

# Introduzione agli Algoritmi

Simone Lidonnici

10 aprile 2024

# Indice

<b>1</b>	<b>Notazione asintotica</b>	<b>2</b>
1.1	Tipi di notazione asintotica . . . . .	2
1.1.1	O grande . . . . .	2
1.1.2	Omega . . . . .	2
1.1.3	Theta . . . . .	3
1.2	Algebra della notazione asintotica . . . . .	3
1.2.1	Gerarchia . . . . .	3
1.2.2	Sommatorie notevoli . . . . .	4
<b>2</b>	<b>Costo computazionale</b>	<b>5</b>
2.1	Costo delle istruzioni . . . . .	5
2.1.1	Istruzioni elementari . . . . .	5
2.1.2	Istruzioni iterative . . . . .	5
2.1.3	Calcolo del costo computazionale . . . . .	6
2.2	Tempi di esecuzione . . . . .	7
<b>3</b>	<b>Algoritmi di Ricerca</b>	<b>8</b>
3.1	Ricerca sequenziale . . . . .	8
3.2	Ricerca binaria . . . . .	8
<b>4</b>	<b>Ricorsione</b>	<b>10</b>
4.1	Iterazione vs Ricorsione . . . . .	10
<b>5</b>	<b>Equazioni di ricorrenza</b>	<b>12</b>
5.1	Scrittura di un'equazione di ricorrenza . . . . .	12
5.2	Metodo iterativo . . . . .	12
5.2.1	Caso particolare: sequanza di Fibonacci . . . . .	13
5.3	Metodo dell'albero . . . . .	14
5.4	Metodo di sostituzione . . . . .	14
5.5	Metodo principale . . . . .	15
<b>6</b>	<b>Algoritmi di Sorting</b>	<b>17</b>
6.1	Insertion sort . . . . .	17
6.2	Selection sort . . . . .	18
6.3	Bubble sort . . . . .	18
6.4	Alberi di decisioni . . . . .	19
6.5	Merge sort . . . . .	19
6.6	Quick sort . . . . .	20

# 1

## Notazione asintotica

### 1.1 Tipi di notazione asintotica

La notazione asintotica serve per valutare l'efficienza di un algoritmo con una formula matematica e permette di confrontare il tasso di crescita di una funzione rispetto ad un'altra. In informatica si usa per stimare quanto aumenta il tempo di un algoritmo al crescere della grandezza dell'input. Ci sono tre tipi di notazioni asintotiche:

- O grande
- $\Omega$  (Omega)
- $\Theta$  (Theta)

#### 1.1.1 O grande

##### Definizione di O grande

Date due funzioni  $f(n), g(n) \geq 0$  si dice che  $f(n)$  è in  $O(g(n))$  se:

$$\exists c, n_0 | f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$

**Esempio:**

$$f(n) = 3n + 3$$

$$f(n) = O(n^2) \text{ perché con } c = 6 \implies cn^2 \geq 3n + 3 \quad \forall n \geq 1$$

#### 1.1.2 Omega

##### Definizione di $\Omega$

Date due funzioni  $f(n), g(n) \geq 0$  si dice che  $f(n)$  è in  $\Omega(g(n))$  se:

$$\exists c, n_0 | f(n) \geq c \cdot g(n) \quad \forall n \geq n_0$$

**Esempio:**

$$f(n) = 2n^2 + 3$$

$$f(n) = \Omega(n) \text{ perché } f(n) \geq n \quad \forall n \geq 1$$

### 1.1.3 Theta

#### Definizione di $\Theta$

Date due funzioni  $f(n), g(n) \geq 0$  si dice che  $f(n)$  è in  $\Theta(g(n))$  se:

$$\exists c_1, c_2, n | c_1 \cdot g(n) \geq f(n) \geq c_2 \cdot g(n) \quad \forall n \geq n_0$$

**Esempio:**

$$f(n) = \log_a n = \Theta(\log_b n) \quad \forall a, b > 0$$

Dimostrazione:

$$\log_a n = \log_b n \cdot \log_a b = \log_b n \cdot c$$

## 1.2 Algebra della notazione asintotica

### 1.2.1 Gerarchia

$$c = O(\log(n))$$

$$\log(n) = O(\sqrt{n})$$

$$\sqrt{n} = O(n^k)$$

$$n^k = O(a^n)$$

$$a^n = O(n!)$$

$$n! = O(n^n)$$

Data una funzione  $f(n)$ , esistono infinite funzioni  $g(n)$  per cui  $f(n) = O(g(n))$  e infinite funzioni  $h(n)$  per cui  $f(n) = \Omega(h(n))$

Possiamo comparare facilmente due funzioni usando i limiti:

$$\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = \begin{cases} k > 0 \implies f(n) = \Theta(g(n)) \\ +\infty \implies f(n) = \Omega(g(n)) \\ 0 \implies f(n) = O(g(n)) \end{cases}$$

#### Regola sulle costanti moltiplicative

Per ogni  $k > 0$  se  $f(n) = O(g(n))$  allora anche  $k \cdot f(n) = O(g(n))$

Questa regola vale solo se  $k$  non è all'esponente.

**Dimostrazione:**

$$\exists c, n_0 | f(n) \leq c \cdot g(n) \quad \forall n \geq n_0 \implies k \cdot f(n) \leq k \cdot c \cdot g(n) \implies k \cdot f(n) = O(g(n))$$

#### Regola sulla commutatività con la somma

Se  $f(n) = O(g(n)) \wedge d(n) = O(h(n)) \implies f(n) + d(n) = O(g(n) + h(n)) = O(\max(g(n), h(n)))$

**Dimostrazione:**

$$f(n) = O(g(n)) \implies \exists c_1, n_1 | f(n) \leq c_1 \cdot g(n)$$

$$d(n) = O(h(n)) \implies \exists c_2, n_2 | d(n) \leq c_2 \cdot h(n)$$

$$f(n) + d(n) \leq c_1 \cdot g(n) + c_2 \cdot h(n) \implies c = \max(c_1, c_2) \implies f(n) + d(n) \leq$$

$$c \cdot g(n) + c \cdot h(n) \implies f(n) + d(n) = O(g(n) + h(n))$$

**Regola sulla commutatività col prodotto**

Se  $f(n) = O(g(n)) \wedge d(n) = O(h(n)) \implies f(n) \cdot d(n) = O(g(n) \cdot h(n))$

**Dimostrazione:**

$$f(n) = O(g(n)) \implies \exists c_1, n_1 | f(n) \leq c_1 \cdot g(n)$$

$$d(n) = O(h(n)) \implies \exists c_2, n_2 | d(n) \leq c_2 \cdot h(n)$$

$$f(n) \cdot d(n) \leq c_1 \cdot g(n) \cdot c_2 \cdot h(n) \implies c = \max(c_1, c_2) \implies f(n) \cdot d(n) \leq$$

$$c \cdot g(n) \cdot c \cdot h(n) \implies f(n) \cdot d(n) = O(g(n) \cdot h(n))$$

**1.2.2 Somme notevoli**

Questa è una lista contenente la maggior parte delle somme notevoli utili per questo corso:

$$\sum_{i=0}^n i = \theta(n^2) = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^n i^c = \theta(n^{c+1})$$

$$\sum_{i=0}^n 2^i = \theta(2^{n+1})$$

$$\sum_{i=0}^n c^i = \begin{cases} \frac{c^{n+1}-1}{c-1} & c > 1 \\ 1 & c \leq 1 \end{cases}$$

$$\sum_{i=0}^n i 2^i = \theta(n 2^n)$$

$$\sum_{i=0}^n i c^i = \theta(n c^n)$$

$$\sum_{i=0}^n \log i = \theta(\log n)$$

$$\sum_{i=0}^n \log^c i = \theta(n \log^c n)$$

$$\sum_{i=0}^n \frac{1}{i} = \theta(\log n)$$

# 2

## Costo computazionale

Possiamo calcolare il costo computazionale di un algoritmo usando il criterio del costo uniforme. Il costo computazionale è una funzione monotona crescente al crescere della dimensione dell'input. Visto che viene utilizzata la notazione asintotica il costo computazionale è calcolabile solo asintoticamente (cioè con input abbastanza grandi). Il costo di un algoritmo è la somma di tutte le istruzioni che lo compongono. Alcuni algoritmi potrebbero avere tempi di esecuzione diversi in base all'input, in questi casi devo calcolare a parità di dimensione di input il caso peggiore e il caso migliore. Se voglio scrivere il tempo di esecuzione a prescindere dall'input devo considerare il caso peggiore.

### 2.1 Costo delle istruzioni

Ogni tipo di istruzione ha un costo computazionale diverso in base a come viene eseguita.

#### 2.1.1 Istruzioni elementari

Le istruzioni elementari hanno costo  $\Theta(1)$  e sono:

- Operazioni aritmetiche
- Lettura di un valore da una variabile
- Assegnazione di un valore
- Condizione logica su un numero costante di operandi
- Stampa di un valore

Nel caso di codici con `if` il costo è il costo della verifica della condizione sommato al massimo tra il caso vero e il caso falso.

#### 2.1.2 Istruzioni iterative

Nel caso di cicli `for` o `while` il costo è pari alla somma del costo di ciascuna iterazione (contando anche la verifica della condizione). Ci sono due casi:

- Se il costo di ogni iterazione è uguale allora basta moltiplicare il costo di una iterazione per il numero di cicli
- Se il costo delle iterazioni è diverso allora bisogna sommare il costo delle iterazioni

Esempi:

---

**Algoritmo:** Massimo di un vettore di  $n$  numeri

---

```
def Trova_max(A):
    n=len(A)
    max=A[0]
    for i in range(1,n) : //  $n-1$  iterazioni
        if A[i]>max :
            max=A[i]
    return max
```

---

$$T(n) = (n - 1) \cdot \Theta(1) = \Theta(n)$$

---

**Algoritmo:** Somma dei primi  $n$  numeri interi

---

```
def Somma(n):
    somma=0
    for i in range(1,n+1) : //  $n$  iterazioni
        somma+=i
    return somma
```

---

$$T(n) = n \cdot \Theta(1) = \Theta(n)$$

---

**Algoritmo:** Valutazione di un polinomio

---

```
def Calcolo_polinomio(A,c):
    somma=A[0]
    for i in range(len(A)) : //  $n$  iterazioni
        potenza=1
        for j in range(i) : //  $i$  iterazioni
            potenza=c*potenza
            somma+=A[i]*potenza
    return somma
```

---

$$T(n) = n\Theta(i) = \sum_{i=1}^n \Theta(i) = \Theta(n^2)$$

### 2.1.3 Calcolo del costo computazionale

Nel caso in cui il caso peggiore ed il caso migliore hanno la stessa formula allora il costo computazionale è tale formula e si utilizza la notazione  $\Theta$ . Nel caso invece le due formule sono diverse allora il costo computazionale si può scrivere con la notazione  $O$  della formula che identifica il caso peggiore oppure posso calcolare il costo medio.

## 2.2 Tempi di esecuzione

Il tempo di esecuzione è calcolato moltiplicando il numero  $n$  di dati per il costo dell'algoritmo e dividendolo per il numero di operazioni al secondo.

$$T(n) = \frac{\text{algoritmo}(n)}{\text{operazioni}/s}$$

**Esempio:**

$10^9$  operazioni al secondo e  $n = 10^6$ :

- $O(n) = \frac{10^6}{10^9} = 10^{-3}s$
- $O(n \log n) = \frac{10^6 \cdot \log(10^6)}{10^9} = 2 \cdot 10^{-2}s$
- $O(n^2) = \frac{(10^6)^2}{10^9} = 10^3s$



# 3

## Algoritmi di Ricerca

Uno dei problemi principali dell'informatica è la ricerca di un elemento in un insieme di dati e per fare ciò si utilizza un algoritmo di ricerca.

L'algoritmo contiene:

- **Input:** un array di  $n$  elementi e un valore da trovare
- **Output:** un indice  $i$  tale che  $A[i]=\text{valore}$  o un valore `None` se il valore non è presente

### 3.1 Ricerca sequenziale

Questo algoritmo di ricerca scorre tutti i valori dell'array uno ad uno e li confronta con il valore, fermandosi quando lo si trova:

---

**Algoritmo:** Ricerca sequenziale

---

```
def cerca_v(A,v):  
    i=0  
    while i<len(A) :  
        if A[i]==v :  
            return i  
    return None
```

---

Caso peggiore:  $T(n) = \Theta(n)$

Caso migliore:  $T(n) = \Theta(1)$

**Costo medio:**

Visto che il caso migliore e peggiore hanno due formule diverse per calcolare il costo medio usiamo l'ipotesi che  $v$  si possa trovare in tutte le posizioni con la stessa possibilità, quindi la probabilità che  $v$  si trovi nella posizione  $i$  è  $\frac{1}{n}$

$$\text{Costo medio} = \frac{1}{n} \sum_{i=0}^n i = \frac{1}{n} \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Possiamo anche usare un'altra formula:

$$\text{Costo medio} = \sum_{i=0}^n i \cdot \frac{\text{numero di permutazioni in cui } v \text{ è nel punto } i}{\text{numero di permutazioni totali}} = \frac{n(n+1)}{2} \frac{(n-1)!}{n!} = \frac{n+1}{2}$$

### 3.2 Ricerca binaria

Questo algoritmo controlla l'elemento centrale dell'array (l'array deve essere ordinato) e lo confronta con il valore, se è più grande controlla la metà superiore, se è minore controlla la metà inferiore, ripetendo il procedimento:

---

**Algoritmo:** Ricerca binaria

---

```

def ricerca_binaria(A,v):
    a=0
    b=len(A)-1
    m=(a+b)/2
    while A[m]!=v :
        if A[m]>v :
            b=m-1
        else
            a=m-1
        if a>b :
            return -1
        m=(a+b)/2
    return m

```

---

Caso peggiore:  $T(n) = \Theta(\log n)$

Caso migliore:  $T(n) = \Theta(1)$

**Costo medio:**

Visto che il caso migliore e peggiore hanno due formule diverse per calcolare il costo medio usiamo l'ipotesi che  $v$  si possa trovare in tutte le posizioni con la stessa possibilità.

Dopo  $i$  iterazioni sono state controllate  $2^{i-1}$  posizioni, quindi la probabilità che il valore sia in una di queste è  $\frac{2^{i-1}}{n}$ .

$$\text{Costo medio} = \frac{1}{n} \sum_{i=1}^{\log n} i \cdot 2^{i-1} \implies \frac{1}{n} (\log n - 1) 2^{\log n} + 1 = \log n + \frac{1}{n} - 1$$

# 4

## Ricorsione

Un algoritmo è ricorsivo quando:

- è espresso in termini di se stesso
- la soluzione del problema è data dalla risoluzione di sotto-problemi di grandezza minore combinati insieme
- la successione dei sotto-problemi deve infine convergere ad un caso base che termina la ricorsione

Esempi:

---

**Algoritmo:** Fattoriale

---

```
def fattoriale(n):  
    if n==0 :  
        | return 1  
    return n*(fattoriale(n-1))
```

---

---

**Algoritmo:** Ricerca binaria ricorsiva

---

```
def ricerca_bin(A,v,i_min=0,i_max=len(A)):  
    if i_min>i_max :  
        | return None  
    m=(i_min+i_max)/2  
    if A[m]==v :  
        | return m  
    elif A[m]>v :  
        | return ricerca_binaria(A,v,i_min,m-1)  
    else  
        | return ricerca_binaria(A,v,m+1,i_max)
```

---

### 4.1 Iterazione vs Ricorsione

In diversi casi potrebbe essere migliore un tipo di algoritmo invece di un altro:

- **Ricorsivo:** se la formulazione della soluzione è aderente al problema stesso e quella iterativa è più complicata
- **Iterativo:** se la soluzione iterativa è evidente o se l'efficienza è importante

Questa distinzione si ha perché ogni funzione ha bisogno di una certa quantità di memoria e le funzioni ricorsive ne richiedono di più.

**Esempio:**

---

**Algoritmo:** Fibonacci iterativo

---

```
def fib(n):  
    if n ≤ 1 :  
        | return n  
    fib1=1  
    fib2=0  
    for i in range(2,n+1) :  
        | fib2=fib1  
        | fib1+=fib2  
    return fib1
```

---

$$T(n) = \Theta(n)$$

---

**Algoritmo:** Fibonacci ricorsivo

---

```
def fib(n):  
    if n ≤ 1 :  
        | return n  
    return fib(n-1)+fib(n-2)
```

---

Il numero di chiamate della funzione cresce molto velocemente perché molti calcoli vengono ripetuti più di una volta e quando si arriva al caso base ci sono una catena di chiamate ancora aperte. Per risolvere questo problema di dimensione  $n$  bisogna risolvere 2 sotto-problemi di dimensione  $n - 1$  e  $n - 2$ .

$$T(n) = T(n - 1) + T(n - 2) + \Theta(n)$$

il costo è quindi esponenziale ma per calcolarlo bisogna usare le equazioni di ricorrenza.

# 5

## Equazioni di ricorrenza

### 5.1 Scrittura di un'equazione di ricorrenza

Per calcolare il costo computazionale di un algoritmo ricorsivo ci si ritrova a dover risolvere una funzione ricorsiva, questa funzione si chiama equazione di ricorrenza.

**Esempio:**

---

**Algoritmo:** Fattoriale ricorsivo

---

```
def fact(n):  
    if n==0 :  
        | return 1  
    return n*fact(n-1)
```

---

Il costo computazionale di questo algoritmo è:

$$T = \begin{cases} T(n) = T(n-1) + \Theta(1) \\ T(0) = \Theta(1) \end{cases}$$

La parte generale deve essere composta dalla somma del costo computazionale non ricorsivo e dalla parte ricorsiva, ci deve inoltre essere almeno un caso base.

Per calcolare il costo computazionale di una equazione di ricorrenza ci sono 4 metodi:

- Metodo iterativo
- Metodo dell'albero
- Metodo di sostituzione
- Metodo principale

### 5.2 Metodo iterativo

Nel metodo iterativo si sviluppa l'equazione di ricorrenza per far si che sia una somma di termini dipendenti dal caso generico e dal caso base.

**Esempio:**

$$T = \begin{cases} T(n) = T(n-1) + \Theta(1) \\ T(0) = \Theta(1) \end{cases}$$

Se sviluppiamo  $T(n)$  come somma dei suoi sotto-termini:

$$T(n) = T(n-1) + \Theta(1) = T(n-2) + 2 \cdot \Theta(1) = T(n-k) + k \cdot \Theta(1)$$

La ricorsione continua finché  $n - k = 1 \implies k = n - 1$  e l'equazione quindi diventerà:

$$T(n) = T(n - k) + k \cdot \Theta(1) = T(n - n + 1) + (n - 1) \cdot \Theta(1) = T(1) + (n - 1) \cdot \Theta(1) = \Theta(n)$$

### 5.2.1 Caso particolare: sequanza di Fibonacci

Equazione di ricorrenza dell'n-esimo numero di Fibonacci:

$$T = \begin{cases} T(n) = T(n - 1) + T(n - 2) + \Theta(1) \\ T(1) = \Theta(1) \end{cases}$$

Se sviluppiamo l'equazione:

$$T(n) = T(n - 1) + T(n - 2) + \Theta(1) = T(n - 2) + 2T(n - 3) + T(n - 3) + 3\Theta(1) = \dots$$

Visto che non si può generalizzare il problema, cerchiamo di calcolare il costo  $O$  e il costo  $\Omega$

**Costo  $O$ :**

$$T(n) = T(n - 1) + T(n - 2) + \Theta(1) \leq T(n - 1) + T(n - 1) + \Theta(1) = T_1(n)$$

Quindi  $T_1(n)$ :

$$T_1 = \begin{cases} T_1(n) = 2T_1(n - 1) + \Theta(1) \\ T_1(1) = \Theta(1) \end{cases}$$

Sviluppando  $T_1(n)$ :

$$T_1(n) = 2T_1(n - 1) + \Theta(1) = 2[2T_1(n - 2) + \Theta(1)] + \Theta(1)$$

$$\text{Caso base} = n - k = 1 \implies k = n - 1$$

Generalizzando:

$$T_1(n) = 2^k T_1(n - k) + \sum_{i=0}^{k-1} 2^i \Theta(1) = 2^{n-1} T_1(1) + \sum_{i=0}^{n-2} 2^i \Theta(1) = \Theta(2^n) + (2^{n-1} - 1)\Theta(1) = \Theta(2^n)$$

$$\text{Quindi visto che } T(n) \leq T_1(n) = \Theta(2^n) \implies T(n) = O(2^n)$$

**Costo  $\Omega$ :**

$$T(n) = T(n - 1) + T(n - 2) + \Theta(1) \geq T(n - 2) + T(n - 2) + \Theta(1) = T_2(n)$$

Quindi  $T_2(n)$ :

$$T_2 = \begin{cases} T_2(n) = 2T_2(n - 2) + \Theta(1) \\ T_2(1) = \Theta(1) \end{cases}$$

Sviluppando  $T_2(n)$ :

$$T_2(n) = 2T_2(n - 2) + \Theta(1) = 2[2T_2(n - 4) + \Theta(1)] + \Theta(1)$$

$$\text{Caso base} = n - 2k = 1 \implies k = \frac{n}{2}$$

Generalizzando:

$$T_2(n) = 2^k T_2(n - 2k) + \sum_{i=0}^{k-1} 2^i \Theta(1) = 2^{\frac{n}{2}} T_2(1) + \sum_{i=0}^{\frac{n}{2}-1} 2^i \Theta(1) = \Theta(2^{\frac{n}{2}}) + (2^{\frac{n}{2}} - 1)\Theta(1) = \Theta(2^{\frac{n}{2}}) = \Theta(\sqrt{2^n})$$

$$\text{Quindi visto che } T(n) \geq T_2(n) = \Theta(\sqrt{2^n}) \implies T(n) = \Omega(\sqrt{2^n})$$

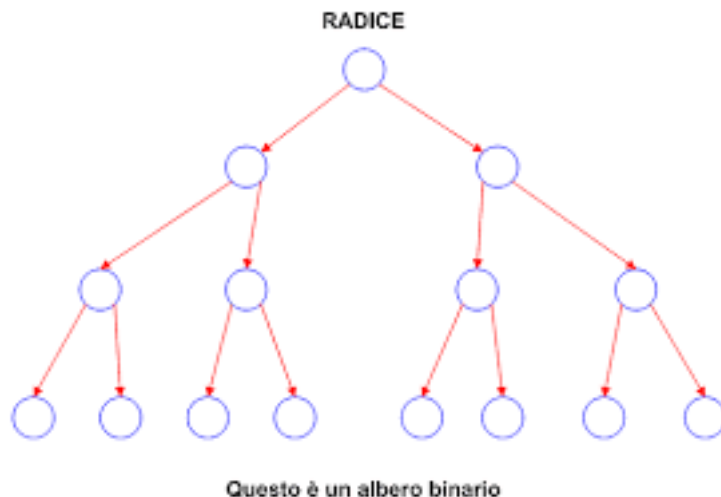
**Costo finale:**

$$\sqrt{2^n} \leq T(n) \leq 2^n$$

## 5.3 Metodo dell'albero

Nel metodo dell'albero si rappresenta graficamente lo sviluppo del costo computazionale attraverso un albero binario per calcolarlo più facilmente.

Un albero binario completo di altezza  $h$  è un albero in cui tutti i nodi hanno due figli tranne quelli nel livello  $h$  che non hanno figli (i nodi nell'ultimo livello si chiamano foglie).



Il numero di nodi nell'ultimo livello (foglie) è  $2^h$

Il numero di nodi all'interno dell'albero è  $\sum_{i=0}^h 2^i = 2^{h+1} - 1$

**Esempio:**

$$T = \begin{cases} T(n) = 2T(\frac{n}{2}) + \Theta(n^2) \\ T(1) = \Theta(1) \end{cases}$$

Il costo di ogni livello é:

Livello	n° nodi	Costo per nodo	Contributo per livello
0	$2^0$	$\Theta(n^2)$	$2^0 \cdot \Theta(n^2)$
1	$2^1$	$\Theta((\frac{n}{2})^2)$	$2^1 \cdot \Theta((\frac{n}{2})^2)$
2	$2^2$	$\Theta((\frac{n}{4})^2)$	$2^2 \cdot \Theta((\frac{n}{4})^2)$
...	...	...	...
$h = \log n$	$2^{\log n}$	$\Theta((\frac{n}{2^{\log n}})^2)$	$2^{\log n} \cdot \Theta((\frac{n}{2^{\log n}})^2)$

Quindi il costo totale dell'albero e quindi dell'equazione è:

$$T(n) = \sum_{k=0}^{\log n} 2^k \Theta((\frac{n}{2^k})^2) = \sum_{k=0}^{\log n} \frac{2^k}{2^{2k}} \Theta(n^2) = \Theta(n^2) \sum_{k=0}^{\log n} \frac{1}{2^k} = \Theta(n^2)$$

## 5.4 Metodo di sostituzione

Nel metodo di sostituzione si ipotizza una soluzione e poi si dimostra per induzione.

**Esempio:**

$$T = \begin{cases} T(n) = T(n-1) + \Theta(1) \\ T(1) = \Theta(1) \end{cases}$$

Ipotizziamo la soluzione:

$$\begin{cases} T(n) = T(n-1) + c \cdot \Theta(1) \\ T(1) = d \end{cases}$$

Con  $c$  e  $d$  valori casuali per cui l'equazione funziona.

Ipotizziamo che la soluzione sia  $\Theta(n)$  quindi dobbiamo dimostrare che sia  $O(n)$  e  $\Omega(n)$ .

### Dimostrare $O$ :

Ipotizziamo  $T(n) = O(n) \implies T(n) \leq k \cdot n$  dove  $k$  è una costante da determinare.

1. Caso base ( $n = 1$ ):  $\begin{cases} T(1) = d \\ T(1) \leq k \end{cases} \implies k \geq d$
2. Ipotesi induttiva:  $T(m) \leq km \forall m < n$
3. Dimostrazione:  

$$T(n) = T(n-1) + c \cdot \Theta(1) \implies T(n-1) \leq k(n-1) \implies T(n) \leq k(n-1) + c \implies$$

$$T(n) \leq kn - k + c \implies kn - k + c \leq kn \implies c \leq k$$
 Quindi:  

$$T(n) = O(n) \forall k \geq c$$

### Dimostrare $\Omega$ :

Ipotizziamo  $T(n) = \Omega(n) \implies T(n) \geq hn$  dove  $h$  è una costante da determinare.

1. Caso base ( $n = 1$ ):  $\begin{cases} T(1) = d \\ T(1) \geq h \end{cases} \implies d \geq h$
2. Ipotesi induttiva:  $T(m) \geq hm \forall m < n$
3. Dimostrazione:  

$$T(n) = T(n-1) + c \cdot \Theta(1) \implies T(n-1) \geq h(n-1) \implies T(n) \geq h(n-1) + c \implies$$

$$T(n) \geq hn - h + c \implies hn - h + c \geq hn \implies c \geq h$$
 Quindi:  

$$T(n) = \Omega(n) \forall h \leq c$$

## 5.5 Metodo principale

Nel metodo principale è un teorema che permette di risolvere le equazioni di ricorrenza.

Funziona solo se  $T(n) = aT(\frac{n}{b}) + f(n)$  e  $T(1) = \Theta(1)$



**Teorema principale**

Dati  $a \geq 1$  e  $b > 1$  una funzione di ricorrenza scritta come:

$$T = \begin{cases} T(n) = a \cdot T(\frac{n}{b}) + f(n) \\ T(1) = \Theta(1) \end{cases}$$

Ha diverse formule risolutive in base a  $f(n)$ :

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{se } f(n) = O(n^{\log_b(a)-\epsilon}) \\ \Theta(n^{\log_b a} \cdot \log n) & \text{se } f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & \text{se } f(n) = \Omega(n^{\log_b(a)+\epsilon}) \text{ e } a \cdot f(\frac{n}{b}) \leq c \cdot f(n) \end{cases}$$

Con  $\epsilon > 0$  e  $c < 1$ .

**Esempio caso 1:**

$$T(n) = 9T(\frac{n}{3}) + \Theta(n)$$

$$n^{\log_b a} = n^{\log_3 9} = n^2$$

$$f(n) = n^{\log_3 9 - \epsilon} \text{ con } \epsilon = 1 \implies T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$$

**Esempio caso 2:**

$$T(n) = T(\frac{2}{3}n) + \Theta(1)$$

$$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$$

$$f(n) = \Theta(n^{\log_b a}) \implies T(n) = \Theta(\log n)$$

**Esempio caso 3:**

$$T(n) = 3T(\frac{n}{4}) + \Theta(n \log n)$$

$$n^{\log_b a} = n^{\log_4 3} = n^{0,7}$$

$$f(n) = \Omega(n^{\log_4 3 + \epsilon}) \text{ con } \epsilon = 0,2 \text{ e } a \cdot f(\frac{n}{b}) = 3\Theta(\frac{n}{4} \log \frac{n}{4}) = \frac{3}{4}(n \log n - n \log 4) \leq c \cdot n \log n \text{ con } c = \frac{3}{4} \implies T(n) = \Theta(n \log n)$$

# 6

## Algoritmi di Sorting

Un **algoritmo di sorting** è un algoritmo che presa un array di elementi non ordinato lo ritorna in modo ordinato.

Vengono divisi in semplici:

- Insertion sort
- Selection sort
- Bubble sort

e complessi:

- Merge sort
- Heap sort
- Quick sort

### 6.1 Insertion sort

L'insertion sort estrae un elemento di indice  $i$  dall'array spostando verso destra tutti gli elementi maggiori per inserirlo nel posto giusto. Questo processo viene ripetuto per ogni elemento.

---

**Algoritmo:** Insertion sort

---

```
def insertion_sort(A):  
    for j in range(1, len(A)) :  
        x=A[j]  
        i=j-1  
        while i ≥ 0 and A[i]>x :  
            A[i+1]=A[i]  
            i-=1  
        A[i+1]=x  
    return A
```

---

**Costo computazionale:**

$$T(n) = \sum_{j=1}^{n-1} (\Theta(1) + t_j \Theta(1) + \Theta(1)) + \Theta(1)$$

Per ogni ciclo  $1 < t_j < j$  quindi:

- Caso migliore:  $T(n) = (n-1)\Theta(1) = \Theta(n)$
- Caso peggiore:  $T(n) = \sum_{j=1}^{n-1} (\Theta(1) + \Theta(j)) = \Theta(n^2)$

## 6.2 Selection sort

Il selection sort cerca il minimo all'interno dell'array, mettendolo in prima posizione per poi cercare il nuovo minore escludendo il primo per metterlo in seconda posizione e così via per ogni elemento.

---

**Algoritmo:** Selection sort

---

```
def selection_sort(A):
    for i in range(len(A)-1) :
        m=i
        for j in range(i+1,len(A)) :
            if A[j]<A[m] :
                m=j
        A[m],A[i]=A[i],A[m]
    return A
```

---

**Costo computazionale:**

$$T(n) = \sum_{i=0}^{n-2} (\Theta(1) + (n-i)\Theta(1) + \Theta(1)) + \Theta(1) = \Theta(n^2)$$

## 6.3 Bubble sort

Il bubble sort confronta da destra verso sinistra ogni coppia di valori, scambiandoli se non sono ordinati, questo processo viene eseguito tante volte quanti sono gli elementi dell'array.

---

**Algoritmo:** Bubble sort

---

```
def bubble_sort(A):
    for i in range(len(A)) :
        for j in range(len(A)-1,i,-1) :
            if A[j]<A[j-1] :
                A[j],A[j-1]=A[j-1],A[j]
    return A
```

---

**Costo computazionale:**

$$T(n) = \sum_{i=0}^{n-1} (\Theta(1) + (n-i)\Theta(1) + \Theta(1)) + \Theta(1) = \Theta(n^2)$$

## 6.4 Alberi di decisioni

Per stabilire un limite al costo computazionale sotto il quale non si può andare si usa uno strumento chiamato albero di decisione. Per algoritmi di ordinamento basati su confronti è un albero binario che rappresenta tutti i possibili confronti: i nodi interni rappresentano un confronto e i figli i possibili esiti del confronto.

Con un numero generico  $n$  di elementi, il numero delle foglie deve avere tutte le permutazioni possibili quindi  $n!$ .

Il numero massimo di foglie di un albero binario è  $2^h$  quindi:

$$2^h \geq n! \implies h \geq \log(n!) = \Theta(n \log n)$$

Con questo possiamo dire che il miglior algoritmo di sorting basato su confronti avrà costo:  $\Omega(n \log n)$

## 6.5 Merge sort

Il merge sort utilizza un algoritmo ricorsivo secondo una tecnica chiamata *divide et impera*, che consiste nel dividere il problema generale in sotto-problemi più piccoli che vengono risolti ricorsivamente e poi tutte le soluzioni vengono combinate.

---

**Algoritmo:** Merge sort

---

```
def merge_sort(A, id_inizio, id_fine):
    if id_inizio < id_fine :
        id_medio = (id_inizio + id_fine) / 2
        merge_sort(A, id_inizio, id_medio)
        merge_sort(A, id_medio + 1, id_fine)
        Fondi(A, id_inizio, id_medio, id_fine)
    return A
```

---

**Costo computazionale:**

$$T(n) = \Theta(1) + 2T\left(\frac{n}{2}\right) + S(n)$$

$S(n)$  è il costo di Fondi che è una funzione che sfrutta il fatto che le due sottosequenze sono ordinate, quindi il minimo sarà il più piccolo tra i due minimi e si continua eliminando di volta in volta i minimi usati.

**Algoritmo:** Fondi

---

```

def Fondi(A,id_inizio,id_medio,id_fine):
    i,j=id_medio,id_medio+1
    B=[]
    while i≤id_medio and j≤id_fine :
        if A[i]≤A[j] :
            B.append(A[i])
            i+=1
        else
            B.append(A[j])
            j+=1
    while i≤id_medio :
        B.append(A[i])
        i+=1
    while j≤id_fine :
        B.append(A[j])
        j+=1
    for i in range(len(B)) :
        A[id_inizio+i]=B[i]
    return B

```

---

**Costo computazionale:**

$$S(n) = \Theta(n)$$

Quindi il costo computazionale del merge sort in totale è un'equazione di ricorrenza:

$$T = \begin{cases} T(n) = 2T(\frac{n}{2}) + \Theta(n) \\ T(1) = \Theta(1) \end{cases} = \Theta(n \log n)$$

**6.6 Quick sort**