci permetteranno di prendere decisioni migliori su quali algoritmi scegliere.

Esercizi

I seguenti esercizi vi offrono l'opportunità di esercitarvi con la ricerca binaria. Le soluzioni di questi esercizi si trovano nell'Appendice online.

1. Quanti passaggi sarebbero necessari per eseguire una ricerca lineare del numero 8 nell'array ordinato `[2, 4, 6, 8, 10, 12, 13]`?
2. Quanti passaggi richiederebbe la ricerca binaria per l'esempio precedente?
3. Qual è il numero massimo di passaggi necessari per eseguire una ricerca binaria su un array di dimensione 100.000?