Appunti di Algoritmi e Strutture Dati

Marco Natali

16 agosto 2018

Capitolo 1

Introduzione agli Algoritmi

Il termine Algoritmo proviene da Mohammed ibn-Musa al-Khwarizmi, matematico uzbeco del IX secolo a.c. da cui proviene la moderna Algebra.
Algoritmo:sequenza di passi che portano alla risoluzione di un problema
Pseudocodice: un linguaggio utilizzato per rappresentare e presentare gli algoritmi in maniera compatta e chiara;ogni libro e programmatore definisce la propria specifica di Pseudocodice ma comunque quasi tutti si ispirano alla sintassi del Pascal,C e Java.
Gli algoritmi permettono di poter migliorare e rendere il più efficiente e veloce la risoluzione di un problema e l'esecuzione di un programma.
Il primo algoritmo che è affrontiamo è INSERTION-SORT che risolve il problema dell'ordinamento di sequenze di dati. L'algoritmo INSERTION-SORT risolve il problema dell'ordinamento definito come:
Input:una sequenza di n numeri $(a_1, a_2, \ldots, a_n)$
Output:una permutazione $(a'_1, a'_2, \ldots, a'_n)$ tale che $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

INSERTION-SORT(A)

```
1  for j = 2 to A.length
2      key = A[j]
3      i = j − 1
4      while i > 0 and A[i] > key
5          A[i + 1] = A[i]
6          i = i − 1
7      A[i + 1] = key
```

L'algoritmo INSERTION-SORT è un algoritmo efficiente per ordinare un ristretto numero di elementi ed opera come farebbe un umano a riordinare le carte da gioco, ossia prendendo una carta alla volta e facendo il riordinamento delle carte una alla volta.
Per poter affermare che l'algoritmo è corretto, ossia risolve il problema, bisogna dimostrare l'invariante del ciclo **For**, attraverso un metodo simile all'induzione matematica.
L'invariante del ciclo è corretta se si riesce a dimostrare tre cose:

Inizializzazione : è corretta prima della prima esecuzione del ciclo.

Conservazione : se è verificata prima di un iterazione del ciclo lo sarà anche dopo l'esecuzione di quell'iterazione del ciclo.

Conclusione : alla fine del ciclo è ancora verificato e ciò ci aiuta a determinare la correttezza di un algoritmo.

La terza proprietà è la più importante in quanto assieme alla condizione che è causato la conclusione del ciclo, si riesce a dimostrare la correttezza dell'algoritmo. L'invariante di ciclo per l'INSERTION-SORT è: All'inizio di ogni iterazione del ciclo **for** il sottoarray $A[1\,..\,j-1]$ è ordinato ed è formato dagli stessi elementi che erano originamente in $A[1\,..\,j-1]$.

Inizializzazione : quando $j = 2$ il sottoarray $A[1\,..\,j-1]$ è formato da un solo elemento che è ordinato ed è l'elemento originale $A[1]$.

Conservazione : all'inizio di ogni esecuzione del ciclo for il sottoarray $A[1\,..\,j-1]$ è formato dai primi $j-1$ elementi dell'array ordinati dal più piccolo al più grande.

Conclusione : Quando $j > A.\mathit{length}$ il ciclo termina e dato che ogni ciclo aumenta j di 1 alla fine del ciclo si avrà $j = n+1$ per cui si ha che $A[1\,..\,n]$ è ordinato ed è formato dagli elementi ordinati che si trovavano in $A[1\,..\,n]$.

L'analisi di un algoritmo, per poter determinare se un algoritmo è efficiente, può avvenire in due maniere:

- **Tempo di Esecuzione**:è il numero di operazioni primitive che vengono eseguite da parte di un algoritmo;l'esecuzione di un'istruzione si assume che richiede un tempo costante per evitare di rendere la valutazione dipendente dall'hardware e dalla bravura del programmatore.
- **Spazio di Esecuzione**:è il numero di spazio in bit occupato in memoria dall'algoritmo ma questa analisi non viene quasi mai eseguita in quanto oramai è superfluo.

Il tempo di esecuzione dell'algoritmo è la somma dei tempi di esecuzione per ogni istruzione eseguita quindi il tempo di esecuzione di INSERTION-SORT è:

$$T(n) = c_1n+c_2(n-1)+c_3(n-1)+c_4\sum_{j=2}^{n} t_j+c_5\sum_{j=2}^{n}(t_j-1)+c_6\sum_{j=2}^{n}(t_j-1)+c_7(n-1)$$

In caso l'algoritmo sia già ordinato, caso migliore, si avrebbe sempre $A[i] < key$ quindi t_j è sempre 1, per cui il tempo di esecuzione sarebbe:

$$\begin{aligned} T(n) &= c_1n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) \\ &= (c_1 + c_2 + c_3 + c_4 + c_5)n - (c_2 + c_3 + c_4 + c_7) \\ &= \Omega(n) \end{aligned}$$

Nel caso migliore si ha che l'algoritmo richiede un tempo lineare che è un $\Omega(n)$. In caso si abbia una sequenza decrescente, corrispondente al caso peggiore, nel

ciclo While bisogna confrontare ogni elemento $A[j]$ con il sottoarray $A[1\,..\,j-1]$ per cui $t_j = j$ per $j = 2, 3, \ldots, n$ e poiche si ha

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1 \quad \sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

il tempo dell'algoritmo INSERTION-SORT nel caso peggiore è il seguente:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{n(n+1)}{2} - 1) + c_5(\frac{n(n-1)}{2}) + c_6(\frac{n(n-1)}{2}) + c_7(n-1) \\ &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{n^2+n-2}{2}) + c_5(\frac{n^2-n}{2}) + c_6(\frac{n^2-n}{2}) + c_7(n-1) \\ &= (\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2})n^2 + (c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7)n - (c_2 + c_3 + c_4 + c_7) \\ &= O(n^2) \end{aligned}$$

Il tempo dell'algoritmo può essere scritto, nel caso peggiore, come $an^2 + bn + c$ che è una funzione quadratica che viene indicata, nel caso peggiore come $O(n^2)$. Nel caso medio mi aspetto, supponendo una distribuzione uniforme della probabilità, che il vettore sia parzialmente ordinato per cui non avendo modo di rendere il ciclo while eseguibile solo una volta, si ha bisogno almeno di n^2 confronti che è $O(n^2)$.
In sintesi i tempi di esecuzione dell'algoritmo INSERTION-SORT sono:

Caso migliore : $\Omega(n)$

Caso peggiore : $O(n^2)$

Caso medio : $O(n^2)$

Un altro algoritmo per risolvere il problema dell'ordinamento è il SELECTION-SORT, che trovando via a via i più piccoli elementi della sequenza e li mette negli elementi più a sinistra.

SELECTION-SORT(A)

```
1  for j = 1 to A.length − 1
2      index = j
3      for i = j + 1 to A.length
4          if A[j] < A[index]
5              index = i
6      temp = A[j]
7      A[j] = min
8      A[index] = temp
```

L'invariante di ciclo del selection sort afferma che al termine del ciclo for più esterno si abbia il sottoarray $A[1\,..\,n]$ ordinato infatti:

Inizializzazione :nella prima iterazione del ciclo for, per $j = 1$ si ha A formato da un solo elemento che è ovviamente ordinato.

Conservazione :ogni iterazione del ciclo cerca il minimo tra $A[j\,..\,n]$ e lo posiziona in $A[j]$ per cui al termine di ogni iterazione l'array $A[1\,..\,j]$ è ordinato.

Conclusione :al termine del ciclo, con $j = n$ si abbia il sottoarray$A[1\,..\,n]$ ordinato che corrisponde all'array originale ordinato.

Il tempo di esecuzione dell'algoritmo è :

$$T(n) = c_1 n + c_2(n-1) + c_3 \sum_{i=2}^{n} i + c_4 \sum_{i=2}^{n} i + c_5 t_{if} + c_6(n-1) + c_7(n-1)$$

Possiamo semplificare l'equazione considerando ogni istruzione costante, ossia $c_i = c$, e sapendo che

$$\sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1 = \frac{n(n-1)}{2} \text{ si ottiene}$$

$$T(n) = cn + 3c(n-1) + 2c(\frac{n(n-1)}{2}) + ct_{if}$$

Il tempo di SELECTION-SORT dipende dai diversi casi:

Caso migliore : l'array è già ordinato per cui $t_{if} = 0$ e si ottiene quindi

$$T(n) = 3cn + cn^2 - 3c = \Omega(n^2)$$

Caso peggiore :l'array è ordinato in maniera decrescente per cui $t_{if} = \sum_{i=2}^{n} i$ e si ottiene quindi un tempo di esecuzione pari a

$$T(n) = 4cn - 3c + \frac{3c}{2}n^2 - \frac{3c}{2}n = O(n^2)$$

Essendo il caso peggiore uguale a caso migliore si ha $T(n) = \Theta(n^2)$

Caso medio :essendo il caso migliore coincidente con il caso peggiore il Tempo di esecuzione nel caso medio pari a $\Theta(n^2)$

1.1 Ricerca di Valori

Un altro importante problema da risolvere è la *ricerca* di valori all'interno di una sequenza di dati in quanto di solito capita di dover effettuare una ricerca all'interno dei dati. Vi sono due algoritmi per effettuare la ricerca di valori all'interno di una sequenza: la ricerca lineare e la ricerca binaria. Il primo algoritmo di ricerca analizzato è il LINEARSEARCH che risolve:

Input: una sequenza di valori interi $A[1\,..\,n]$ e un valore intero *key*

Output: un indice i tale che $A[i] = key$ altrimenti ritorna NIL

La ricerca lineare analizza tutti gli elementi fino a quando non trova l'elemento da ricercare per cui lo pseudocodice è il seguente:

```
LINEARSEARCH(A, key)
1  for j = 1 to A.length
2      if A[j] == key
3          return j
4  return NIL
```

L'invariante di ciclo della ricerca lineare afferma che al termine del ciclo for si abbia il valore dell'indice nella sequenza del valore ricercato:

Inizializzazione :nella prima iterazione del ciclo for, per $j = 1$ si ha A formato da un solo elemento che è ovviamente ordinato.

Conservazione :ad ogni iterazione del ciclo si ha che il valore dell'indice dell'elemento da trovare è NIL a meno che nell'esecuzione del ciclo venga trovato l'elemento e sia l'indice uguale a j.

Conclusione :al termine del ciclo, con $j = n$ si ha che l'indice del valore ricercato è j in caso sia stato trovato l'elemento altrimenti NIL.

Il tempo di esecuzione del LINEARSEARCH nei diversi casi è il seguente:

Caso migliore: l'elemento *key* viene trovato direttamente nel primo elemento per cui viene risolto in un tempo costante $\Omega(1)$

Caso peggiore: l'elemento *key* non viene trovato nella sequenza quindi

$$T(n) = c(n+1) + cn + c = O(n)$$

Il secondo algoritmo di ricerca è il BINARYSEARCH, algoritmo divide et impera in cui viene già previsto che la sequenza di valori sia ordinata e sfruttando ciò ad ogni passo viene eliminata una parte della sequenza.
La ricerca binaria ad ogni passo controlla l'elemento mediano per vedere se è il valore ricercato altrimenti richiama l'algoritmo sulla sottosequenza sinistra in caso il valore da trovare sia minore oppure lo effettua sulla sottosequenza destra.

```
BINARYSEARCH(A, left, right, key)
 1  if left == right
 2      if A[left] == key
 3          return left
 4      else return NIL
 5  else
 6      mid = (left + right)/2
 7      if A[mid] == key
 8          return mid
 9      if A[mid] > key
10          return BINARYSEARCH(A, left, mid − 1, key)
11      else return BINARYSEARCH(A, mid + 1, right, key)
```

Il tempo di esecuzione del BINARYSEARCH nei diversi casi è il seguente:

Caso migliore: l'elemento *key* viene trovato direttamente nell'elemento mediano per cui viene risolto in un tempo costante $\Omega(1)$

Caso peggiore: l'elemento *key* non viene trovato nella sequenza quindi

$$T(n) = \begin{cases} 3c = \Theta(1) & \text{se } n = 1 \\ T(\frac{n}{2}) + 4c & \text{se } n > 1 \end{cases}$$

Applicando il secondo caso del teorema dell'esperto in quanto $4c = \Theta(n^{\log_2 1}) = \Theta(1)$ si ha che $T(n) = \Theta(\log n)$

Capitolo 2

Divide et Impera

Nello sviluppo dell'algoritmo INSERTION-SORT abbiamo utilizzato una struttura incrementale, detta anche iterativa, ma in informatica per sviluppare gli algoritmi si può utilizzare una forma alternativa, chiamata *Divide et Impera*, come verrà specificato in questo paragrafo.
Molti utili algoritmi sono ricorsivi, ossia per risolvere un particolare problema chiamano se stessi su sottoproblemi dello stesso tipo. Generalmente gli algoritmi ricorsivi adottano un approccio **divide et impera**, il quale prevede tre passi a ogni livello di ricorsione:

Divide il problema viene diviso in un certo numero di sottoproblemi, istanze più piccole del problema.

Impera i sottoproblemi vengono risolti in maniera ricorsiva

Combina le soluzioni dei sottoproblemi vengono combinate per generare la soluzione del problema originario

2.1 MergeSort

L'algoritmo MERGE-SORT, che risolve il problema dell'ordinamento, opera seguendo il paradigma divide et impera:

Divide divide la sequenza di n elementi in due sottosequenze di $n/2$ elementi ciascuna.

Impera ordina le due sottosequenze in maniera ricorsiva mediante l'algoritmo MERGE-SORT.

Combina fonde le due sottosequenze ordinate per generare la sequenza ordinata.

Per effettuare la fusione utilizzo una procedura ausiliaria MERGE($A, left, mid, right$), dove A è un array e $left, mid, right$ sono degli indici tali che $left \leq mid \leq right$ e la procedura assume che i sottoarray $A[left\,..\,mid]$ e $A[mid + 1\,..\,right]$ siano ordinati e li fonde per formare il sottoarray $A[left\,..\,right]$ ordinato. Utilizziamo un elemento sentinella, elemento con un valore speciale tipo ∞, per semplificare il nostro pseudocodice.

MERGE($A, left, mid, right$)

```
 1  n_1 = q − p + 1
 2  n_2 = r − q
 3  crea due nuovi array L[1 .. n_1] e R[1 .. n_2]
 4  for i = 1 to n_1
 5        L[i] = A[p + i − 1]
 6  for j = 1 to n_2
 7        R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = left to right
13        if L[i] ≤ R[j]
14              A[k] = L[i]
15              i = i + 1
16  else A[k] = R[j]
17        j = j + 1
```

la procedura MERGE ha un costo $\Theta(n)$, con $n = right - mid + 1$, in quanto i tre cicli for presenti nell'algoritmo richiedono nel caso peggiore n iterazioni e non essendo annidati richiedono soltanto un tempo lineare di esecuzione. Ora possiamo utilizzare le procedura merge nell'algoritmo MERGE-SORT, il quale ordina gli elementi nel sottoarray $A[left\,..\,right]$. In caso $left \geq right$, il sottoarray ha al massimo un elemento e, quindi, è già ordinato; altrimenti il passo "Divide" calcola semplicemente un indice q che separa il sottoarray in due sottoarray di $n/2$ elementi, come mostrato dal pseudocodice:

MERGE-SORT($A, left, right$)

```
1  if left < right
2        mid = (left + right)/2
3        MERGE-SORT(A, left, mid)
4        MERGE-SORT(A, mid + 1, right)
5        MERGE(A, left, mid, right)
```

Per ordinare l'intera sequenza $A = (A[1], A[2], \ldots, A[n])$ effettuiamo la chiamata iniziale MERGE-SORT($A, 1, A.\,length$)

2.2 Analisi algoritmi divide et impera

In caso un algoritmo contiene una chiamata a se stesso, il suo tempo di esecuzione spesso può essere espresso mediante un'**equazione di ricorrenza**, in cui si esprime il tempo di esecuzione totale in funzione del tempo di esecuzione dei sottoproblemi.
Una ricorrenza per un algoritmo divide et impera si basa sui 3 passi del paradigma di base; se la dimensione del problema è sufficientemente piccola, per esempio $n \leq c$ per qualche costante c, allora il tempo di esecuzione è costante, indicato con $\Theta(1)$. In caso contrario serve un tempo $aT(n/b)$ per risolvere i sottoproblemi, con a indicante il numero di sottoproblemi generati e b indicante il rapporto di

grandezza tra il problema e i sottoproblemi, ed indicando con $D(n)$ il tempo per dividere i sottoproblemi e indicando con $C(n)$ il tempo per combinare le soluzioni dei sottoproblemi si ottiene la seguente ricorrenza

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{se } n > c \end{cases}$$

2.2.1 Analisi di Merge Sort

Lo pseudocodice di MERGE-SORT funziona per qualsiasi dimensione ma, al fine di agevolare i calcoli, supponiamo che la dimensione del problema originale sia una potenza di 2, per cui ogni passo divide genera due sottosequenze di dimensione pari a $n/2$.
L'algoritmo merge sort se applicato a un solo elemento impiega un tempo costante $\Theta(1)$, altrimenti suddividiamo il tempo di esecuzione nel seguente modo:

Divide :questo passo semplicemente calcola il centro del sottoarray quindi richiede un tempo costante $\Theta(1)$

Impera :risolviamo in maniera ricorsiva i due sottoproblemi di dimensione $n/2$ e ciò richiede $2T(n/2)$ per la risoluzione

Combina :la procedura MERGE richiede un tempo $\Theta(n)$

Quando sommiamo $\Theta(1)$ e $\Theta(n)$ otteniamo una funzione lineare che è $\Theta(n)$ e per cui l'equazione di ricorrenza di MERGE-SORT è:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{se } n > 1 \end{cases}$$

Per risolvere codesta equazione di ricorrenza vi sono 3 modalità , che saranno poi analizzate, però comunque il tempo di esecuzione nel caso peggiore è $O(n \log n)$.
Per risolvere l'equazione di ricorrenza, funzione che descrive il tempo di esecuzione in funzione del tempo di esecuzione dei sottoproblemi, vi sono 4 metodi:

Metodo di Sostituzione :ipotiziamo un tempo di esecuzione e utilizziamo l'induzione matematica per dimostrare la correttezza dell'ipotesi

Metodo dell'Esperto :fornisce i limiti dell'equazioni di ricorrenza che rispettano determinate condizioni(Analizzato nei prossimi paragrafi)

Metodo di espansione si espande l'equazione di ricorrenza fino ad arrivare ai casi base ad esempio $T(n) = T(n-1) + 3$ si espande $T(n-1), T(n-2)$ fino ad arrivare a $T(1)$.

2.3 Il metodo di Sostituzione

Il metodo di sostituzione è un tecnica di risoluzione delle equazioni di ricorrenza degli algoritmi divide et impera, che richiede due passi:

1. Ipotizzare la forma della risoluzione.

2. Usare l'induzione matematica per trovare le costanti e dimostrare che la soluzione proposta funziona ed è corretta.

Teorema 1. $T(n) = 2T(n/2) + n = \Theta(n \ln n)$

Dimostrazione. Per dimostrare che $T(n) = \Theta(n \ln n)$ bisogna provare che $T(n) \leq cn \ln n$ per una costante $c > 0$

$$\begin{aligned} T(n) &\leq 2(cn/2 \lg n/2) + n \\ &\leq cn \lg n - cn \lg 2 + n \\ &\leq cn \lg n - cn + n \\ &\leq cn \lg n \text{per} c \geq 1 \end{aligned}$$

L'induzione matematica richiede di verificare i casi base ora □

Per scegliere una buona ipotesi da verificare mediante il metodo di sostituzione richiede fantasia, esperienza. Per aiutarci ad ottenere una buona ipotesi si potrebbe utilizzare l'albero di sostituzione e poi dimostrare l'ipotesi mediante induzione, con il metodo di sostituzione, oppure ci sono delle euristiche per diventare dei buoni indovini.
Le euristiche per formulare buone ipotesi sono le seguenti:

- Se una ricorrenza è simile ad una già analizzata conviene provare a dimostrare la stessa soluzione come ad esempio:

  $T(n) = 2T(\frac{n}{2} + 17) + n$è simile all'equazione precedente ed è un$\Theta(n \lg n)$

- Si inizia a dimostrare dei limiti superiori ed inferiori molto larghi e poi restringere l'incertezza alzando il limite inferiore ed abbassando il limite superiore fino a convergere con il risultato corretto.

Ci sono dei casi in cui ipotiziamo correttamente un limite asintotico per la ricorrenza ma in qualche modo sembra che i calcoli matematici non tornino nell'Induzione. Per superare questo ostacolo spesso basta correggere l'ipotesi sottraendo un termine di ordine inferiore per far quadrare i conti, come nell'esempio:

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

Supponiamo che $T(n) = O(n)$ ossia $T(n) \leq cn$, otteniamo nella ricorrenza:

$$\begin{aligned} T(n) &\leq c\lfloor n/2 \rfloor + c\lceil n/2 \rceil + 1 \\ &= cn + 1 \end{aligned}$$

Questa equazione non implica che $T(n) \leq cn$ qualunque sia il valore di c però l'intuizione che $T(n) = O(n)$ è corretta solo che per provarla dobbiamo utilizzare un ipotesi induttiva più forte, ossia $T(n) \leq cn - d$ con $d \geq 0$ rappresentante una costante.

$$\begin{aligned} T(n) &\leq (c\lfloor n/2 \rfloor - d) + (c\lceil n/2 \rceil - d) + 1 \\ &\leq cn - 2d + 1 \\ &\leq cn - d \quad \text{per ogni} d \geq 1 \end{aligned}$$

2.4 Teorema dell'Esperto

Il teorema dell'Esperto è una per risolvere in maniera semplice ed immediata le equazioni di ricorrenza della forma $T(n) = aT(\frac{n}{b}) + f(n)$ dove $a \geq 1, b > 1$ e $f(n)$ una funzione asintoticamente positiva. Utilizzare il metodo dell'esperto richiede di memorizzare tre diversi casi e grazie a quelli posso risolvere una grande quantità di equazione di ricorrenza in maniera semplice e veloce.

Teorema 2 (Master Theorem)**.** *Sia $a \geq 1, b > 1$ e $f(n)$ una funzione asintoticamente positiva e sia abbia un equazione di ricorrenza nella forma $T(n) = aT(\frac{n}{b}) + f(n)$ allora:*

1. *se $f(n) = O(n^{\log_b a - \epsilon})$ per $\epsilon > 0$, allora $T(n) = \Theta(n^{\log_b a})$*
2. *se $f(n) = \Theta(n^{\log_b a})$, allora $T(n) = \Theta(n^{\log_b a} \log n)$*
3. *se $f(n) = \Omega(n^{\log_b a + \epsilon})$ per $\epsilon > 0$ e se $af(\frac{n}{b}) \leq cf(n)$ per una costante $c < 1$ e n sufficientemente grande, allora $T(n) = \Theta(f(n))$*

Intuitivamente confrontiamo in tutti e tre i casi $f(n)$ con $n^{\log_b a}$ e il più grande tra di essi determina la soluzione della ricorrenza ma bisogna stare attenti che vi deve essere una differenza polinomiale, controllata tramite ϵ tra $f(n)$ e $n^{\log_b a}$.
Esempio:
Data un equazione $T(n) = 2T(\frac{n}{2}) + cn$ determinare il tempo tramite il metodo dell'Esperto. $f(n) = \Theta(n^{\log_2 2}) = \Theta(n)$ per cui si applica il secondo caso del Teorema dell'esperto quindi $T(n) = \Theta(n^{\log_2 2} \log n) = \Theta(n \log n)$.

Capitolo 3

Crescita delle Funzioni

Nel valutare l'algoritmo INSERTION-SORT si ottiene una funzione $T(n) = an + b$ che è una funzione lineare.
Durante la valutazione di un algoritmo difficilmente si riesce a quantificare con esattezza le costanti coinvolte per cui si analizza il comportamento della funzione al tendere di n all'infinito.
Al tal fine si utilizzano le notazioni $O,\ \Omega,\ \Theta$ definite come segue:

- $f(n) \in O(g(n))$ se e solo se $\exists c \geq 0\ \exists m \geq 0 : f(n) \leq cg(n)\ \forall n \geq m$
- $f(n) \in \Omega(g(n))$ se e solo se $\exists c, m \geq 0 : f(n) \geq cg(n) \forall n \geq m$
- $f(n) \in \Theta(g(n))$ se e solo se $\exists c_1, c_2, m \geq 0 : c_1 g(n) \leq f(n) \leq cg(n)\ \forall n \geq m$

Per maggiore chiarezza si utilizza un'abuso di linguaggio scrivendo $f(n) = O(g(n))$ al posto di $f(n) \in O(g(n))$ e che da queste definizioni si può ricavare che $f(n) = \Theta(g(n))$ se e solo se $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$. Per poter definire se una funzione appartiene a una notazione bisogna mostrarlo attraverso una dimostrazione formale con l'utilizzo dell'induzione matematica ma fortunatamente c'è il seguente teorema per semplificare il calcolo della notazione:

Teorema 3. *Dato un polinomio del tipo* $P(n) = \sum_{i=0}^{d} a_i n^i$ *dove* a_i *sono i coefficienti e* $a_d > 0$ *si ha che* $P(n) = \Theta(n^d)$

Esempio:

$$P(n) = 5n^3 + 6n^2 + 3 = \Theta(n^3)$$

Capitolo 4

QuickSort

Il QUICK-SORT è un algoritmo divide et impera in loco ossia senza utilizzare una struttura di appoggio per effettuare l'ordinamento e funziona in maniera ottimale nell'implementazione sui calcolatori attuali.
Venne sviluppato ed ideato nel 1959 dall'importantissimo informatico C.H. Hoare e viene utilizzato come algoritmo di default delle libreria dei maggiori linguaggi.
L'algoritmo QUICK-SORT esegue i seguenti passi divide et impera:

Divide riarrangia l'array $A[p\,..\,r]$ in due sottoarray, eventualmente nulli, $A[p\,..\,q-1]$ e $A[q+1\,..\,r]$ tali che tutti gli elementi del primo sottoarray sono minori o uguali a $A[q]$ e tutti gli elementi del secondo sottoarray sono maggiori o uguali a $A[q]$.
Calcolare l'indice di q viene effettuato nella procedura di riarrangiamento.

Impera ordina ricorsivamente i due sottoarray $A[p\,..\,q-1]$ e $A[q+1\,..\,r]$.

Combina dato che i due sottoarray sono già ordinati per cui non viene eseguito nulla.

L'algoritmo QUICK-SORT è il seguente

QUICK-SORT(A, p, r)
1 **if** $p < r$
2 $\quad q =$ PARTITION(A, p, r)
3 $\quad$ QUICK-SORT$(A, p, q-1)$
4 $\quad$ QUICK-SORT$(A, q+1, r)$

Per effettuare l'ordinamento di un array A viene effettuata la chiamata iniziale QUICK-SORT(A,1,$A.\,length$).
La chiave dell'algoritmo è la procedura PARTITION che può essere implementata in due maniere:la prima, inventata da Hoare e quindi quella originale, prende come elemento pivot il primo elemento mentre la seconda versione, inventata da Lomuro, utilizza l'ultimo elemento;lo pseudocodice delle due partition è il seguente:

LOMURO-PARTITION(A, p, r)

```
1  pivot = A[r]
2  indice = p − 1
3  for j = p to r − 1
4       if A[j] ≤ pivot
5            indice = indice + 1
6            scambia A[idindice] con A[j]
7  scambia A[indice + 1] con A[r]
8  return indice + 1
```

HOARE-PARTITION(A, p, r)

```
1  pivot = A[p]
2  i = p
3  for j = p to r
4       if A[j] < pivot
5            i = i + 1
6            scambia A[i] con A[j]
7  scambia A[i] con A[r]
8  return i
```

Il tempo di esecuzione della procedura PARTITION, sia Lomuro che Hoare, è il seguente: $T(n) = c_1 + c_2 + c_3(n+1) + c_4 n + c_5 n(t_{if}) + c_6 n(t_{if}) + c_7 + c_8$.

Caso migliore :tutti gli elementi sono maggiori del pivot per cui $t_{if} = 0$ $T(n) = (c_3 + c_4)n + (c_1 + c_2 + c_3 + c_7 + c_8) = \Theta(n)$

Caso peggiore :tutti gli elementi sono inferiori del pivot per cui $t_{if} = 1$ $T(n) = (c_3 + c_4 + c_5 + c_6)n + (c_1 + c_2 + c_3 + c_7 + c_8) = \Theta(n)$

4.1 Analisi tempo QuickSort

L'equazione di ricorrenza generale del Quicksort è la seguente:

$$T(n) = \begin{cases} 0 & \text{se } n = 0, 1 \\ T(n-j) + T(j) + \Theta(n) & \text{se } n > 1 \end{cases}$$

L'analisi del tempo di esecuzione del QUICK-SORT è in base al fatto se la partizione dell'array è bilanciata o meno ossia se i due sottoproblemi da risolvere sono più o meno della stessa dimensione.

Caso peggiore :la procedura di partizione produce due sottoproblemi: uno di $n-1$ elementi e l'altro di 0 elementi. $T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$ Attraverso il metodo di sostituzione arrivo a $T(n) = O(n^2)$

Caso migliore :la procedura di partizione produce due sottoproblemi di $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$ elementi per cui, ignorando le condizioni di ceil e floor, il tempo di esecuzione è $T(n) = 2T(n/2) + \Theta(n) = \Omega(n \log n)$ per il teorema dell'esperto.

Caso medio :

Una versione randomizzata del quicksort, utile per ottenere una buona prestazione attesa in tutti gli input, consiste nel prendere come pivot ad ogni iterazione un elemento a caso tra p e r così l'elemento pivot avrà la stessa possibilità di essere un elemento del sottoarray per cui ci aspettiamo che la ripartizione dell'array sia ben bilanciata in media.

RANDOMIZED-PARTITION(A, p, r)
1 i = RANDOM(p, r)
2 scambia $A[i]$ con $A[r]$
3 **return** PARTITION(A, p, r)

La procedura randomizzata Quicksort chiama RANDOMIZED-PARTITION invece che PARTITION, per cui il suo pseudocodice è il seguente:

RANDOMIZED-QUICKSORT(A, p, r)
1 **if** $p < r$
2 $\quad q =$ RANDOMIZED-PARTITION(A, p, r)
3 $\quad$ RANDOMIZED-QUICKSORT$(A, p, q - 1)$
4 $\quad$ RANDOMIZED-QUICKSORT$(A, q + 1, r)$

Il tempo di esecuzione del Quicksort randomizzato coincide ovviamente a quello normale solo che il caso medio si verifica molto spesso ed è molto raro il caso peggiore per cui di solito viene utilizzata la versione randomizzata del Quicksort.

Capitolo 5

Ordinamento in tempo lineare

Gli algoritmi di ordinamento analizzati fino ad ora hanno un importante proprietà, ossia effettuano l'ordinamento soltanto mediante il confronto tra gli elementi in input. In questo capitolo verrà analizzato il limite minimo di confronti da effettuare e verranno analizzati altri due algoritmi di Ordinamento: il counting e il radix sort.

5.1 Ordinamento in base ai confronti

In un ordinamento in base ai confronti per riordinare la sequenza $(a_1, a_2, \ldots, a_n)$ bisogna effettuare uno solo dei seguenti test $a_i > a_j, a_i < a_j, a_i \leq a_j, a_i \geq a_j, a_i = a_j$. Assumendo di avere tutti gli elementi di una sequenza distinti e considerando che i confronti sono tutti equivalent, in quanto permettono di stabilire l'ordine di due elementi, consideriamo solo i confronti $a_i \leq a_j$: definiamo come *albero dei confronti* un albero binario in cui vengono rappresentati i confronti fra gli elementi effettuati da un particolare algoritmo di ordinamento; l'esecuzione dell'algoritmo di ordinamento corrisponde al tracciare un cammino semplice dalla radice dell'albero di decisione fino ad arrivare ad una foglia.
La lunghezza del cammino semplice più lungo dell'albero determina il numero di confronti effettuati nel caso peggiore dall'algoritmo di ordinamento; di conseguenza il numero di confronti nel caso peggiore corrisponde all'altezza del suo albero di decisione.

Teorema 4. *Qualsiasi algoritmo di ordinamento per confronti richiede* $\Omega(n \log n)$ *nel caso peggiore.*

Dimostrazione. Per riuscire a dimostrare il seguente teorema è sufficiente determinare l'altezza del suo albero di decisione dove ogni permutazione appare come una foglia raggiungibile.
Consideriamo un albero di decisione di altezza h con l foglie raggiungibili che corrisponde a un ordinamento per confronti di n elementi: poichè ciascuna delle $n!$ permutazioni dell'input compaiono in una foglia, si ha $n! \leq l$, e dal momento che un albero binario di altezza h non ha più di 2^h foglie si ottiene $n! \leq l \leq 2^h$.

Passando ai logaritmi si ricava che $h \geq \log(n!)$ in quanto log è monotonicamente crescente da cui otteniamo che $\Theta(n \log n)$, come dimostriamo adesso.

$$\begin{aligned}\log(n!) &= \log(n * n - 1 * \cdots * 2 * 1) \\ &= \log 1 + \log 2 + \cdots + \log(n-1) + \log n \\ &\leq \log(n^n) \\ &\leq n \log n = O(nlogn)\end{aligned}$$

Resta da dimostrare ora che $\log n!$ sia $\Omega(nlogn)$ per riuscire a dimostrare che qualsiasi algoritmo di ordinamento richiede $n \log n$ per effettuare l'ordinamento:

$$\begin{aligned}\log(n!) &= \log 1 + \log 2 + \cdots + \log(n-1) + \log n \\ &\geq \log n/2 + \log(n/2+1) + \cdots + \log(n-1) + \log n \\ &\geq n/2 \log n/2 \\ &= \Omega(n \log n)\end{aligned}$$

□

5.2 Counting-Sort

Il CountingSort è un algoritmo di ordinamento in cui viene assunto che ognuno dei n elementi sia un numero intero nel range $[0, k]$ e per effettuare l'ordinamento determina per ogni elemento x il numero degli elementi con valore minore rispetto ad x; questa informazione viene utilizzata per piazzare l'elemento x nella corretta posizione nell'array di output infatti ad esempio se 17 elementi sono minori di x viene messo x nella posizione 18 nell'array di output.
Nello pseudocodice del CountingSort, utilizziamo l'array $A[1\,..\,n]$ come array di input, l'array $B[1\,..\,n]$ come array di output ed infine $C[0\,..\,k]$ come array temporaneo per calcolare il numero di elementi inferiori, come verrà mostrato di seguito:

An important property of counting sort is that it is stable: numbers with the same value appear in the output array in the same order as they do in the input array. That is, it breaks ties between two numbers by the rule that whichever number appears first in the input array appears first in the output array. Normally, the property of stability is important only when satellite data are carried around with the element being sorted. Counting sort's stability is important for another reason: counting sort is often used as a subroutine in radix sort. As we shall see in the next section, in order for radix sort to work correctly, counting sort must be stable.

5.3 Radix-Sort

Capitolo 6

Strutture Dati Elementari

In questo capitolo verranno definite le strutture dati elementari, ma prima di poterle definire bisogna definire il concetto di tipo di dato.
Il tipo di dato è un modello matematico in cui sono definite un certo numero di operazioni e in ogni linguaggio di programmazione vengono definiti e previsti dei tipi di dato detti *primitivi*, come ad esempio i numeri interi, i caratteri ed ecc... ma può essere comodo e conveniente definire altre tipologie di dati per rendere più facile e chiara la definizione e l'implementazione di un algoritmo.
In generale le proprietà di un tipo di dato devono dipendere soltanto dalla sua specifica ed essere indipendenti dalla modalità in cui vengono rappresentati per cui si dice che un tipo di dato è *astratto*, in quanto il dato è astratto rispetto alla sua rappresentazione.
Il vantaggio di avere i tipi di dato astratti consiste nel poter utilizzare il dato senza conoscere la sua rappresentazione ed eventuali modifiche alla rappresentazione del dato non comportano alcun cambiamento nell'utilizzo del dato da parte dell'utilizzatore.

6.1 Liste

Le liste implementano il concetto matematico di sequenza lineare di oggetti, in cui si possono eventualmente ripetere gli elementi.
La lista è una struttura dati lineare dinamica in cui l'accesso all'elemento successivo della sequenza avviene tramite un puntatore all'elemento successivo ed un elemento è composto da un valore , chiamato *element* e da due puntatori *prev* e *next*, i quali puntano all'elemento precedente o successivo della lista.
In caso *prev* = NIL l'elemento non ha nessun predecessore ed è la *head* della lista mentre il puntatore *next* = NIL l'elemento non ha nessun successore per cui è la coda della lista. Vi sono diversi tipologie di caratteristiche che una lista può possedere, anche in maniera multipla ossia può possedere più di una proprietà, come si può notare dal seguente elenco:

- singolarmente o doppiamente concatenata: in caso una lista sia singolarmente concatenata si ha soltanto il collegamento con l'elemento successivo, per cui viene omesso il puntatore *prev*, mentre nella lista doppiamente concatenata si ha il collegamento, tramite i puntatori, con l'elemento precedente e l'elemento successivo.

- ordinata: una lista si dice ordinata se è previsto un ordinamento tra i valori degli elementi presenti in una lista.
- circolare: il puntatore *prev* della testa della lista punta alla coda mentre il puntatore *next* della coda della lista punta alla testa per cui si può dire che la lista è un anello di elementi.

La specifica di una sequenza, qualsiasi implementazione abbia, è la seguente:
void insert(**List** L,**Item** x):inserisce un elemento in una lista
List* delete(**List** L): elimina il primo elemento da una lista
List* search(**List** L,**Item** key): cerca l'elemento key all'interno della lista.
Item min(**List** L):calcola e restituisce l'elemento minimo della lista.
Item max(**List** L):calcola e restituisce l'elemento massimo della lista.
Item successor(**List** L,**Item** key):calcola l'elemento predecessore di un elemento della lista.
Item predecessor(**List** L,**Item** key):calcola il successore di un elemento della lista.

Forniamo ora un implementazione di una lista doppiamente concatenata non ordinata in cui un elemento della lista è composto da un dato chiamato *element* e da due puntatori, chiamati *prev* e *next*, che puntano all'elemento precedente e successivo.
La prima operazione implementata in una lista è la ricerca di un elemento che opera secondo l'algoritmo di ricerca Lineare in quanto non essendo ordinati i valori della lista non è possibile implentare la ricerca tramite la ricerca binaria. Lo pseudocodice della ricerca di un elemento di una lista è il seguente:

LIST-SEARCH(L, key)

```
1  x = L.head
2  while x ≠ NIL and x.element ≠ key
3        x = x.next
4  return x
```

La ricerca di un elemento da una lista richiede il seguente tempo con i diversi casi:

$$T(n) = c + c(t_w + 1) + ct_w + c$$

Caso migliore :l'elemento da ricercare viene trovato al primo elemento della lista per cui $t_w = 0$ indi il tempo di esecuzione è $T(n) = c + c + c = \Omega(1)$

Caso peggiore : l'elemento non è presente nella lista per cui $t_w = n$

$$T(n) = c + cn + c + cn + c = 2cn + 3c = O(n)$$

Il secondo metodo in una lista è LIST-INSERT in cui si suppone che il valore dell'elemento da inserire sia stato già impostato ossia *element* abbia già il valore desiderato.

LIST-INSERT(L, x)

```
1  x.next = L.head
2  if L.head ≠ NIL
3        L.head.prev = x
4  L.head = x
5  x.prev = NIL
```

Il tempo di esecuzione $T(n) = 5c = \Theta(1)$ e tra il caso migliore e peggiore non vi è alcuna differenza se non il fatto che non viene eseguita soltanto la terza istruzione.
La rimozione di un elemento da una lista, implementata tramite LIST-DELETE, prevede di darne il puntatore all'elemento alla procedura per cui per effettuare la rimozione di un elemento qualsiasi della lista bisogna effettuare la chiamata a LIST-SEARCH prima per ottenere l'elemento da eliminare. Lo pseudocodice per la rimozione di un elemento è il seguente:

LIST-DELETE(L, x)

```
1  if x.prev ≠ NIL
2        x.prev = x.next
3  else L.head = x.next
4  if x.next ≠ NIL
5        x.next.prev = x.prev
```

Il tempo di esecuzione è $T(n) = 4c = \Theta(1)$ e tra il caso peggiore e migliore non cambia nulla però va detto che se si volesse implementare la rimozione da un elemento qualsiasi della lista si avrebbe un tempo di esecuzione $\Theta(n)$ in quanto si avrebbe la chiamata alla procedura LIST-SEARCH per stabilire l'elemento da rimuovere.
Per trovare il massimo e/o il minimo di una lista bisogna scansionare tutta la lista e scegliere l'elemento con valore massimo/minimo come si farebbe anche se si dovesse trovare il massimo di un array; lo pseudocodice del massimo e del minimo sono i seguenti:

List* LIST-MAX(**List**L)

```
1  if L.head == NIL
2        return NIL
3        max = L.head
4        elem = L.head.next
5        while elem ≠ NIL
6              if elem.value > max.value
7                    max = elem
8              elem = elem.next
9        return max
```

Senza effettuare un'analisi linea per linea si può facilmente notare che il calcolo del massimo di una lista di n elementi richieda $\Theta(n)$ in quanto bisogna scansionare tuttti gli elementi per poterne determinare il massimo.

List*LIST-MIN(**List**L)

```
1  if L.head == NIL
2        return NIL
3        min = L.head
4        elem = L.head.next
5        while elem ≠ NIL
6              if elem.value < min.value
7                    min = elem
8              elem = elem.next
9        return min
```

Ovviamente, come anche già notato nel calcolo del massimo di una lista, il tempo di esecuzione per il calcolo del minimo è $\Theta(n)$ in quanto bisogna sempre scansionare la lista.
Le ultime due procedure da implementare sono quello di trovare l'elemento successore e predecessore come valore di un elemento ossia viene definito come successore il più piccolo elemento più grande dell'elemento dato mentre il predecessore il più grande elemento più piccolo come valore dell'elemento dato;ovviamente, come anche già visto per il minimo e il massimo, bisogna scandirre tutta la lista per stabilire il prede/successore e lo pseudocodice è il seguente:

List*LIST-SUCCESSOR(**List**L, **int**key)

```
1  if L.head == NIL
2       return NIL
3       succ = L.head
4       while succ ≠ NIL and succvalue ≤ key
5       if succ == NIL return NIL
6       elem = succ.next
7       while elem ≠ NIL
8            if elem.value > key and elem.value < succ.value
9                 succ = elem
10           elem = elem.next
11      return succ
```

Il tempo di esecuzione è $\Theta(n)$ in quanto sia nel caso migliore che in quello peggiore bisogna analizzare tutti gli elementi della lista.

List*LIST-PREDECCESSOR(**List**L, **int**key)

```
1  if L.head == NIL
2       return NIL
3       prev = L.head
4       while prev ≠ NIL and prevvalue ≥ key
5       if prev == NIL return NIL
6       elem = prev.next
7       while elem ≠ NIL
8            if elem.value < key and elem.value > prev.value
9                 prev = elem
10           elem = elem.next
11      return prev
```

Come anche per il successore, il tempo di esecuzione per trovare il predecessore di un elemento è $\Theta(n)$ in quanto bisogna analizzare tutti gli elementi di una lista.
Le altre implementazioni delle diverse tipologie di liste sono similari soltanto che implementano o meno l'ordinamento tra gli elementi, considerano o meno il puntatore prev ed altri considerazioni fatte in base alla tipologia di lista.
Un implementazione alternativa della sequenza, anche se meno intutiva e naturale di quella presentata fino ad ora, è quella tramite la memorizzazione degli elementi in un vettore, in cui la posizione di un elemento corrisponde all'indice del vettore. Questa implementazione permette di passare in maniera costante da un elemento ad un altro, di accorgersi se si supera un estremo della sequenza, di modificare o

leggere il valore di un elemento anche tramite un accesso diretto tramite indice, ma sfortunatamente richiede di conoscere la dimensione massima della sequenza per evitare sprechi di memoria e il tempo di inserimento e cancellazione richiede la scansione della sequenza per cui a tempo $\Theta(n)$.

6.2 Code

La *Queue*, in italiano *coda*, è una struttura dati di tipo FIFO(First in First out) per memorizzare una sequenza di elementi, in cui l'inserimento di un elemento avviene in coda alla sequenza mentre la rimozione avviene in testa alla sequenza. Una coda può essere implementata attraverso array oppure delle liste a seconda della scelta implementatica e della capacità di stabilire un limite massimo di elementi utilizzati.
Essendo una coda una particolare tipologia di sequenza può essere implementata facilmente utilizzando una lista e le sue operazioni però, a differenza dello stack, la scelta della lista utilizzata per l'implementazione cambia il tempo di esecuzione delle operazioni, infatti soltanto con una lista bidirezionale si ottiene un tempo $\Theta(1)$ in tutte le operazioni per cui noi utilizziamo una lista bidirezionale per implementare una coda.
Il metodo isEmpty indica se la coda contiene o meno degli elementi

```
isEmpty()
1  return (Q.head == NIL)
```

Il metodo enqueue(Q, x) inserisce un elemento nella coda e si assume che x sia un elemento definito prima della chiamata al metodo.

```
Enqueue(Q, x)
1  Q.tail.next = x
2  Q.tail = x
```

L'esecuzione del metodo enqueue richiede $2c$ ossia $\Theta(1)$ per ciò l'inserimento in una coda richiede un tempo costante.
Il metodo dequeue() estra il primo elemento della coda ed è implementato come

```
Dequeue()
1  if isEmpty()
2        return NIL
3        temp = Q.headelement
4        Q.head = Q.head.next
5        return temp
```

L'operazione dequeue richede un tempo costante $\Theta(1)$ per effettuare la rimozione.
Dopo aver visto come viene implementata una coda tramite puntatori, consideriamo l'implementazione tramite un vettore $Q[1\,..\,n]$ e ai campi $Q.head$ e $Q.tail$ per accedere all'elemento in testa e in coda ai vettore della coda;lo pseudocodice della coda tramite un vettore è il seguente:

ENQUEUE(Q, x)

```
1  Q[Q.tail] = x
2  if Q.tail == Q.length
3        Q.tail = 1
4  else Q.tail = Q.tail + 1
```

DEQUEUE(Q)

```
1  x = Q[Q.head]
2  if Q.head == Q.length
3        Q.head = 1
4  else Q.head = Q.head + 1
5        return x
```

QUEUE-EMPTY(Q)

```
1  return (Q.head ≤ 0)
```

Il tempo di esecuzione delle seguente procedure è sempre costante $\Theta(1)$ come anche la coda tramite sequenza ma ha senso implementare tramite vettore se e soltanto se si sa determinare il numero degli elementi per evitare uno spreco di memoria.

6.3 Stack

Lo *stack*, è una struttura dati di tipo *LIFO*(Last in first out), utilizzata in tutti i linguaggi di programmazione per effettuare la memorizzazione di tutti i dati di tipo statico e dei record di attivazione, che può essere vista come un caso particolare di sequenza in cui l'inserimento avviene alla fine della sequenza e la rimozione avviene sempre in fondo.
Uno stack può essere implementato attraverso array oppure delle liste a seconda della scelta implementatica e della capacità di stabilire un limite massimo di elementi utilizzati.
Essendo lo stack un particolare tipo di sequenza, essa può essere simulata tramite le operazioni di una lista, in particolare quella singolarmente concatenata, però è prassi comune utilizzare nomi diversi per indicarne le operazioni per migliore chiarezza.
void PUSH(S, x) inserisce un elemento in testa allo stack
Item* POP(S) rimuove l'elemento in testa allo stack
boolean STACKEMPTY indica se lo stack è vuoto o meno.
Lo pseudocodice delle operazioni di uno stack implementate tramite liste sono:

PUSH(S, x)

```
1  x.next = S.top
2  S.top = x
```

La procedura POP rimuove l'ultimo elemento inserito nello stack e lo ritorna come valore; in caso lo stack è vuoto genera underflow come errore.

```
POP(S)
1  temp = S.top
2  if STACKEMPTY(S)
3        error "Underflow"
4  S.top = S.top.next
5  return temp
```

La procedura STACKEMPTY indica se lo stack è vuoto e viene utilizzato per assicurarsi di non provare ad accedere allo stack vuoto per la rimozione

```
STACKEMPTY(S)
1  return S.top == NIL
```

Utilizzando le liste per implementare lo stack otteniamo in tutte le operazioni l' impiego di tempo costante $\Theta(1)$.
L'implementazione dello stack tramite un vettore ha lo stesso impiego di tempo $\Theta(1)$ in tutte le operazioni però per evitare uno spreco di memoria bisogna sapere il numero di elementi necessari e soprattutto non è possibile superare il numero di elementi massimo stabilito alla creazione dello stack.
La realizzazione delle operazioni dello stack tramite un vettore sono le seguenti:

```
PUSH(S, x)
1  S.top = S.top + 1
2  S[S.top] = x
```

```
POP(S)
1  if STACKEMPTY(S)
2        error "Underflow"
3  else S.top = S.top − 1
4        return S[S.top + 1]
```

```
STACKEMPTY(S)
1  return S.top ≤ 0
```

Un famoso esempio di utilizzo dello stack è quello di valutare delle espressioni algebriche scritte in input tramite stringhe come cercheremo di fare ora:

Capitolo 7

Heap

Lo *heap* è una struttura dati rappresentata da un array A che può essere vista come un albero binario in cui ogni nodo dell'albero è un elemento dell'array, chiamato *chiave*.
L'array A ha 2 attributi: $A.length$ per rappresentare la lunghezza dell'array e $A.heap\text{-}size$ che indica il numero degli elementi dell'heap memorizzati in cui $0 \leq A.heap\text{-}size \leq A.length$. Ci sono due tipologie di Heap:*max-heap*, utilizzato nel heapSort e *min-heap*, utilizzato principalmente per implementare code prioritarie;queste due tipologie verranno analizzate entrambe in seguito in questo paragrafo.
Per poter effettuare l'accesso ai nodi left,right e parent si utilizzano le seguenti 3 procedure
In un *max-heap* è soddisfatta per ogni nodo i la seguente proprietà: $A[\textsc{Parent}(i)] \geq A[i]$ per cui il massimo valore della array si trova nella radice dell'heap mentre in un *min-heap* avviene il contrario ossia in ogni nodo si ha $A[\textsc{Parent}(i)] \leq A[i]$ e la radice rappresenta l'elemento minimo dell'array. Essendo lo heap definito tramite un albero binario la sua altezza è $\Theta(\log n)$.
Capitolo 6 Heap

7.1 Alberi

Si definisce come *Albero libero*, un DAG connesso con un solo nodo sorgente, detto *radice*, in cui ogni nodo diverso dalla radice ha un solo nodo entrante.
I nodi privi di archi entranti sono detti *foglie* dell'albero.
L'albero è una struttura matematica importantissima in informatica utilizzata per rappresentare una serie di situazioni, come ad esempio organizzazioni gerarchiche di dati, procedimenti enumerativi o decisionali, e ve ne esistono un'infinita di implementazioni di alberi però iniziamo ad analizzare per prima gli alberi liberi binari.
I metodi di visita di un albero binario sono 3:

- inorder: si visiona prima il sottoalbero sinistro poi il nodo e infine il sottoalbero destro
- preorder: si visiona prima il nodo poi i suoi sottoalberi
- postorder: si visionano prima i sottoalberi ed infine il nodo

Il primo metodo di visita viene usato soprattutto negli alberi binari di ricerca per stampare gli elementi dell'albero in maniera crescente mentre in un albero binario normale la scelta di quale metodo di visita utilizzare è inifluente e ogni programmatore sceglie nell'utilizzo quale metodo di visita utilizzare per stampare l'albero. Il cammino dalla radice ad un elemento foglia dell'albero richiede al massimo $O(h)$, in cui h è l'altezza dell'albero, in quanto richiede di scendere di livello fino ad arrivare alle foglie, che si trovano al livello h.
La specifica di un albero binario, in cui ogni implementazione per essere valida deve prevedere:
Item search(Tree T,Item k);
void insert(Tree T,Item x);
Item delete(Tree T,Item x);
Item mininum(Tree T);
Item maxinum(Tree T);
Item predecessor(Tree T,Item x);
Item successor(Tree T,Item x);

Capitolo 8

Alberi Binari di Ricerca

Dato un insieme di nodi in cui è definita una relazione d'ordine, si definisce come *albero di ricerca* un albero in cui tutti i nodi della radice sinistra sono minori della radice e tutti i nodi a destra della radice sono maggiori e ogni sottoalbero è anch'esso un albero di ricerca.
I metodi di visita di un albero visti nel capitolo precedente sono valide anche per gli alberi binari di ricerca in particolare la visita INORDER consente di mostrare i valori dell'albero ordinati in maniera corretta.
Le operazioni definite di solito in un albero binario di ricerca sono le seguenti:
Item search(Tree T,Item k);
void insert(Tree T,Item z);
void delete(Tree T,Item z);
Item min(Tree T);
Item maxinum(Tree T);
Item predecessor(Tree T,Item x);
Item successor(Tree T,Item x);
Tutte le seguenti operazioni richiedono un tempo $O(h)$ con h indicante l'altezza dell'albero per cui è meglio riuscire a mantenere il più possibile l'albero bilanciato per ottenere l'altezza $h = \log n$.
L'operazione SEARCH permette di ricercare un elemento all'interno di un albero binario di ricerca;per la proprietà di avere un ordinamento tra gli elementi dell'albero la ricerca di un elemento ricalca quella della ricerca binaria per cui il pseudocodice è:

BST-SEARCH($TreeT, Itemk$)
1 $x = T.root$
2 **if** $x ==$ NIL or $x.key == k.key$
3 **return** x
4 **if** $x.key < k.key$
5 **return** TREE-SEARCH($T.right, k$)
6 **else return** TREE-SEARCH($T.left, k$)

La ricerca di un elemento richiede la scansione dell'albero dalla radice alle foglie che sappiamo essere pari a $O(h)$ con h indicante l'altezza dell'albero. La procedura TREE-MIN calcola il minimo dell'albero binario di ricerca sfruttando il fatto che l'elemento minimo dell'albero si trova nel sottoalbero più a sinistra presente nell'albero per cui lo pseudocodice è facilmente implementabile come:

BST-MIN($TreeT$)

```
1  if T.left == NIL
2        return T
3  else return TREE-MIN(T.left)
```

La scansione dell'albero per trovare il sottoalbero più a sinistra richiede $O(h)$ con h rappresentante l'altezza dell'albero;in caso l'albero sia bilanciato il tempo sarebbe $O(\log n)$. Simmetricamente e con il tempo di esecuzione uguale, è l'algoritmo di ricerca del massimo di un albero di ricerca, in cui il massimo si trova nel sottoalbero più a destra:

BST-MAX($TreeT$)

```
1  if T.right == NIL
2        return T
3  else return TREE-MAX(T.right)
```

Scandendo l'array andando sempre nel sottoalbero destro richiede un tempo $O(h)$, come anche già visto per la ricerca del minimo, con h indicante l'altezza dell'albero.

BST-SUCCESSOR($TreeT, Itemx$)

```
1  if x.right ≠ NIL
2        return TREE-MIN(T.right)
3        y = x.p
4        while y ≠ NIL and x == y.right
5              x = y
6              y = y.p return y
```

BST-PREDECESSOR($TreeT, Itemx$)

```
1  if x.left ≠ NIL
2        return TREE-MAXINUM(T.left)
3  y = x.p
4  while y ≠ NIL and x == y.left
5        x = y
6        y = y.p
7  return y
```

L'inserimento di un'elemento in un albero binario di ricerca prevede di effettuare un cammino, considerando il valore dell'elemento, per trovare la prima posizione nulla, che rispetta la proprietà di albero di ricerca, dove inserire l'elemento.
Analogamente alle altre operazioni su alberi l'inserimento richiede $O(h)$ in un albero di altezza h e il suo pseudocodice è il seguente:

```
BST-INSERT(Tree T, Item z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4        y = x
 5        if x.value < z.value
 6              x = x.right
 7        else x = x.left
 8        z.p = y
 9        if y == NIL
10              T.root = z
11        elseif y.value < z.value
12              y.right = z
13        else y.left = z
```