Appendice

Soluzioni degli esercizi

Capitolo 1

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo

1. Analizziamo ciascuno dei casi.
   a. La lettura da un array richiede sempre un solo passaggio.
   b. La ricerca di un elemento inesistente all'interno di un array di dimensioni 100 richiede 100 passaggi, poiché il computer deve ispezionare ciascun elemento dell'array prima di determinare che l'elemento non è presente nell'array.
   c. L'inserimento richiede 101 passaggi: 100 spostamenti di ciascun elemento a destra e un passaggio per inserire il nuovo elemento nella parte anteriore dell'array.
   d. L'inserimento alla fine di un array richiede sempre un solo passaggio.
   e. L'eliminazione richiede 100 passaggi: prima il computer cancella il primo elemento e poi sposta i restanti 99 elementi verso sinistra, uno alla volta.
   f. L'eliminazione alla fine di un array richiede sempre un passaggio.
2. Analizziamo ciascuno dei casi.
   a. Come per un array, la lettura dal set basato su array richiede un solo passaggio.
   b. Come per un array, la ricerca nel set basato su array richiede 100 passaggi, poiché ispezioniamo ogni elemento prima di concludere che l'elemento non è presente.
   c. Per inserirlo nel set, dobbiamo prima condurre una ricerca completa per assicurarci che il valore non esista già all'interno del set. Questa ricerca richiederà 100 passaggi. Quindi dobbiamo spostare tutti i 100 elementi a destra per fare spazio al nuovo valore. Infine, inseriamo il nuovo valore all'inizio del set. Si tratta di un totale di 201 passaggi.
   d. Questo inserimento richiede 101 passaggi. Di nuovo, dobbiamo condurre una ricerca completa nel set prima dell'inserimento, un'operazione che richiede 100 passaggi. Concludiamo poi con il passaggio finale di inserimento del nuovo valore alla fine del set.
   e. L'eliminazione richiede 100 passaggi, proprio come per un array classico.
   f. L'eliminazione richiede un solo passaggio, proprio come per array classico.
3. Se l'array contiene N elementi, la ricerca di tutte le istanze della stringa `"apple"` in un array richiede N passaggi. Quando cerchiamo un solo esempio, possiamo interrompere la ricerca non appena lo troviamo. Ma se dobbiamo trovare tutte le sue istanze, non abbiamo altra scelta che ispezionare ogni elemento dell'intero array.

Capitolo 2

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

1. La ricerca lineare su questo array richiederebbe quattro passaggi. Partiamo dall'inizio dell'array e controlliamo ogni elemento da sinistra a destra. Poiché l'`8` è il quarto numero, lo troveremo in quattro passaggi.
2. In questo caso la ricerca binaria richiederebbe un solo passaggio. Iniziamo la ricerca binaria dall'elemento più centrale e, guarda caso, l'`8` è proprio l'elemento più centrale!
3. Per risolvere questo problema, dobbiamo contare quante volte dimezziamo 100.000 fino ad arrivare a 1. Se continuiamo a dividere 100.000 per 2, vediamo che ci vogliono sedici dimezzamenti prima di arrivare a circa 1,53. Se dimezziamo questo valore una diciassettesima volta, otteniamo circa 0,76, che è già inferiore a 1. Ciò suggerisce che, nello scenario peggiore, siano necessari sedici o diciassette passaggi per eseguire una ricerca binaria su 100.000 elementi. Se la vostra risposta è stata sedici o diciassette, significa che avete afferrato bene questo concetto. Ottimo!
   In pratica, la ricerca binaria richiede sedici passaggi. Possiamo vederlo se svolgiamo (ed è noioso) ogni passaggio.
   Iniziamo con 100.000 elementi. Quando si ha a che fare con un numero pari di elementi, non c'è un elemento esatto al centro. Gli elementi centrali sono due e ne scegliamo arbitrariamente uno. Centrandoci su questo elemento, lo abbiamo sostanzialmente eliminato dal conto, lasciando 50.000 elementi da un lato e 49.999 dall'altro. Nel peggiore dei casi, il valore che stiamo cercando è tra 50.000 elementi.
   Eseguiamo qualche altra ricerca. La nostra seconda ricerca ci lascia con 25.000 elementi. La nostra terza ricerca ci lascia 12.500 elementi rimanenti. La quarta ricerca riduce gli elementi a 6.250. La quinta ci lascia con 3.125 elementi.
   Ora, questa è la prima volta che incontriamo un numero dispari di elementi. Qui c'è un singolo elemento centrale e, quando lo cerchiamo, ciascun lato avrà 1.562 elementi. Pertanto, quando facciamo i conti per la nostra prossima ricerca, possiamo sottrarre 1 da 3.125 (poiché abbiamo rimosso l'elemento centrale dall'immagine) e dividere 3.124 per 2. Questo, come abbiamo visto, ci dà 1.562.
   Se proseguiamo con questo schema, ne consegue che sono necessari sedici passaggi per dimezzare 100.000 fino a trovare il valore che stiamo cercando.

Capitolo 3

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

1. Questa funzione opera in un tempo O(1). Possiamo considerare N come l'anno passato alla funzione. Ma indipendentemente dall'anno, l'algoritmo svolge sempre lo stesso numero di passaggi.
2. Questa funzione opera in un tempo O(N). Per N elementi contenuti nell'array, il ciclo verrà eseguito N volte.
3. Questa funzione opera in un tempo O(log N). In questo caso, N è il numero `number_of_grains` che viene passato alla funzione. Il ciclo viene eseguito finché `placed_`

`grains < number_of_grains`, ma `placed_grains` parte da `1` e raddoppia ogni volta che viene eseguito il ciclo. Se, per esempio, `number_of_grains` fosse `256`, continueremmo a raddoppiare `placed_grains` nove volte fino a raggiungere 256, il che significa che il ciclo verrebbe eseguito 8 volte per N uguale a 256. Se `number_of_grains` fosse `512`, il ciclo verrebbe eseguito 9 volte e se `number_of_grains` fosse `1024`, il ciclo verrebbe eseguito 10 volte. Poiché il ciclo viene eseguito una sola volta in più ogni volta che N viene raddoppiato, questo viene considerato O(log N).

4. Questa funzione opera in un tempo O(N). N è il numero di stringhe all'interno dell'array e il ciclo richiede N passaggi.
5. Questa funzione opera in un tempo O(1). Possiamo considerare N come le dimensioni dell'array, ma l'algoritmo richiede un numero fisso di passaggi, indipendentemente dalla grandezza di N. L'algoritmo tiene conto del fatto che N sia pari o dispari, ma in entrambi i casi richiede lo stesso numero di passaggi.

Capitolo 4

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

1. Ecco la tabella completata:

Numero elementi	O(N)	O(log N)	$O(N^2)$
100	100	Circa 7	10.000
2000	2000	Circa 11	4.000.000

2. L'array ha sedici elementi, poiché 16^2 è uguale a 256. Un altro modo per dirlo è che la radice quadrata di 256 è 16.
3. L'algoritmo ha una complessità temporale pari a $O(N^2)$. N, in questo caso, rappresenta le dimensioni dell'array. Abbiamo un ciclo esterno che esegue l'iterazione sull'array N volte e, per ognuna di queste volte, un ciclo interno esegue l'iterazione sullo stesso array N volte. Ciò si traduce in N^2 passaggi.
4. La versione seguente opera in un tempo O(N), poiché iteriamo sull'array una sola volta:

```
def greatest_number(array):
    if not array:
        return None

    greatest_number_so_far = array[0]

    for i in array:
        if i > greatest_number_so_far:
            greatest_number_so_far = i

    return greatest_number_so_far
```

Capitolo 5

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

1. Dopo aver eliminato le costanti, possiamo ridurre l'espressione a O(N).
2. Dopo aver eliminato la costante, possiamo ridurre l'espressione a $O(N^2)$.
3. Questo è un algoritmo che opera in un tempo O(N), dove N è uguale alle dimensioni dell'array. Sebbene esistano due cicli distinti che elaborano gli N elementi, questo è semplicemente 2N, che viene ridotto a O(N) dopo aver eliminato la costante.
4. Questo è un algoritmo che opera in un tempo O(N), dove N è uguale alle dimensioni dell'array. All'interno del ciclo eseguiamo tre passaggi, il che significa che il nostro algoritmo richiede 3N passaggi. Tuttavia, questo si riduce a O(N) quando eliminiamo la costante.
5. Questo è un algoritmo che opera in un tempo $O(N^2)$, dove N è uguale alle dimensioni dell'array. Sebbene eseguiamo il ciclo interno solo per metà del tempo, ciò significa semplicemente che l'algoritmo viene eseguito per $N^2 / 2$ passaggi. Tuttavia, la divisione per 2 è una costante, quindi la esprimiamo semplicemente come $O(N^2)$.

Capitolo 6

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

1. Nella notazione Big O, $2N^2 + 2N + 1$ viene ridotto a $O(N^2)$. Dopo aver eliminato tutte le costanti, ci rimane $N^2 + N$, ma tralasciamo anche N, poiché è di ordine inferiore rispetto a N^2.
2. Poiché log N è di ordine inferiore a N, viene semplicemente ridotto a O(N).
3. La cosa importante da notare qui è che la funzione termina non appena troviamo una coppia la cui somma è `10`. Lo scenario migliore, quindi, è quando la somma dei *primi due numeri* dà `10`, poiché possiamo terminare la funzione prima ancora che inizino i cicli. Uno scenario medio potrebbe essere quando i due numeri si trovano da qualche parte nel mezzo dell'array. Gli scenari peggiori si verificano quando non ci sono due numeri che danno somma `10`, nel qual caso dobbiamo esaurire completamente entrambi i cicli. Lo scenario peggiore opera in un tempo $O(N^2)$, dove N è uguale alle dimensioni dell'array.
4. Questo algoritmo ha un'efficienza pari a O(N), poiché le dimensioni dell'array sono pari a N e il ciclo itera attraverso tutti gli N elementi.
   Questo algoritmo continua il ciclo anche se trova una "X" prima della fine dell'array. Possiamo rendere il codice più efficiente se restituiamo `True` non appena troviamo una "X":

```
def contains_X(string):
    for char in string:
        if char == "X":
            return True

    return False
```

Capitolo 7

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

1. Qui, N è uguale alle dimensioni dell'array. Il nostro ciclo viene eseguito per N / 2 volte, poiché elabora due valori a ogni ciclo. Tuttavia, questo viene espresso come O(N), perché tralasciamo la costante.
2. In questo caso è leggermente complicato definire N, poiché abbiamo a che fare con due array distinti. L'algoritmo elabora ciascun valore una sola volta, quindi potremmo decidere di chiamare N il numero totale di valori di entrambi gli array e la complessità temporale sarebbe O(N). Se vogliamo essere più letterali e dire che un array contiene N elementi e l'altro M elementi, potremmo in alternativa esprimere l'efficienza come O(N + M). Tuttavia, poiché stiamo semplicemente sommando N e M insieme, è più semplice utilizzare N per fare riferimento al numero totale di elementi di dati su entrambi gli array e chiamarlo O(N).
3. Nello scenario peggiore, questo algoritmo viene eseguito (approssimativamente) tante volte quanto il numero di caratteri nell'ago moltiplicato per il numero di caratteri nel pagliaio. Immaginiamo, per esempio, di cercare l'ago `ab` nel pagliaio `aaaaaaaaaab`. Ogni volta che il nostro ciclo esterno raggiunge un nuovo `a`, eseguiamo un ciclo interno alla ricerca di `ab`.
   Poiché l'ago e il pagliaio possono contenere numeri differenti di caratteri, questo opera in un tempo O(N * M).
4. N è uguale alle dimensioni dell'array e la complessità temporale opera in un tempo $O(N^3)$, poiché l'elaborazione prevede tre cicli annidati. In realtà, il ciclo centrale viene eseguito N / 2 volte e il ciclo più interno viene eseguito N / 4 volte, quindi questo è N *(N / 2) *(N / 4), ovvero N^3 / 8 passaggi. Ma tralasciamo la costante, quindi rimane $O(N^3)$.
5. N è uguale alle dimensioni dell'array `resumes`. Poiché a ogni ciclo eliminiamo metà dei curriculum di `resumes`, abbiamo un algoritmo con efficienza temporale O(log N).

Capitolo 8

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

1. La seguente implementazione, prima memorizza i valori del primo array in una tabella hash e poi confronta ciascun valore del secondo array con quella tabella hash:

```
def get_intersection(array1, array2):
    intersection = []
    hash_table = {}

    for value in array1:
        hash_table[value] = True

    for value in array2:
        if hash_table.get(value):
            intersection.append(value)

    return intersection
```

Questo algoritmo ha un'efficienza pari a O(N).

2. La seguente implementazione controlla ogni stringa presente nell'array. Se la stringa non è ancora nella tabella hash, viene aggiunta. Se la stringa è nella tabella hash, significa che è già stata aggiunta in precedenza, il che significa che è duplicata! Questo algoritmo ha una complessità temporale pari a O(N):

```
def find_duplicate(array):
    hash_table = {}

    for value in array:
        if hash_table.get(value):
            return value
        else:
            hash_table[value] = True

    return None
```

3. La seguente implementazione inizia creando una tabella hash con tutti i caratteri che incontriamo nella stringa. Successivamente, iteriamo su ciascun carattere dell'alfabeto e controlliamo se il carattere è contenuto nella nostra tabella hash. Se non c'è, significa che il carattere manca dalla stringa, quindi lo restituiamo:

```
def find_missing_letter(string):
    hash_table = {}

    for char in string:
        hash_table[char] = True

    alphabet = "abcdefghijklmnopqrstuvwxyz"

    for char in alphabet:
        if not hash_table.get(char):
            return char

    return None
```

4. La seguente implementazione inizia iterando su ogni carattere nella stringa. Se il carattere non esiste ancora nella tabella hash, il carattere viene aggiunto alla tabella hash come chiave con il valore `1`, indicando che il carattere è stato trovato. Se il carattere è già presente nella tabella hash, incrementiamo semplicemente il valore di `1`. Quindi, se il carattere `"e"` ha valore `3`, significa che la `"e"` esiste tre volte all'interno della stringa.
   Quindi iteriamo nuovamente sui caratteri e restituiamo il primo carattere che esiste una sola volta all'interno della stringa. Questo algoritmo opera in un tempo O(N):

```
def first_non_duplicate(string):
    hash_table = {}
```

```
    for char in string:
        if hash_table.get(char):
            hash_table[char] += 1
        else:
            hash_table[char] = 1

    for char in string:
        if hash_table.get(char) == 1:
            return char

    return None
```

Capitolo 9

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

1. Presumibilmente vorremmo essere gentili con i chiamanti e rispondere alle loro telefonate nell'ordine in cui sono arrivate. Per questo, utilizzeremmo una coda, che elabora i dati in modo FIFO (First In, First Out).
2. Saremmo in grado di leggere il `4`, che ora è l'elemento in cima allo stack. Questo perché avremo estratto il `6` e il `5`, che in precedenza erano posizionati sopra il `4`.
3. Saremmo in grado di leggere il `3`, che ora è in testa alla coda, dopo aver tolto dalla coda l'`1` e il `2`.
4. Possiamo sfruttare lo stack perché estraiamo ogni elemento nell'ordine inverso rispetto a quello in cui è stato inserito. Quindi per prima cosa inseriremo ogni carattere della stringa nello stack. Quindi li estrarremo uno per uno, aggiungendoli alla fine di una nuova stringa:

   ```
   import stack as stack_module

   def reverse(string):
       stack = stack_module.Stack()
       new_string = ""

       for char in string:
           stack.push(char)

       while stack.read():
           new_string += stack.pop()

       return new_string
   ```

   Lo `stack_module` cui si fa riferimento qui è semplicemente la nostra implementazione `Stack` del Capitolo 9, che abbiamo salvato in un file chiamato `stack.py`.

Capitolo 10

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

1. Il caso base è se `low > high`: ovvero vogliamo interrompere la ricorsione una volta che `low` ha superato `high`. Altrimenti finiremmo per stampare numeri più grandi del numero `high`, fino all'infinito.
2. Avremmo una ricorsione infinita! `factorial(10)` richiama `factorial(8)`, che richiama `factorial(6)`, che richiama `factorial(4)`, che richiama `factorial(2)`, che richiama `factorial(0)`. Poiché il nostro caso base è se `number == 1:`, non raggiungeremo mai il numero `1`, quindi la ricorsione continuerà indefinitamente. `factorial(0)` richiamerà `factorial(-2)` e così via.
3. Diciamo che `low` è `1` e `high` è `10`. Quando richiamiamo `sum(1, 10)`, questo a sua volta restituisce `10 + sum(1, 9)`. Cioè, restituiamo la somma di `10` più qualsiasi cosa sia la somma da 1 a 9. `sum(1, 9)` finisce per richiamare `sum(1, 8)`, che a sua volta richiama `sum(1, 7)` e così via.
   Vogliamo che l'ultima chiamata sia `sum(1, 1)`, in cui vogliamo semplicemente restituire il numero `1`. Questo diventa il nostro caso base:

   ```
   def sum(low, high):
       # Caso base:
       if high == low:
           return low

       return high + sum(low, high - 1)
   ```

4. Questo approccio è simile all'esempio della directory di file:

   ```
   def print_all_items(array):
       for value in array:
           # Se il valore corrente è una "lista" Python, in altre parole, un array:
           if isinstance(value, list):
               print_all_items(value)
           else:
               print(value)
   ```

   Iteriamo su ogni elemento dell'array esterno. Se il valore è esso stesso un array, richiamiamo ricorsivamente la funzione su quel subarray. Altrimenti, è il caso base in cui mostriamo semplicemente il valore sullo schermo.

Capitolo 11

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

1. Richiamiamo la funzione `character_count()`. Il primo passo è far finta che la funzione `character_count()` sia già stata implementata.
   Successivamente, dobbiamo identificare il sottoproblema. Se il nostro problema è l'array `["ab", "c", "def", "ghij"]`, allora il nostro sottoproblema può essere lo stesso

array cui togliamo una stringa. Diciamo, per la precisione, che il nostro sottoproblema è l'array meno la prima stringa, ovvero `["c", "def", "ghij"]`.
Vediamo ora che cosa succede quando applichiamo al sottoproblema la funzione "già implementata". Se dovessimo richiamare `character_count(["c", "def", "ghij"])`, ci restituirebbe il valore `8`, poiché ci sono otto caratteri in totale.
Quindi, per risolvere il nostro problema originale, tutto ciò che dobbiamo fare è aggiungere la lunghezza della prima stringa (`"ab"`) al risultato della chiamata alla funzione `character_count()` sul sottoproblema.
Ecco una possibile implementazione:

```
def character_count(array):
    # Caso base: quando l'array è vuoto:
    if not array:
        return 0

    return len(array[0]) + character_count(array[1:])
```

Tenete presente che il caso base è un array vuoto, nel qual caso ci sono zero caratteri da contare.

2. Innanzitutto, facciamo finta che la funzione `select_even()` esista già. Successivamente, identifichiamo il sottoproblema. Se proviamo a selezionare tutti i numeri pari nell'array dell'esempio `[1, 2, 3, 4, 5]`, potremmo dire che il sottoproblema riguarda tutti i numeri nell'array dopo il primo. Immaginiamo quindi che `select_even([2, 3, 4, 5])` funzioni già e restituisca `[2, 4]`.
   Poiché il primo numero nell'array è `1`, non vogliamo fare altro che restituire `[2, 4]`. Tuttavia, se il primo numero nell'array fosse uno `0`, vorremmo restituire `[2, 4]` con l'aggiunta di `0`.
   Il nostro caso base è un array vuoto. Ecco una possibile implementazione:

```
def select_even(array):
    if not array:
        return []

    if array[0] % 2 == 0:
        return [array[0]] + select_even(array[1:])
    else:
        return select_even(array[1:])
```

3. La definizione di numero triangolare è `n` più il numero precedente della sequenza, dove `n` si riferisce al punto in cui il numero cade nello schema. Per esempio, se stiamo calcolando il settimo numero della sequenza, allora `n` è uguale a `7`. Se il nome della funzione è `triangle()`, possiamo esprimere la cosa semplicemente come `n + triangle(n - 1)`. Il caso base è quando `n` è `1`.

```
def triangle(n):
    if n == 1:
        return 1

    return n + triangle(n - 1)
```

4. Facciamo finta che la funzione, `index_of_x()`, sia già stata implementata. Successivamente, diciamo che il sottoproblema è la nostra stringa meno il suo primo carattere. Per esempio, se la nostra stringa di input è `"hex"`, il sottoproblema è `"ex"`.
   Ora, `index_of_x("ex")` restituirebbe `1`. Per calcolare l'indice della `"x"` per la stringa originale, aggiungeremo `1` a questo risultato, poiché la `"h"` aggiuntiva nella parte anteriore della stringa sposta la `"x"` avanti di un indice. Ecco il nostro codice:

```
def index_of_x(string):
    if string[0] == "x":
        return 0

    return index_of_x(string[1:]) + 1
```

5. Questo esercizio è simile al problema delle scale. Analizziamolo.
   Dalla posizione di partenza abbiamo solo due scelte di movimento. Possiamo spostarci di uno spazio a destra o di uno spazio verso il basso.
   Ciò significa che il numero totale di percorsi minimi unici sarà il numero di percorsi dallo spazio a destra di S più il numero di percorsi dallo spazio sotto S.
   Il numero di percorsi dallo spazio a destra di S è lo stesso del calcolo dei percorsi in una griglia di sei colonne e tre righe, come potete vedere qui:

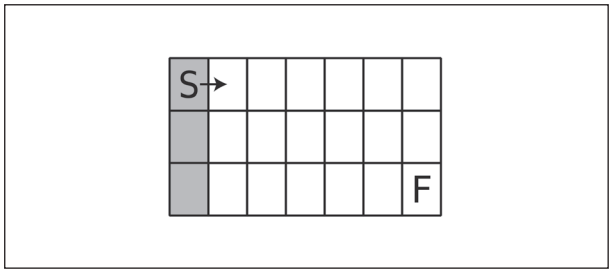


Figura A.1

   Il numero di percorsi dallo spazio sotto la S è l'equivalente dei percorsi in una griglia di sette colonne e due righe:

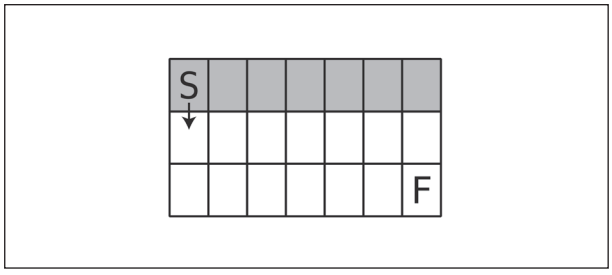


Figura A.2

La ricorsione ci permette di esprimerlo magnificamente:

```
return unique_paths(rows - 1, columns) + unique_paths(rows, columns - 1)
```

Tutto quello che dobbiamo fare, ora, è aggiungere il caso base. I possibili casi base includono quando abbiamo solo una riga o una colonna, poiché in questi casi abbiamo a disposizione un solo percorso.
Ecco la funzione completa:

```
def unique_paths(rows, columns):
    if rows == 1 or columns == 1:
        return 1

    return unique_paths(rows - 1, columns) + unique_paths(rows, columns - 1)
```

Capitolo 12

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

1. Il problema, qui, è che la funzione richiama ricorsivamente se stessa due volte ogni volta che viene eseguita. Facciamo in modo che si richiami una sola volta:

```
def add_until_100(array):
    if not array:
        return 0

    sum_of_remaining_numbers = add_until_100(array[1:])

    if array[0] + sum_of_remaining_numbers > 100:
        return sum_of_remaining_numbers
    else:
        return array[0] + sum_of_remaining_numbers
```

2. Ecco la versione memoizzata:

```
def golomb(n, memo):
    if n == 1:
        return 1

    if n not in memo:
        memo[n] = 1 + golomb(n - golomb(golomb(n - 1, memo), memo), memo)

    return memo[n]
```

3. Per realizzare la memoizzazione, dobbiamo creare una chiave che tenga conto sia del numero di righe sia del numero di colonne. A tal fine, possiamo fare in modo che la chiave si basi sulla riga e anche sulla colonna.

```
def unique_paths(rows, columns, memo):
    if rows == 1 or columns == 1:
        return 1
```

```
    if (rows, columns) not in memo:
        memo[(rows, columns)] = (unique_paths(rows - 1, columns, memo) + \
                unique_paths(rows, columns - 1, memo))

    return memo[(rows, columns)]
```

Notate che, come chiave, usiamo una tupla Python `(rows, columns)` invece di un array`[rows, columns]`. Questo perché Python non consente l'utilizzo di array come chiavi di una tabella hash. In questo libro non abbiamo trattato le tuple, ma in breve si tratta di array immutabili. In altre parole, una volta creata, una tupla non potrà più essere modificata.

Capitolo 13

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

1. Se ordiniamo i numeri, sappiamo che i tre numeri più grandi saranno alla fine dell'array e possiamo semplicemente moltiplicarli. L'ordinamento richiede un tempo O(N log N):

```
def greatest_product_of_3(array):
    array.sort()

    return array[-1] * array[-2] * array[-3]
```

   Questo codice dà per scontato che nell'array vi siano almeno tre valori. È possibile aggiungere del codice per gestire gli array più piccoli.

2. Se preordiniamo l'array, possiamo aspettarci che ogni numero si trovi al proprio indice. Cioè, lo `0` dovrebbe essere all'indice `0`, l'`1` all'indice `1` e così via. Possiamo quindi scorrere l'array cercando un numero che non sia uguale all'indice. Una volta trovato, sappiamo che abbiamo trovato il numero mancante:

```
def find_missing_number(array):
    array.sort()

    for index, num in enumerate(array):
        if num != index:
            return index

    return None
```

   L'ordinamento richiede N log N passaggi e il ciclo successivo richiede N passaggi. Tuttavia, riduciamo l'espressione (N log N) + N a O(N log N), poiché la N aggiunta è di ordine inferiore rispetto a N log N.

3. Questa implementazione utilizza cicli annidati e opera in un tempo $O(N^2)$:

```
def max(array):
    if not array:
        return None
```

```
    for i in array:
        i_is_greatest_number = True

        for j in array:
            if j > i:
                i_is_greatest_number = False

        if i_is_greatest_number:
            return i
```

L'implementazione successiva ordina semplicemente l'array e restituisce l'ultimo numero. L'ordinamento opera in un tempo O(N log N):

```
def max(array):
    if not array:
        return None

    array.sort()

    return array[-1]
```

La nostra prossima e ultima implementazione opera in un tempo O(N), poiché iteriamo una sola volta sull'array:

```
def max(array):
    if not array:
        return None

    greatest_number_so_far = array[0]

    for number in array:
        if number > greatest_number_so_far:
            greatest_number_so_far = number

    return greatest_number_so_far
```

Capitolo 14

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

1. Un modo per farlo è con un semplice ciclo `while`:

```
def print_list(self):
    current_node = self.first_node

    while current_node:
        print(current_node.data)
        current_node = current_node.next_node
```

2. Con una lista a doppio concatenamento, abbiamo accesso immediato agli ultimi nodi e possiamo seguire i collegamenti del "nodo precedente" per accedere ai nodi precedenti. Il seguente codice è fondamentalmente l'inverso dell'esercizio precedente:

```
def reverse_print(self):
    current_node = self.last_node

    while current_node:
        print(current_node.data)
        current_node = current_node.previous_node
```

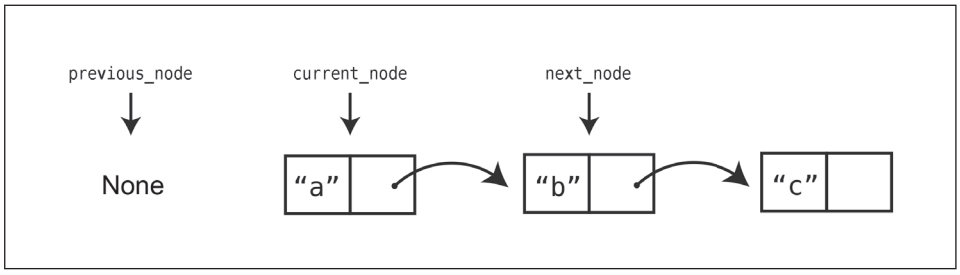
3. Qui utilizziamo un ciclo `while` per spostarci attraverso ciascun nodo. Tuttavia, prima di andare avanti, controlliamo il collegamento del nodo per assicurarci che il nodo successivo esista:

```
def last(self):
    current_node = self.first_node

    while current_node.next_node:
        current_node = current_node.next_node

    return current_node.data
```

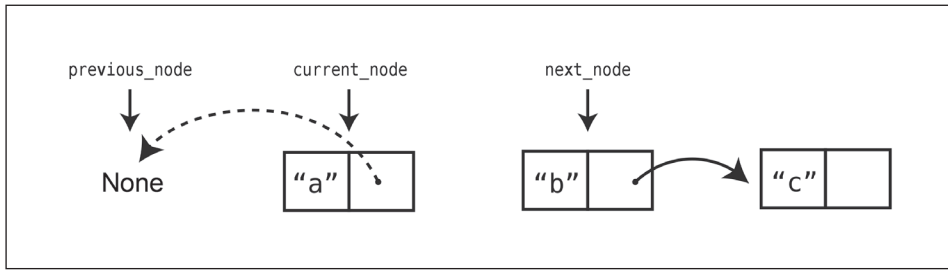
   Per divertimento, ecco un'implementazione alternativa che utilizza la ricorsione:

```
def recursive_last(self, current_node=None):
    if not current_node:
        current_node = self.first_node

    if current_node.next_node:
        return self.recursive_last(current_node.next_node)
    else:
        return current_node.data
```

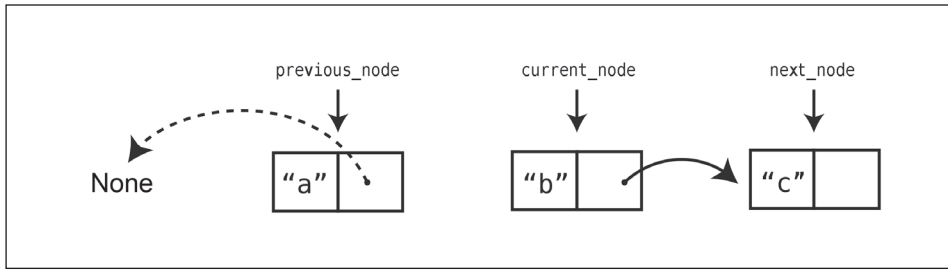
4. Un modo per invertire una lista concatenata consiste nello scorrerla tenendo traccia di tre variabili.
   La variabile primaria è `current_node`, che è il nodo primario sul quale stiamo iterando. Teniamo traccia anche del `next_node`, che è il nodo immediatamente successivo al `current_node`. E teniamo traccia anche del `previous_node`, che è il nodo immediatamente precedente il `current_node`. Osservate la figura seguente:



Notate che quando iniziamo e `current_node` è il primo nodo, `previous_node` punta a `None`; non ci sono nodi prima del primo nodo.
Una volta impostate le tre variabili, procediamo con il nostro algoritmo, che inizia un ciclo.
All'interno del ciclo, prima modifichiamo il collegamento del `current_node` in modo che punti al `previous_node`:



Poi spostiamo tutte le variabili a destra:

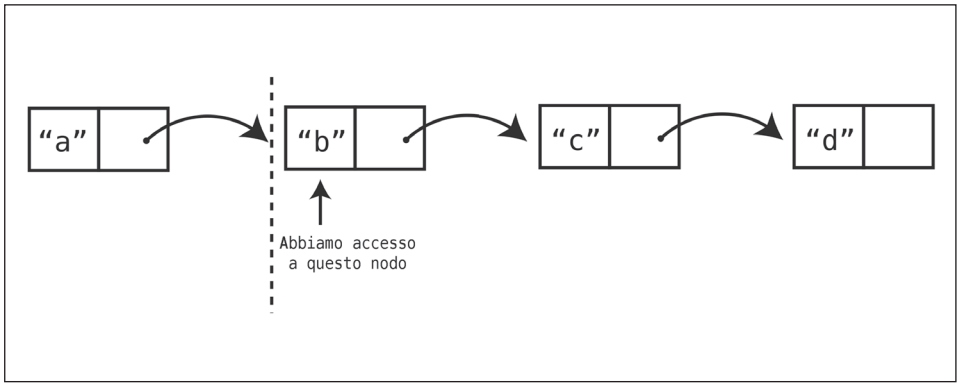


Iniziamo di nuovo il ciclo, ripetendo questo processo di modifica del collegamento del `current_node` in modo che punti al `previous_node`, finché non raggiungiamo la fine della lista. Una volta raggiunta la fine, la lista sarà completamente invertita.
Ecco l'implementazione di questo algoritmo:

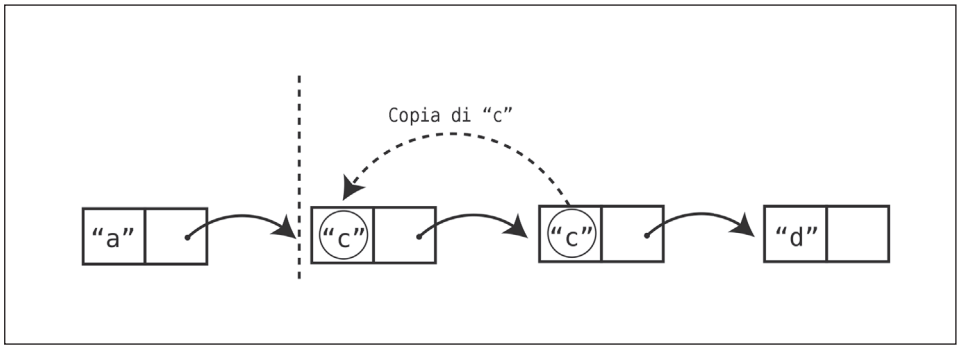
```
def reverse(self):
    previous_node = None
    current_node = self.first_node

    while current_node:
        next_node = current_node.next_node
        current_node.next_node = previous_node

        previous_node = current_node
        current_node = next_node

    self.first_node = previous_node
```

5. Che ci crediate o no, possiamo eliminare un nodo centrale senza avere accesso a nessuno dei nodi che lo precedono.

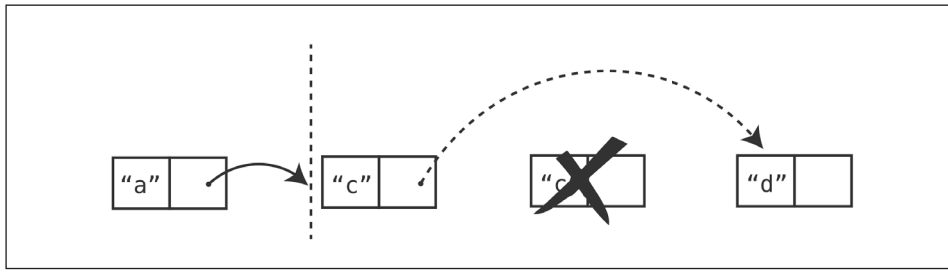
   Di seguito è riportato un esempio. Abbiamo quattro nodi, ma abbiamo accesso solo al nodo `"b"`. Ciò significa che non abbiamo accesso al nodo `"a"`, poiché i collegamenti puntano solo in avanti in una classica lista concatenata. Lo abbiamo indicato utilizzando una linea tratteggiata; cioè non abbiamo accesso a nessun nodo a sinistra della linea tratteggiata:



   Ora, ecco come possiamo eliminare il nodo `"b"` (anche se non abbiamo accesso al nodo `"a"`). Per chiarezza, chiameremo questo nodo il "nodo di accesso", poiché è il primo nodo cui abbiamo accesso.
   Innanzitutto, prendiamo il nodo successivo del nodo di accesso e copiamo i suoi dati nel nodo di accesso, sovrascrivendo i dati del nodo di accesso. Nel nostro esempio, ciò significa copiare la stringa `"c"` nel nostro nodo di accesso:



   Quindi cambiamo il collegamento del nodo di accesso e lo facciamo puntare al nodo che si trova due nodi alla sua destra. Questo elimina di fatto il nodo `"c"` originale:



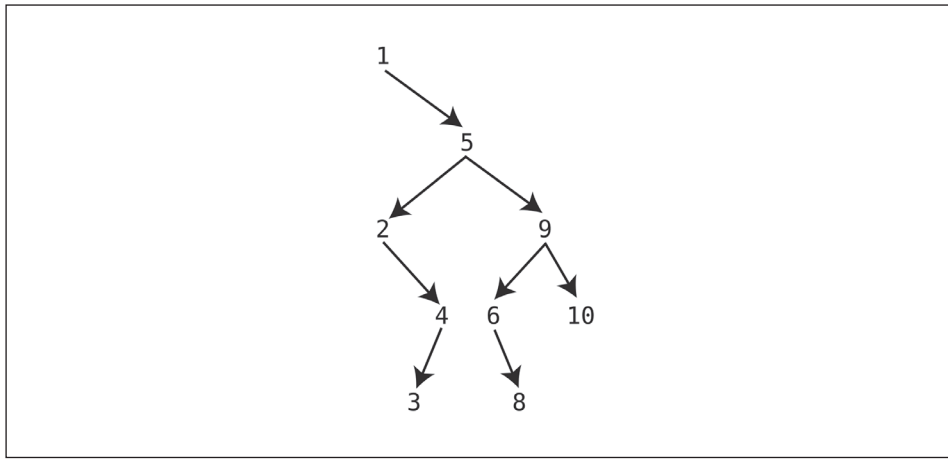
Il codice per farlo è davvero breve:

```
def delete_node(node):
    node.data = node.next_node.data
    node.next_node = node.next_node.next_node
```

Capitolo 15

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

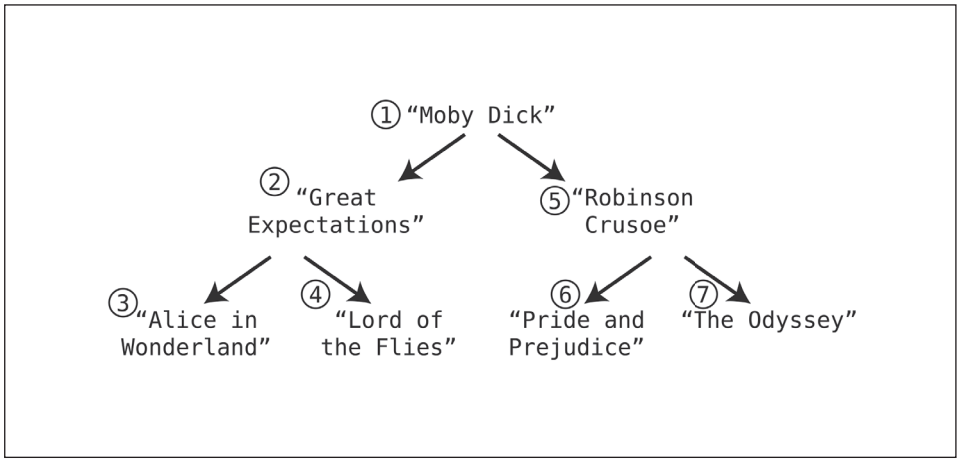
1. L'albero dovrebbe somigliare a quello nella figura. Notate che non è ben bilanciato, poiché il nodo radice ha solo un sottoalbero destro e nessun sottoalbero sinistro:



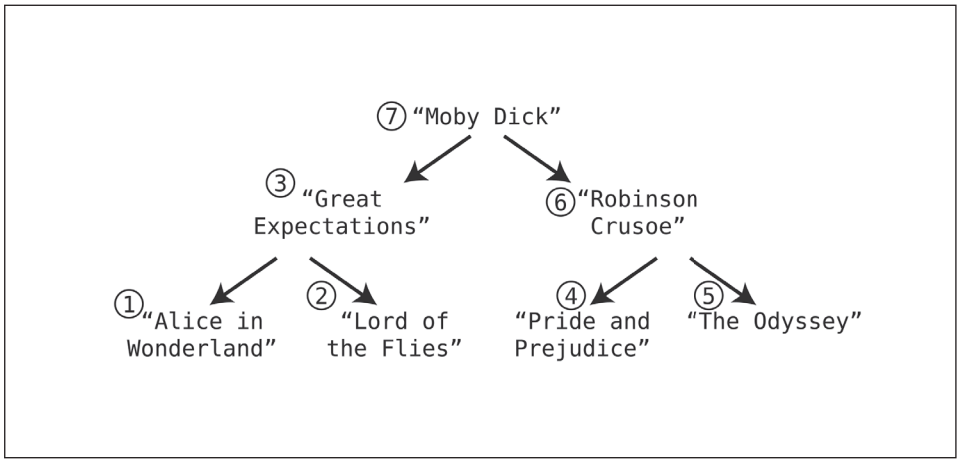
2. La ricerca all'interno di un albero binario di ricerca bilanciato richiede un massimo di circa log(N) passaggi. Quindi, se N è 1.000, la ricerca dovrebbe richiedere un massimo di circa 10 passaggi.
3. Il valore più grande all'interno di un albero binario di ricerca sarà sempre il nodo in basso più a destra. Possiamo trovarlo seguendo ricorsivamente il figlio destro di ciascun nodo fio a raggiungere il fondo:

```
def max(node):
    if node.right_child:
        return max(node.right_child)
    else:
        return node.value
```

4. Ecco l'ordine per l'attraversamento in preordine:



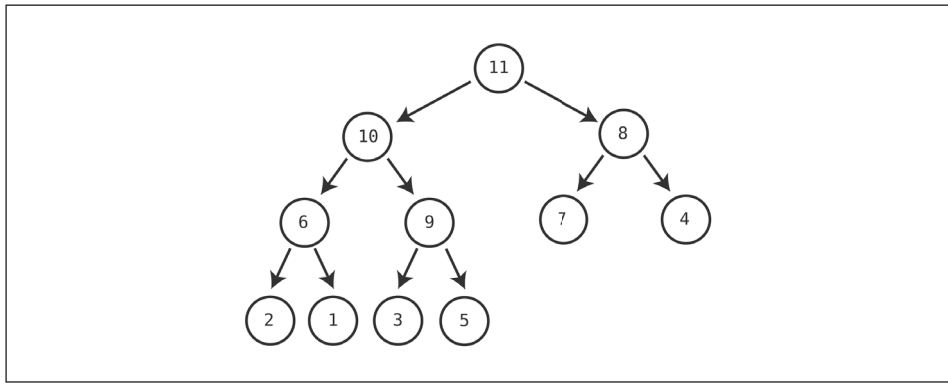
5. Ecco l'ordine per l'attraversamento in postordine:



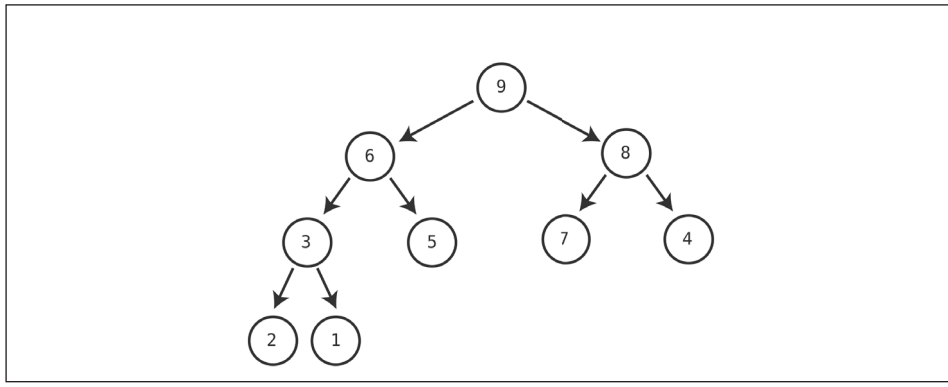
Capitolo 16

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

1. Dopo aver inserito un 11, lo heap avrà il seguente aspetto:



2. Dopo aver eliminato il nodo radice, lo heap avrà il seguente aspetto:



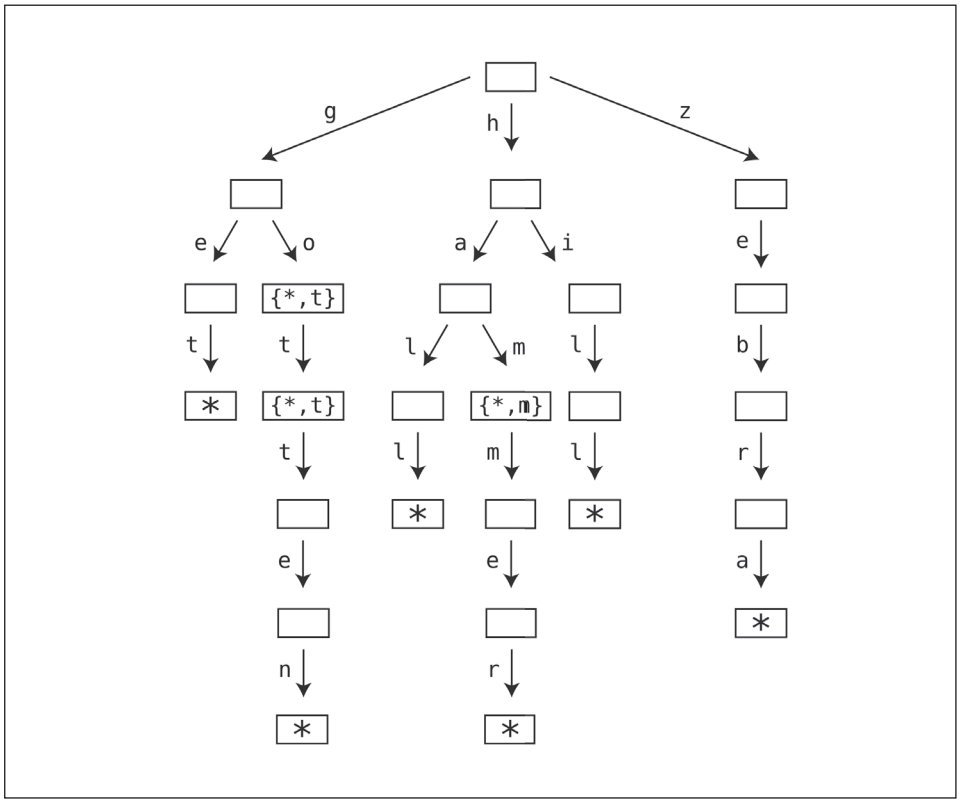
3. I numeri sarebbero in perfetto ordine decrescente. Questo per un max-heap. Per un min-heap, sarebbero in ordine crescente.
   Vi rendete conto di quel che significa? Significa che avete appena scoperto un altro algoritmo di ordinamento!
   L'Heapsort è un algoritmo di ordinamento che inserisce tutti i valori in uno heap e poi li estrae. Come potete vedere da questo esercizio, i valori finiscono sempre in ordine.
   Come il Quicksort, l'Heapsort opera in un tempo O(N log N). Questo perché dobbiamo inserire N valori nello heap e ogni inserimento richiede log N passaggi. Sebbene esistano versioni più elaborate dell'Heapsort che cercano di massimizzarne l'efficienza, questa è l'idea di base.

Capitolo 17

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

1. Questo trie memorizza le parole: `"tag"`, `"tan"`, `"tank"`, `"tap"`, `"today"`, `"total"`, `"we"`, `"well"` e `"went"`.

2. Ecco un trie che memorizza le parole `"get"`, `"go"`, `"got"`, `"gotten"`, `"hall"`, `"ham"`, `"hammer"`, `"hill"` e `"zebra"`:



3. Il codice seguente parte dal nodo del trie e itera su ciascuno dei suoi figli. Per ogni figlio, mostra la chiave e poi si richiama ricorsivamente sul nodo figlio:

```
def traverse(self, node=None):
    current_node = node or self.root

    for key, child_node in current_node.children.items():
        print(key)

        if key != "*":
            self.traverse(child_node)
```

4. La nostra implementazione di correzione automatica è una combinazione delle funzioni `search()` e `collect_all_words()`:

```
def autocorrect(self, word):
    current_node = self.root
    word_found_so_far = ""
```

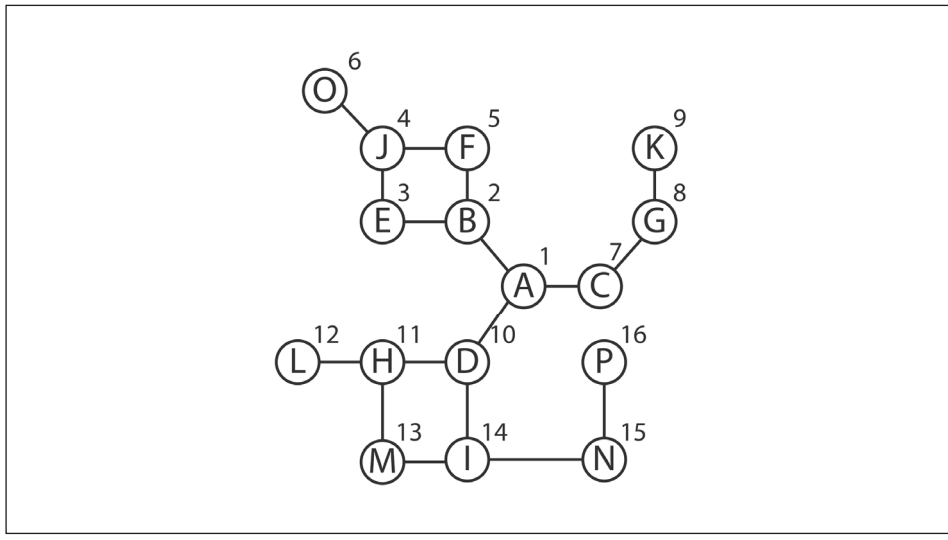
```
        for char in word:
            if current_node.children.get(char):
                word_found_so_far += char
                current_node = current_node.children.get(char)
            else:
                return word_found_so_far + self.collect_all_words([], current_node)[0]

        return word
```

L'approccio di base è che prima cerchiamo nel trie il prefisso più lungo possibile. Quando raggiungiamo un vicolo cieco, invece di restituire semplicemente `None` (come fa la funzione `search()`), richiamiamo `collect_all_words()` sul nodo corrente per raccogliere tutti i suffissi che derivano da quel nodo. Utilizziamo quindi il primo suffisso dell'array e lo concateniamo con il prefisso per suggerire una nuova parola all'utente.

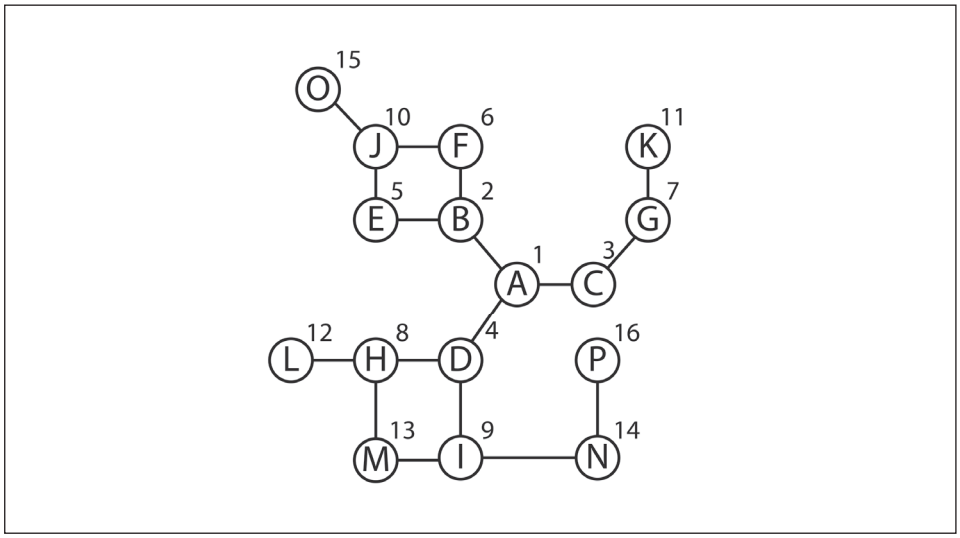
Capitolo 18

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

1. Se un utente sta navigando su `"nails"`, il sito web consiglierà `"nail polish"`, `"needles"`, `"pins"` e `"hammer"`.
2. L'ordine della ricerca in profondità (*depth-first*) sarebbe A-B-E-J-F-O-C-G-K-D-H-L-M-I-N-P, come mostra l'immagine seguente:



3. L'ordine della ricerca in ampiezza (*breadth-first*) sarebbe A-B-C-D-E-F-G-H-I-J-K-L-M-N-O-P, come mostrato nell'immagine seguente:



4. Quella che segue è un'implementazione della ricerca in ampiezza:

```
import queue_implementation

def bfs(starting_vertex, search_value):
    queue = queue_implementation.Queue()
    visited_vertices = {}
    visited_vertices[starting_vertex.value] = True
    queue.enqueue(starting_vertex)

    while queue.read():
        current_vertex = queue.dequeue()
        if current_vertex.value == search_value:
            return current_vertex

        for adjacent_vertex in current_vertex.adjacent_vertices:

            if not visited_vertices.get(adjacent_vertex.value):
                visited_vertices[adjacent_vertex.value] = True
                queue.enqueue(adjacent_vertex)

    return None
```

5. Per trovare il percorso più breve in un *grafo non ponderato*, utilizzeremo la ricerca in ampiezza. La caratteristica principale della ricerca in ampiezza è quella di rimanere il più a lungo possibile vicino al vertice di partenza. Questa funzione servirà come chiave per trovare il percorso più breve.
   Applichiamo questo ragionamento al nostro esempio di social networking. Poiché la ricerca in ampiezza rimane vicino a Idris il più a lungo possibile, finiremo per

trovare Lina prima, attraverso il percorso più breve possibile. Solo più avanti nella ricerca avremmo ritrovato Lina, attraverso percorsi più lunghi. Potremmo addirittura interrompere la ricerca non appena troveremo Lina. La nostra implementazione, che segue, non termina in anticipo, ma potete modificarla perché lo faccia.
Quando visitiamo ciascun vertice per la prima volta, quindi, sappiamo che il vertice corrente fa sempre parte del percorso più breve dal vertice iniziale al vertice che stiamo visitando. Ricordate, con la ricerca in ampiezza il vertice corrente e il vertice che stiamo visitando non sono necessariamente gli stessi.
Per esempio, quando visiteremo Lina per la prima volta, Kamil sarà il vertice corrente. Questo perché nella ricerca in ampiezza arriveremo prima a Lisa attraverso Kamil che attraverso Sasha. Quando visiteremo Lina (attraverso Kamil), possiamo memorizzare in una tabella che il percorso più breve da Idris a Lina sarà attraverso Kamil. Questa tabella è simile alla `cheapest_previous_stopover_city_table` dell'algoritmo di Dijkstra. Infatti, ogni volta che visitiamo un vertice, il percorso più breve da Idris a quel vertice passerà attraverso il vertice corrente. Memorizzeremo tutti questi dati in una tabella chiamata `previous_vertex_table`.
Infine, possiamo utilizzare questi dati per andare a ritroso da Lina a Idris per costruire il percorso più breve tra i due.
Ecco la nostra implementazione:

```
import queue_implementation

def shortest_path(first_vertex, second_vertex, visited_vertices):
    queue = queue_implementation.Queue()
    previous_vertex_table = {}

    visited_vertices[first_vertex.value] = True
    queue.enqueue(first_vertex)

    while queue.read():
        current_vertex = queue.dequeue()

        for adjacent_vertex in current_vertex.adjacent_vertices:
            if not visited_vertices.get(adjacent_vertex.value):
                visited_vertices[adjacent_vertex.value] = True
                queue.enqueue(adjacent_vertex)
                previous_vertex_table[adjacent_vertex.value] = current_vertex.value

    shortest_path = []
    current_vertex_value = second_vertex.value

    while current_vertex_value != first_vertex.value:
        shortest_path.insert(0, current_vertex_value)
        current_vertex_value = previous_vertex_table.get(current_vertex_value)

    shortest_path.insert(0, first_vertex.value)

    return shortest_path
```

Capitolo 19

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

1. La complessità spaziale è $O(N^2)$. Questo perché la funzione crea l'array `collection`, che finirà per contenere N^2 stringhe.
2. Questa implementazione occupa uno spazio O(N), poiché creiamo un `new_array` contenente N elementi.
3. La seguente implementazione utilizza questo algoritmo: scambiamo di posizione il primo elemento con l'ultimo. Quindi scambiamo il secondo elemento con il penultimo. Procediamo quindi a scambiare il terzo elemento con il terzultimo elemento e così via. Poiché tutto viene fatto sul posto e non creiamo nuovi dati, abbiamo una complessità spaziale pari a O(1).

```
def reverse(array):
    i = 0

    while i < len(array) // 2:
        mirror_of_i = len(array) - 1 - i
        array[i], array[mirror_of_i] = array[mirror_of_i], array[i]

        i += 1

    return array
```

   Anche se, dietro le quinte, Python potrebbe creare una variabile temporanea per eseguire ogni scambio, non abbiamo mai più di quel piccolo pezzo di dati memorizzato in qualsiasi momento durante l'esecuzione dell'algoritmo.
4. Ecco la tabella completata:

Versione	Complessità temporale	Complessità spaziale
Versione 1	O(N)	O(N)
Versione 2	O(N)	O(1)
Versione 3	O(N)	O(N)

Tutte e tre le versioni vengono eseguite per tanti passaggi quanti sono i numeri nell'array, quindi la complessità temporale è O(N) per tutte.
La Versione 1 crea un nuovo array per memorizzare i numeri raddoppiati. Questo array avrà la stessa lunghezza dell'array originale, quindi occuperà uno spazio O(N).
La Versione 2 modifica l'array originale sul posto, quindi non occupa spazio aggiuntivo. Questo è espresso come O(1).
Anche la Versione 3 modifica l'array originale in posizione. Tuttavia, poiché la funzione è ricorsiva, lo stack delle chiamate al suo picco avrà N chiamate, occupando O(N) spazio.

Capitolo 20

Queste sono le soluzioni dei quesiti proposti nella sezione Esercizi del capitolo.

1. Possiamo ottimizzare questo algoritmo se ci chiediamo: "Se potessi trovare magicamente l'informazione desiderata in un tempo O(1), potrei rendere più veloce il mio algoritmo?".
   Nello specifico, mentre iteriamo su un array, vorremmo cercare "magicamente" quell'atleta dall'altro array in un tempo O(1). Per farlo, possiamo prima trasformare uno degli array in una tabella hash. Utilizzeremo come chiave il nome completo (ovvero il nome e il cognome) e come valore `True` (o qualsiasi elemento arbitrario). Una volta trasformato un array in questa tabella hash, iteriamo sull'altro array. Ogni volta che incontriamo un atleta, eseguiamo una ricerca O(1) nella tabella hash per vedere se quell'atleta pratica già l'altro sport. Se è così, aggiungiamo quell'atleta al nostro array `multisport_athletes`, che restituiamo alla fine della funzione.
   Ecco il codice per questo approccio:

```
def find_multisport_athletes(array_1, array_2):
    hash_table = {}
    multisport_athletes = []

    for athlete in array_1:
        hash_table[athlete["first_name"] + " " + athlete["last_name"]] = True

    for athlete in array_2:
        if hash_table.get(athlete["first_name"] + " " + athlete["last_name"]):
            multisport_athletes.append(athlete["first_name"] + " " +
                                       athlete["last_name"])

    return multisport_athletes
```

   Questo algoritmo opera in un tempo O(N + M), poiché iteriamo su ciascun set di giocatori una sola volta.

2. Per questo algoritmo, il fatto di generare esempi per individuare uno schema sarà estremamente utile.
   Prendiamo un array che ha sei numeri interi e vediamo che cosa accadrebbe se rimuovessimo ogni volta un numero intero differente.

```
[1, 2, 3, 4, 5, 6]: mancante 0: somma = 21
[0, 2, 3, 4, 5, 6]: mancante 1: somma = 20
[0, 1, 3, 4, 5, 6]: mancante 2: somma = 19
[0, 1, 2, 4, 5, 6]: mancanti 3: somma = 18
[0, 1, 2, 3, 5, 6]: mancante 4: somma = 17
[0, 1, 2, 3, 4, 6]: mancante 5: somma = 16
```

   Hmm. Quando rimuoviamo lo `0`, la somma è `21`. Quando rimuoviamo l'`1`, la somma è `20`. Quando rimuoviamo il `2`, la somma è `19` e così via. Questo è sicuramente uno schema!
   Prima di andare oltre, chiamiamo il `21` in questo caso la "somma totale" (`full_sum`). Questa è la somma dei numeri dell'array quando manca solo lo 0.

Se analizziamo attentamente questi casi, vedremo che la somma di qualsiasi array è inferiore alla somma totale dell'importo del numero mancante. Per esempio, quando manca il 4, la somma è 17, che è quattro meno di 21. E quando manca l'1, la somma è 20, che è 1 meno di 21.
Quindi, possiamo iniziare il nostro algoritmo calcolando qual è la somma totale. Possiamo quindi sottrarre la somma effettiva dalla somma totale e quello sarà il nostro numero mancante.
Ecco il codice per farlo:

```
def find_missing_number(array):
    full_sum = 0

    for num in range(1, len(array) + 1):
        full_sum += num

    current_sum = 0

    for num in array:
        current_sum += num

    return full_sum - current_sum
```

Questo algoritmo opera in un tempo O(N). Sono necessari N passaggi per calcolare la somma totale e poi altri N passaggi per calcolare la somma effettiva. Si tratta di 2N passaggi, che si riducono a O(N).

3. Possiamo rendere questa funzione molto più veloce se utilizziamo un algoritmo greedy. Forse questo non dovrebbe essere una sorpresa, dato che il nostro codice cerca di ottenere il massimo profitto possibile dalle azioni.
   Per ottenere il massimo profitto, vogliamo acquistare alla quotazione più bassa possibile e vendere alla quotazione più alta possibile. Il nostro algoritmo avido inizia assegnando la prima quotazione come `buy_price`. Quindi iteriamo su tutte le quotazioni e non appena troviamo una quotazione più bassa, la rendiamo il nuovo `buy_price`.
   Allo stesso modo, mentre iteriamo sulle quotazioni, controlliamo quanto profitto otterremmo se vendessimo a quella cifra. Questo viene calcolato sottraendo `buy_price` dalla quotazione corrente. In modo avido, salviamo questo profitto nella variabile `greatest_profit`. Mentre iteriamo attraverso tutte le quotazioni, ogni volta che troviamo un profitto maggiore, lo trasformiamo in `greatest_profit`.
   Quando avremo finito di scorrere le quotazioni, `greatest_profit` manterrà il massimo profitto possibile che possiamo ottenere acquistando e vendendo le azioni una volta.
   Ecco il codice del nostro algoritmo:

```
def find_greatest_profit(array):
    buy_price = array[0]
    greatest_profit = 0

    for price in array:
        potential_profit = price - buy_price

        if price < buy_price:
            buy_price = price
```

```
        elif potential_profit > greatest_profit:
            greatest_profit = potential_profit

    return greatest_profit
```

Poiché iteriamo su N prezzi una sola volta, la funzione richiede un tempo O(N). Non solo abbiamo guadagnato un sacco di soldi, ma lo abbiamo fatto velocemente.

4. Questo è un altro algoritmo in cui generare esempi per trovare un modello sarà la chiave per ottimizzarlo.
   Come ho suggerito nell'esercizio, è possibile che il prodotto più grande sia il risultato di numeri negativi. Esaminiamo vari esempi di array e i loro prodotti più grandi formati da due numeri:

```
[-5, -4, -3, 0, 3, 4] -> Prodotto più grande: 20(-5 * -4)
[-9, -2, -1, 2, 3, 7] -> Prodotto più grande: 21(3 * 7)
[-7, -4, -3, 0, 4, 6] -> Prodotto più grande: 28(-7 * -4)
[-6, -5, -1, 2, 3, 9] -> Prodotto più grande: 30(-6 * -5)
[-9, -4, -3, 0, 6, 7] -> Prodotto più grande: 42(6 * 7)
```

   Vedere tutti questi casi può aiutarci a capire che il prodotto più grande può essere formato solo dai due numeri più grandi positivi o dai due numeri più bassi negativi. A questo punto, dovremmo progettare il nostro algoritmo per tenere traccia di questi quattro numeri:

   - Il numero più grande;
   - Il secondo numero più grande;
   - Il numero più piccolo;
   - Il secondo numero più piccolo.

   Possiamo quindi confrontare il prodotto dei due numeri più grandi con il prodotto dei due numeri più piccoli. E il prodotto maggiore sarà anche il prodotto più grande dell'array.
   Ora, come troviamo i due numeri più grandi e i due numeri più piccoli? Se ordinassimo l'array, sarebbe facile. Ma avremmo ancora O(N log N), e le istruzioni dicono che possiamo ottenere O(N).
   In effetti, possiamo trovare tutti e quattro i numeri in un unico passaggio attraverso l'array. È tempo di diventare di nuovo avidi.
   Ecco il codice, seguito dalla sua spiegazione:

```
def greatest_product(array):
    greatest_number = float("-inf")
    second_to_greatest_number = float("-inf")

    lowest_number = float("inf")
    second_to_lowest_number = float("inf")

    for number in array:
        if number >= greatest_number:
            second_to_greatest_number = greatest_number
            greatest_number = number
        elif number > second_to_greatest_number:
            second_to_greatest_number = number
```

```
        if number <= lowest_number:
            second_to_lowest_number = lowest_number
            lowest_number = number
        elif number < second_to_lowest_number:
            second_to_lowest_number = number

    greatest_product_from_two_highest = (greatest_number * second_to_greatest_number)

    greatest_product_from_two_lowest = (lowest_number * second_to_lowest_number)

    if (greatest_product_from_two_highest > greatest_product_from_two_lowest):
        return greatest_product_from_two_highest
    else:
        return greatest_product_from_two_lowest
```

Prima di iniziare il ciclo, impostiamo `greatest_number` e `second_to_greatest_number` come *infinito negativo*. Ciò garantisce che inizino da un valore inferiore a qualsiasi numero attualmente presente nell'array.
Quindi iteriamo su ciascun numero. Se il numero corrente è maggiore di `greatest_number`, trasformiamo avidamente il numero corrente nel nuovo `greatest_number`. Se abbiamo già trovato un `second_to_greatest_number`, riassegniamo il `second_to_greatest_number` in modo che sia qualunque cosa fosse il `greatest_number` prima di raggiungere il numero corrente. Ciò garantisce che il `second_to_greatest_number` sia effettivamente il penultimo numero.
Se il numero corrente sul quale stiamo eseguendo l'iterazione è minore di `greatest_number` ma maggiore di `second_to_greatest_number`, aggiorniamo `second_to_greatest_number` in modo che divenga il numero corrente.
Seguiamo lo stesso processo per trovare il `lowest_number` e il `second_to_lowest_number`.
Una volta trovati tutti e quattro i numeri, calcoliamo i prodotti dei due numeri più alti e i prodotti dei due numeri più bassi e restituiamo il prodotto maggiore.

5. La chiave per ottimizzare questo algoritmo è il fatto che stiamo ordinando un numero finito di valori. Nello specifico, ci sono solo undici tipi di letture della temperatura che possiamo trovare in questo array, vale a dire:

   ```
   95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105
   ```

   Supponiamo che il nostro array di input sia:

   ```
   [98, 99, 95, 105, 104, 99, 101, 99, 101, 97]
   ```

   Se immaginiamo il nostro array di temperature come una tabella hash, possiamo memorizzare ciascuna temperatura come chiave e il numero di occorrenze come valore. Sarebbe simile a questa:

   ```
   {98:1, 99:3, 95:1, 105:1, 104:1, 101:2, 97:1}
   ```

   Con questo in mente, possiamo eseguire un ciclo che va da 95 a 105 e controllare nella tabella hash quante occorrenze ci sono di quella temperatura. Ognuna di queste ricerche richiede un tempo pari a solo O(1).

Quindi utilizziamo quel numero di occorrenze per popolare un nuovo array. Poiché il nostro ciclo è impostato per andare da 95 a 105, il nostro array finirà in perfetto ordine ascendente.
Ecco il codice per farlo:

```
def sort_temperatures(array):
    hash_table = {}

    for temperature in array:
        if temperature in hash_table:
            hash_table[temperature] += 1
        else:
            hash_table[temperature] = 1

    sorted_temperatures = []
    temperature = 95

    while temperature <= 105:
        if temperature in hash_table:
            for i in range(hash_table[temperature]):
                sorted_temperatures.append(temperature)

        temperature += 1

    return sorted_temperatures
```

Analizziamo ora l'efficienza di questo algoritmo. Abbiamo bisogno di N passaggi per creare la tabella hash. Quindi eseguiamo un ciclo undici volte per tutte le possibili temperature da 95 a 105.
A ogni ciclo, eseguiamo un ciclo annidato per popolare `sorted_temperatures` con le temperature. Tuttavia, questo ciclo interno non finirà mai per essere eseguito più volte delle N temperature dell'array di input. Questo perché il ciclo interno viene eseguito una sola volta per ciascuna temperatura presente nell'array originale.
Pertanto, abbiamo N passaggi per creare la tabella hash, undici passaggi per il ciclo esterno e N passaggi per il ciclo interno. Questo significa 2N + 11, che si riduce a un ottimo O(N).
Questo è un classico algoritmo di ordinamento chiamato *counting sort*. È utile ogni volta che abbiamo a che fare con un intervallo relativamente piccolo di possibili valori di input, come nel nostro caso in cui ci sono solo undici valori possibili.

6. Questa ottimizzazione utilizza le ricerche magiche nel modo più brillante che abbia mai visto.
   Immaginiamo di scorrere l'array di numeri e di incontrare un 5. Poniamoci la domanda di ricerca magica: "Se potessi trovare magicamente l'informazione desiderata in un tempo O(1), potrei rendere più veloce il mio algoritmo?".
   Bene, per determinare se il 5 fa parte della sequenza consecutiva più lunga, vogliamo sapere se c'è un 6 nell'array. Vorremmo anche sapere se c'è un 7, un 8 e così via.
   Possiamo ottenere queste ricerche in un tempo O(1) se prima memorizziamo tutti i numeri del nostro array in una tabella hash; ovvero, l'array `[10, 5, 12, 3, 55, 30, 4, 11, 2]` potrebbe avere il seguente aspetto se spostassimo i dati in una tabella hash:

```
{10: True, 5: True, 12: True, 3: True, 55: True, 30: True, 4: True, 11: True, 2: True}
```

In questo caso, se incontriamo il 2, possiamo eseguire un ciclo che continua a verificare il numero successivo nella tabella hash. Se lo trova, aumentiamo di uno la lunghezza della sequenza corrente. Il ciclo ripete questo processo finché non riesce a trovare il numero successivo nella sequenza. Ognuna di queste ricerche richiede un solo passaggio.

Ma, potreste chiedervi, in che modo questo aiuta? Immaginate che il nostro array sia `[6, 5, 4, 3, 2, 1]`. Quando iteriamo sul 6, scopriremo che non c'è una sequenza che si sviluppa da lì. Quando raggiungiamo il 5, troveremo la sequenza 5-6. Quando raggiungiamo il 4, troveremo la sequenza 4-5-6. Giunti al 3 troveremo la sequenza 3-4-5-6 e così via. Finiremo comunque per eseguire circa $N^2 / 2$ passaggi per trovare tutte quelle sequenze.

La risposta è che inizieremo a costruire una sequenza solo se il numero corrente è l'ultimo numero della sequenza. Quindi non costruiremo 4-5-6 quando c'è un 3 nell'array.

Ma come facciamo a sapere se il numero corrente è l'ultimo di una sequenza? Facendo una ricerca magica!

Come? Prima di eseguire un ciclo per trovare una sequenza, eseguiremo una ricerca O(1) della tabella hash per verificare se esiste un numero inferiore di 1 rispetto al numero corrente. Quindi, se il numero corrente è 4, controlleremo prima se nell'array c'è un 3. Se c'è, non ci preoccuperemo di costruire una sequenza. Vogliamo solo costruire una sequenza a partire dal numero più basso di quella sequenza; altrimenti avremo passaggi ridondanti.

Ecco il codice per farlo:

```
def longest_sequence_length(array):
    hash_table = {}
    greatest_sequence_length = 0

    for number in array:
        hash_table[number] = True

    for number in array:
        if not hash_table.get(number - 1):
            current_sequence_length = 1
            current_number = number

            while hash_table.get(current_number + 1):
                current_number += 1
                current_sequence_length += 1

                if current_sequence_length > greatest_sequence_length:
                    greatest_sequence_length = current_sequence_length

    return greatest_sequence_length
```

In questo algoritmo, eseguiamo N passaggi per costruire la tabella hash. Poi altri N passaggi per scorrere l'array. E circa altri N passaggi cercando i numeri nella tabella hash per costruire le diverse sequenze. Anche sommati, si tratta di circa 3N, che si riduce a O(N).