# Appunti di Algoritmi e Strutture Dati

Marco Natali

6 maggio 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 quando si effettua un qualsiasi Algoritmo può capitare di dover effettuare una ricerca all'interno dei dati. Vi sono due algoritmi per effettuare la ricerca di valori all'interno di una sequenza: il primo è LINEARSEARCH che effettua una scansione della sequenza mentre l'altro è il BINARYSEARCH, algoritmo divide et impera che opera su sequenze ordinate. Il secondo è utile quando si sa che i dati saranno e sono ordinati altrimenti effettuare l'ordinamento rende vano il guadagno dell'algoritmo rispetto alla ricerca Lineare. 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

```
LINEARSEARCH(A, key)
1  for j = 1 to A.length
2      if A[j] == key
3          return j
4  return 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 che prevede che la sequenza di valori sia già ordinata in cui ad ogni passo viene eliminata una parte della sequenza in quanto i valori di quella parte di sequenza si è riusciti a determinare che sono fuori del range.

```
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 specificato in questo paragrafo.
Molti utili algoritmi sono ricorsivi, ossia per risolvere un particolare problema questi algoritmi chiamano se stessi per risolvere 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

Un esempio del paradigma divide et impera viene dato dall'algoritmo MERGE-SORT, algoritmo che risolve il problema dell'ordinamento
L'algoritmo MERGE-SORT 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 $\infty$, 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 il tempo di esecuzione di 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'albero di Ricorsione** :converte l'equazione in un albero i cui nodi rappresentano i costi ai vari livelli della ricorsione

**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 al caso base.

## 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 $c > 0$

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

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 Albero di Ricorsione

In un albero di Ricorsione ogni nodo rappresenta il costo del sottoproblema nella chiamata ricorsiva di risoluzione dell'algoritmo fino ad arrivare al caso base. Dato un albero l'altezza è al massimo il $\log n$ dove $n$ indica il numero dei nodi per cui l'ultimo livello della ricorsione ci aspettiamo che intervenga per il $\log n$ al computo del tempo di esecuzione dell'Algoritmo. Di solito si utilizza l'albero di ricorsione per generare un ipotesi da dimostrare mediante induzione, con il metodo di sostituzione, per cui si può tollerare un pò di incertezza nell'analisi, ad esempio togliere le costanti ed evitare di considerare i floor e ceil quando si effettua la divisione dell'input in due sottoproblemi.
Albero di ricorsione dell'equazione di ricorrenza $T(n) = 3T(n/4) + \Theta(n^2)$

## 2.5 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 $\epsilon$ 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 $f(n) \in O(g(n))$.
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.
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      q = PARTITION(A, p, r)
3      QUICK-SORT(A, p, q − 1)
4      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 implementato come:

```
PARTITION(A, p, r)
1  pivot = A[r]
2  indice = p − 1
3  for j = p to r
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
```

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

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

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

## 4.1 Analisi tempo QuickSort

L'analisi del tempo di esecuzione del QUICK-SORT è in base al fatto se la partizione dell'array è bilanciata o meno.

**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** :effettuare l'analisi con una ripartizione 9 a 1

# Capitolo 5

# 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.

## 5.1 Liste

Le liste sono una struttura dati elementare che implementa il concetto matematico di sequenza lineare di oggetti, in cui si possono eventualmente ripetere gli elementi all'interno della sequenza.
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.

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** $x \neq$ NIL and $x.element \neq key$
3 $\quad x = x.next$
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 il valore desiderato.

LIST-INSERT$(L, x)$
1 $x.next = L.head$
2 **if** $L.head \neq$ NIL
3 $\quad L.head.prev = x$
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.
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 $\Theta(n)$.

## 5.2 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.

## 5.3 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, 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 però è prassi comune utilizzare nomi diversi per indicarne le operazioni per migliore chiarezza.
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:
.
Un alternativa implementazione dello stack avviene tramite

# Capitolo 6

# 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 $\Theta(\log n)$.
Strutture dati aggiornate suffix-tree FM-index suffix-array wavelet-tree Bloom filters Sequence Bloom trees Tabelle Hash  Capitolo 6 Heap