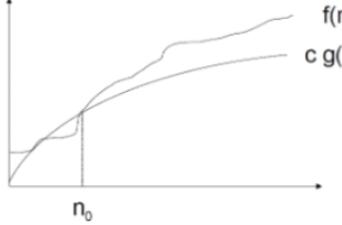
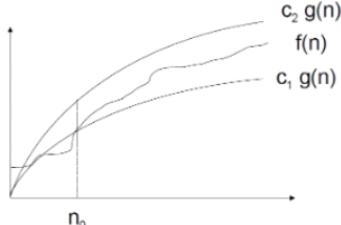


# Asintoti

martedì 28 gennaio 2025 15:00

Date due funzioni  $f(n), g(n) \geq 0$

Notazione O grande ( $O$ )	Notazione Omega ( $\Omega$ )	Notazione Theta ( $\Theta$ )
<b>Limite Asintotico Superiore</b> $f(n)$ è in $O(g(n))$ se $\exists c, n_0$ (costanti) t.c. $0 \leq f(n) \leq c*g(n) \forall n \geq n_0$ Determinare la più piccola $g(n)$ per $O$  Sia $f(n)$ un polinomio di grado $m$ , allora possiamo dire che $f(n)$ è in $O(n^m)$ <b>Se <math>f(n)</math>:</b> <ul style="list-style-type: none"> <li>• polinomio grado <math>m = O(n^m)</math></li> <li>• <math>\log^a(n) = O(n^{1/b}) \forall a, b \geq 1</math></li> <li>• <math>n^{1/a}(\sqrt[b]{n}) = O(n^b) \forall a, b \geq 1</math></li> <li>• <math>n^a = O(b^n) \forall a \geq 1, b \geq 2</math></li> </ul> <b>Esempio</b> $f(n) = 5n^2 + 3n + 7$ , allora $f(n)$ è in $O(n^2)$ quindi $\leq c*n^2$ per $c = 11$ e $n_0 = 1$ , cioè $11n^2 \geq 5n^2 + 3n + 7 \forall n \geq 1$	<b>Limite Asintotico Inferiore</b> $f(n)$ è in $\Omega(g(n))$ se $\exists c, n_0$ (costanti) t.c. $f(n) \geq c*g(n) \forall n \geq n_0$ Determinare la più grande $g(n)$ per $\Omega$  Sia $f(n)$ un polinomio di grado $m$ , allora possiamo dire che $f(n)$ è in $\Omega(n^m)$ <b>Se <math>f(n)</math>:</b> <ul style="list-style-type: none"> <li>• polinomio grado <math>m = \Omega(n^m)</math></li> <li>• <math>n^{1/b}(\sqrt[b]{n}) = \Omega(\log^a(n)) \forall a, b \geq 1</math></li> <li>• <math>b^n = \Omega(n^a) \forall a \geq 1, b \geq 2</math></li> </ul> <b>Esempio</b> $f(n) = 5n^2 + 3n + 7$ , allora $f(n)$ è in $\Omega(n^2)$ quindi $\geq c*n^2$ per ogni $n$ , se $c \leq 2$ poiché $5n^2$ è il termine dominante che garantisce che $f(n)$ cresca almeno quanto $n^2$	<b>Limite Asintotico Stretto</b> $f(n)$ è in $\Theta(g(n))$ se $\exists c, n_0$ (costanti) t.c. $0 \leq f(n) \leq c*g(n) \forall n \geq n_0$ Cioè, $f(n)$ è in $\Theta(g(n))$ se: $\Omega(g(n)) \leq f(n) \leq O(g(n))$ Determinare la più piccola $g(n)$ per $\Theta$  Sia $f(n)$ un polinomio di grado $m$ , allora possiamo dire che $f(n)$ è in $O(n^m)$ <b>Esempio</b> $f(n) = 5n^2 + 3n + 7$ , $f(n)$ è in $\Theta(n^2)$ , perché per $n$ sufficientemente grande, $n^2$ descrive esattamente la crescita di $f(n)$ , ignorando le costanti e i termini meno dominanti

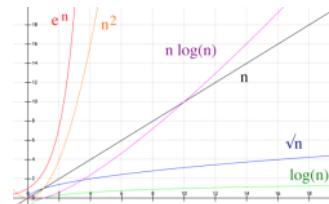
## Ordini di Grandezza

La scala di O grandi e Omega segue la scala degli ordini di grandezza delle successioni numeriche (per  $n \rightarrow \infty$ )

$$k < \log_a(n) < \sqrt[b]{n} (n^{1/b}) < n^c < k^n < n! < n^n (< = \text{precede})$$

Es.

- $f(n) = \log(n) \Rightarrow f(n)$  è in  $O(\sqrt{n})$  e  $\Omega(1)$
- $f(n) = n^a \Rightarrow f(n)$  è in  $O(2^n)$  e  $\Omega(\sqrt[n]{n})$



## Calcolo Notazione tramite Limiti

Se  $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)}$  è uguale a:

- $k > 0 \Rightarrow f(n) = \Theta(g(n))$
- $\infty \Rightarrow f(n) = \Omega(g(n))$  ma  $f(n) \neq \Theta(g(n))$
- $0 \Rightarrow f(n) = O(g(n))$  ma  $f(n) \neq \Theta(g(n))$

Ricordarsi gli ordini di grandezza per vedere quale funzione cresce più rapidamente dell'altra

Se il limite non esiste, allora non possiamo usare il metodo. Siccome le funzioni (tempi di esecuzione) sono tutti positivi, i limiti sono tutti non negativi

### Esempi

- 1)  $f(n) = n^2, g(n) = 2^n$ . Limite:  $\lim_{n \rightarrow \infty} \frac{n^2}{2^n}$ . Man mano che  $n$  aumenta,  $2^n$  cresce più rapidamente di  $n^2$ , quindi il limite tende a 0. Pertanto,  $n^2 = O(2^n)$
- 2)  $f(n) = 3n^2 + 5n + 4, g(n) = 8n + 2$ . Limite:  $\lim_{n \rightarrow \infty} \frac{3n^2 + 5n + 4}{8n + 2} = \frac{3n^2(1 + \frac{5}{n} + \frac{4}{n^2})}{8n(1 + \frac{2}{n})} = \frac{3n^2}{8n} = \frac{3n}{8}$ . Poiché  $3n$  cresce più rapidamente di una costante (8), abbiamo  $g(n) = \Omega(f(n))$
- 3)  $f(n) = 17n^2 + 3n + 9, g(n) = 8n^2 + 3$ . Limite:  $\lim_{n \rightarrow \infty} \frac{17n^2 + 3n + 9}{8n^2 + 3} = \frac{17n^2(1 + \frac{3}{n} + \frac{9}{n^2})}{8n^2(1 + \frac{3}{n^2})} = \frac{17n^2}{8n^2} = \frac{17}{8}$ . Poiché  $17/8$  è una costante maggiore di 0, abbiamo  $f(n) = \Theta(g(n))$
- 4)  $f(n) = n^{0.5}, g(n) = \log(n)$ . Limite:  $\lim_{n \rightarrow \infty} \frac{n^{0.5}}{\log(n)} = \frac{\frac{1}{2}n^{-0.5}}{\log(n)} = \frac{\sqrt{n}}{\log(n)}$ . Abbiamo che  $\sqrt{n}$  cresce più velocemente di  $\log(n)$ , quindi il limite tende a  $\infty$ . Pertanto  $\log(n) = \Omega(\sqrt{n})$

## Algebra della Notazione Asintotica

(Usa X per indicare qualsiasi limite asintotico ( $O, \Omega, \Theta$ ))

Per semplificare il calcolo del costo computazionale si possono usare tre regole algebriche:

- **Regola delle costanti moltiplicative:** le costanti moltiplicative si possono ignorare (a meno che non sia nell'esponente (es.  $O(2^{kn})$ ))  
 se  $f(n)$  è in  $X(g(n)) \Rightarrow k*f(n)$  è in  $X(g(n))$   
 Esempio:  
 $f(n) = 5n^2 \Rightarrow f(n)$  è in  $X(n^2)$
- **Regola della commutatività con somma:** il limite asintotico di una somma di funzioni è determinato dal termine con la crescita più veloce (massimo)  
 se  $f(n) = p(n) + q(n) \Rightarrow f(n)$  è in  $X(\max(p(n), q(n)))$   
 Esempio:  
 $f(n) = 5n^2 + n \Rightarrow f(n)$  è in  $X(n^2)$
- **Regola della commutatività con prodotto:** il limite asintotico di un prodotto di funzioni è determinato dal limite asintotico del prodotto delle singole funzioni  
 se  $f(n) = p(n)*q(n) \Rightarrow f(n)$  è in  $X(p(n)*q(n))$   
 Esempio:  
 $f(n) = n*\log(n) \Rightarrow f(n)$  è in  $X(n*\log(n))$

### Esempi

Esempi del prof sulla ricerca del limite asintotico stretto

$$1) f(n) = 3n^2 + 7 \rightarrow 3n^2 = \Theta(n^2) \text{ e } 7 = \Theta(1) = O(n^2), \text{ quindi } 3n^2 + 7 = \Theta(n^2)$$

- 2)  $f(n) = 3n2^n + 4n^4 \rightarrow \Theta(n)*\Theta(2^n) + \Theta(n^4) = \Theta(n2^n)$   
 3)  $f(n) = 2^{n+1} \rightarrow 2^{n+1} = 2^*2^n = \Theta(2^n)$   
 4)  $f(n) = 2^{2n} \rightarrow 2^{2n} = 2^n * 2^n = \Theta(2^n)*\Theta(2^n) = \Theta(2^{2n})$  (le costanti moltiplicative non si possono ignorare se sono nell'esponente)  
 5)  $f(n) = 5*2^{\log(n)} + 13 \rightarrow 5*2^{\log(n)} = 5^*n = \Theta(n)$  e  $13 = \Theta(1)$ , quindi  $\Theta(n) + \Theta(1) = \Theta(n)$   
 6)  $f(n) = \log_n(n) + 8*2^{\log(n)} + 3 \rightarrow \log_n(n) = \Theta(\log_n(n)), 8*2^{\log(n)} = 8^*n^n = \Theta(n^n)$  e  $3 = \Theta(1)$ , quindi  $\Theta(\log_n(n)) + \Theta(n^n) + \Theta(1) = \Theta(n^n)$

### Esercizi per Casa

Calcolare andamento asintotico delle funzioni:

- $f(n) = n^2 * \log(n) \rightarrow \Theta(n^2)*\Theta(\log(n)) = \Theta(n^2*\log(n))$
- $f(n) = 3n * \log(n) + 2n^2 \rightarrow \Theta(n)*\Theta(\log(n)) + \Theta(n^2) = \Theta(n^2)$  poiché  $n^2$  cresce più velocemente di  $n * \log(n)$
- $f(n) = 2^{\log(n/2)} + 5n \rightarrow 2^{\log(n)-\log(2)} + 5n = 2^{\log(n)-1} + 5n = \frac{n}{2} + 5n = \Theta(n) + \Theta(n) = \Theta(n)$
- $f(n) = 4^{\log(n)} \rightarrow (2^2)^{\log(n)} = 2^{2*\log(n)} = n^2 = \Theta(n^2)$
- $f(n) = (\sqrt{2})^{\log(n)} \rightarrow (2^{1/2})^{\log(n)} = 2^{\frac{1}{2}*\log(n)} = n^{\frac{1}{2}} = \sqrt{n} = \Theta(\sqrt{n})$

Classifica le seguenti funzioni per ordine di crescita. Trova ordinamento  $g_1, g_2, \dots, g_n$  che soddisfi  $g_1 = \Omega(g_2), g_2 = \Omega(g_3), \dots$

Partiziona poi la lista in classi di equivalenza in modo che le funzioni  $f(n)$  e  $g(n)$  sono nella stessa classe se e solo se  $f(n) = \Theta(g(n))$

$$1) (\sqrt[n]{n})^{\log(n)} = \left(n^{\frac{1}{n}}\right)^{\log(n)} = n^{\frac{1}{n}\log(n)} = n^{\log(\sqrt[n]{n})}$$

$$2) n^2$$

$$3) n!$$

$$4) (\log(n))!$$

$$5) \log(n^n)$$

$$6) n^3$$

$$7) \log(n!)$$

$$8) 2^n$$

$$9) n * 2^n$$

$$10) \log(n)$$

$$11) n * \log(n)$$

$$12) (3/2)^n$$

$$13) e^n$$

$$14) 5$$

$$15) 4^{\log(n)} = (2^2)^{\log(n)} = 2^{2\log(n)} = n^2$$

$$16) 2/3$$

$$17) \log^2(n)$$

Ordinamento:

$$2/3 \rightarrow 5 \rightarrow \log(n) \rightarrow \log^2(n) \rightarrow \log(n!) \rightarrow \log(n^n) \rightarrow n * \log(n) \rightarrow n^2 \rightarrow 4^{\log(n)} \rightarrow n^3 \rightarrow (3/2)^n \rightarrow e^n \rightarrow 2^n \rightarrow n * 2^n \rightarrow (\log(n))! \rightarrow n! \rightarrow (\sqrt[n]{n})^{\log(n)}$$

# Costo Computazionale

martedì 28 gennaio 2025 16:29

Il calcolo del **costo computazionale** dipende dalla **quantità di dati in input**, quindi bisogna trovare il **parametro** corrispondente:

- In un **algoritmo di ordinamento** sarà il **num. di dati** da ordinare
- In un **algoritmo che lavora su una matrice** sarà il num. di righe e colonne
- In un **algoritmo che opera su alberi** sarà il num. di nodi

## Costo Istruzioni

Ci sono **3 tipi di istr.**:

Istr. Elementari	Blocchi if/else	Blocchi iterativi
<p>Costo = <math>\Theta(1)</math> Non dipendono dalla dim. di input (es. op. aritm, read/write var.) Hanno <b>tempo costante</b> quindi costo <math>\Theta(1)</math></p> <p><b>Esempio:</b> <code>var = 10 Θ(1) var += 10 * 10 Θ(1) + Θ(1) + Θ(1) = Θ(1) print("Il valore di var è: ", var) Θ(1) costo tot = Θ(1)+Θ(1)+Θ(1) = Θ(1)</code></p>	<p>Costo = costo condizione + max(Costolif, CostoElse)</p> <p><b>Esempio:</b> <code>if(a &gt; b): condizione = Θ(1)     a += b Θ(1)     print("Il valore di a+b è", a) Θ(1)     blocco if = Θ(1) else:     print("Il valore di a è", a) Θ(1)     blocco else = Θ(1)</code></p> <p>costo tot = condizione + max(Costolif, CostoElse) = <math>\Theta(1) + \max(\Theta(1), \Theta(1)) = \Theta(1)</math></p>	<p>Costo = N. iter. * Costo Iter. + ultima iter. di verifica</p> <p><b>Esempio:</b> <code>for i in range(len(A)):     # n iter. + Θ(1) dell'ultima verifica     sum += A[i] Θ(1)</code></p> <p>costo totale = NumIter * CostoIter + Ultimalter = <math>n * \Theta(1) + \Theta(1) = \Theta(n)</math></p>

Un algoritmo può avere tempi di esecuzione diversi in base all'input (**caso migliore o peggiore**)

Per essere più precisi sul calcolo useremo  $\Theta$ , laddove non sia possibile **approssimiamo per difetto ( $\Omega$ ) o eccesso ( $O$ )**

## Ricerca

Un problema ricorrente è la **ricerca di un elemento in un insieme di dati**.

- **Input:** array **A** di  $n$  elem. e un **val. v** da cercare al suo interno
- **Output:** l'**indice** corrispondente alla posizione dell'elem. **v** trovato in **A**, altrimenti -1 se non viene trovato

Abbiamo due tipi di ricerca principali

Ricerca Sequenziale	Ricerca Binaria
<ul style="list-style-type: none"><li>• si scorre sequenzialmente l'array</li><li>• Confronta ogni el. con v</li><li>• Se l'el. = v, restituisco l'indice</li><li>• Se arriva fino alla fine se non lo trova, restituisce -1</li></ul>	<ul style="list-style-type: none"><li>• si considerà l'el. <b>centrale m</b></li><li>• se <math>v = m</math>, restituisco l'indice</li><li>• se <math>v &lt; m</math>, ripeto la ricerca <b>nella metà inferiore</b></li><li>• se <math>v &gt; m</math>, ripeto la ricerca <b>nella metà superiore</b></li><li>• ripetiamo fino a ridurre l'array ad un singolo el. o finché non trovo v</li></ul>
<p><b>Analisi</b></p> <ul style="list-style-type: none"><li>• <b>Caso migliore:</b> v è il <b>primo elem. (<math>\Theta(1)</math>)</b></li><li>• <b>Caso peggiore:</b> v non è presente, si analizza comunque tutto l'array (<math>\Theta(n)</math>)</li><li>• <b>Caso medio:</b> con una distribuzione uniforme, il ciclo in media si esegue <math>\frac{n+1}{2}</math> volte (<math>\Theta(n)</math>)</li></ul> <p><b>Caso peggiore e medio:</b> <math>O(n)</math></p> <pre>def Ricerca_Sequenziale(A, v):     for i in range(len(A)):         if A[i] == v:             return i     return -1</pre>	<p><b>Analisi</b></p> <ul style="list-style-type: none"><li>• <b>Caso migliore:</b> v è al centro (<math>\Theta(1)</math>)</li><li>• <b>Caso peggiore:</b> l'array viene dimezzato fino a un singolo elem. (<math>\Theta(\log(n))</math>)</li><li>• <b>Caso medio:</b> simile al caso peggiore, poiché il num. di iter. è legato al dimezzamento progressivo dell'array</li></ul> <p><b>Caso peggiore e medio:</b> <math>O(\log(n))</math> (meglio del sequenziale)</p> <pre>def Ricerca_Binaria(A, v):     a = 0 # il primo indice di A     b = len(A) - 1 # l'ultimo indice di A     m = (a+b)//2 # l'indice a metà di A     while A[m] != v: # finché non trovo v         if A[m] &gt; v: # se v è minore di A[m]             b = m - 1 # prendo la metà inferiore         else: # se v è maggiore di A[m]             a = m + 1 # prendo la metà superiore         if a &gt; b: # se v non è in A             return -1         m = (a+b)//2 # ricalcolo il nuovo valore di m     return m</pre>

## Ricorsione

Un algoritmo è detto **ricorsivo** quando è espresso in **termini di se stesso**. Esso ha sempre queste proprietà:

- **Parte ricorsiva:** la soluzione è costruita risolvendo (ricorsivamente) **uno o più sottoproblemi** di dim. minore, combinando poi le soluzioni
- **Caso base:** La successione dei **sottoproblemi** (sempre più piccoli) deve convergere ad **sottoproblema** che sia un **caso base** in cui la **ricorsione termina**

## Esempi

```

def Ricerca_Binaria_seq(A, v):
    if len(A) == 0: # se A è vuoto
        return -1
    m = len(A)//2 # calcolo l'indice a metà di A
    if A[m] == v: # se v sta a metà
        return m
    if v < A[m]: # se v si trova nella metà inferiore
        return Ricerca_Binaria_seq(A[:m], v)
    else: # se v si trova nella metà superiore
        return Ricerca_Binaria_seq(A[m+1:], v)

```

#### Calcolo del fattoriale

```

def Fattoriale(n):
    if(n == 0):
        return 1
    return n*Fattoriale(n-1)

```

$n! = \begin{cases} ricorsiva: n * (n - 1)! \\ caso base: 0! = 1 \end{cases}$   
 Costo:  
 $T = \begin{cases} T(n) = n * T(n - 1) + \Theta(1) \\ T(0) = 1 \end{cases}$

#### Numero di Fibonacci

```

def Fib(n):
    if(n ≤ 1):
        return n
    return Fib(n-1)+Fib(n-2)

```

Costo:  
 $T = \begin{cases} T(n) = T(n - 1) + T(n - 2) + \Theta(1) \\ T(1) = T(0) = 1 \end{cases}$

# Equazioni di Ricorrenza

mercoledì 29 gennaio 2025 11:07

Trovare la **funzione del costo computazionale** è immediato ma deve essere risolta per essere rappresentata con la **notazione asintotica**. Una equazione di ricorrenza  $T(n)$  deve essere costituita da almeno **2 addendi**:

- **La parte ricorsiva**
- **Caso base:** rappresentante il **costo computazionale** di ciò che avviene nella funzione

Ad Esempio:

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

## Riassunto dei passaggi per applicare i metodi

Gli esempi utilizzano questo tipo di eq. di ricorrenza, però sono esempi quindi non sono tutti i casi

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

### Metodo Iterativo

Espandi ricorsivamente l'eq. fino a riconoscere un pattern e scrivere la sol. in forma di sommatoria.

#### Passaggi generali:

- **Expansione:** Sostituisci ripetutamente la chiamata ricorsiva col passo successivo.

$$T(n) = a \left[ aT\left(\frac{n}{b^2}\right) + f\left(\frac{n}{b}\right) \right] + f(n) = a^2T\left(\frac{n}{b^2}\right) + af\left(\frac{n}{b}\right) + f(n)$$

- **Generalizzazione:** Generalizza per  $k$  liv. se si trova un pattern

$$T(n) = a^kT\left(\frac{n}{b^k}\right) + \sum_{i=0}^{k-1} a^i f\left(\frac{n}{b^i}\right)$$

- **Troviamo quando finisce la relazione:** Trovare quando  $k$  arriva al caso base.

Spesso:  $\frac{n}{b^k} = 1 \Rightarrow k = \log_b(n)$

- **Calcola la sommatoria:** Sostituisci  $k$  nella formula generale e usa formule note (es. somma geometrica) per trovare l'ordine di grandezza

### Metodo dell'Albero

Disegna un albero in cui ogni nodo rappresenta il costo in un particolare passo della ricorsione, e somma i costi a ogni livello per ottenere il costo totale.

#### Passaggi generali:

- **Rappresentazione ad albero:** ogni nodo è un istanza del problema  
Nel nostro esempio il nodo radice ha costo  $f(n)$  e ha  $a$  figli, ognuno con dimensione  $n/b$

- **Num. liv., costo per nodo e per livello:**

- Ad ogni liv. i avremmo  $a^i$  nodi
- Il costo di un nodo al liv. i è  $f\left(\frac{n}{b^i}\right)$
- Il costo del liv. i sarà  $a^i * f\left(\frac{n}{b^i}\right)$

- **Troviamo quando finisce la relazione:** trovare quando  $k$  arriva al caso base.

Spesso:  $\frac{n}{b^k} = 1 \Rightarrow k = \log_b(n)$

- **Somma totale:** Somma il costo di tutti i liv. fino a raggiungere il caso base e poi, come nel metodo iterativo, trova l'ordine di grandezza.

$$\sum_{i=0}^k a^i f\left(\frac{n}{b^i}\right)$$

### Metodo della Sostituzione

Fai un'ipotesi legittima su una soluzione per  $T(n)$  e dimostrala per induzione

#### Passaggi generali:

- **Fai un'ipotesi:** trova una ipotesi t.c  $T(n) \leq c^*g(n)$  ( $O(g(n))$ ) e  $T(n) \geq c^*g(n)$  ( $\Omega(g(n))$ ) per una funz  $g(n)$  e una costante  $c$ .

- **Eseguiamo due verifiche:** Verifichiamo per sostituzione le due ipotesi

- **Sostituzione:** Sostituisci l'ipotesi nel caso base e nel caso generale.

Es. per verificare  $T(n) \leq O(g(n))$

$$T(n) \leq a * \left( cg\left(\frac{n}{b}\right) \right) + f(n)$$

- **Verifica e aggiusta:** Dimostra se la diseguaglianza è vera per un val. di  $c$ , se necessario aggiusta l'ipotesi (aggiungendo termini addizionali come costanti extra o termini minori)

- Se entrambi i casi  $T(n) = O(g(n))$  e  $T(n) = \Omega(g(n))$  allora  $T(n) = \Theta(g(n))$

### Metodo Principale

Il **metodo principale** fornisce una sol. diretta per ricorrenze con la forma uguale a quella dell'esempio. dove  $a \geq 1$  e  $b > 1$ . Funziona solo se

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \text{ e } T(1) = \Theta(1)$$

#### Calcola $n^{\log_b(a)}$ e confronta con $f(n)$ :

- Ci sono tre casi
- **Caso 1:** Se  $f(n) = O(n^{\log_b(a)-\epsilon})$  per  $\epsilon > 0$   
 $(f(n))$  viene dominato da  $n^{\log_b(a)}$ , allora  
 $T(n) = \Theta(n^{\log_b(a)})$
  - **Caso 2:** Se  $f(n) = \Theta(n^{\log_b(a)})$   
 $(f(n))$  ha lo stesso costo di  $n^{\log_b(a)}$   
 $T(n) = \Theta(n^{\log_b(a)} * \log(n))$
  - **Caso 3:** Se  $f(n) = \Omega(n^{\log_b(a)+\epsilon})$  per  $\epsilon > 0$  e  $a*f\left(\frac{n}{b}\right) \leq c^*f(n)$  per  $c < 1$  e  $n$  grande  
 $(f(n))$  domina  $n^{\log_b(a)}$ , allora  
 $T(n) = \Theta(f(n))$

## Inapplicabilità dei metodi

### Metodo Iterativo

**Non applicabile** quando l'espansione della ricorrenza **non porta a un pattern chiaro** o una **formula chiusa facilmente identificabile**.

#### Problemi tipici non applicabili:

- Eq. con termini non facilmente esprimibili in funzione di  $n$  (es. dipendenze complesse da più val. precedenti)
- Ricorrenze **non omogenee** con termini variabili (es.  $T(n) = T(n-1) + T(n-2) + n$ )
- Se il num. di iter. richiesto per arrivare alla base è difficile da determinare

### Metodo dell'Albero

**Non applicabile** quando la struttura della ricorrenza **non si presta** ad una **suddivisione chiara** o quando il num. di chiamate ricorsive **non segue un modello semplice**.

#### Problemi tipici non applicabili:

- Se la funz. di ricorrenza **non si suddivide** in problemi di dim. costante
- Se il num. di sottoproblemi **cambia** in modo **non prevedibile**
- Se i costi delle chiamate ricorsive **non sono facilmente sommabili** in una forma chiusa

### Metodo della Sostituzione

**Non applicabile** quando non si riesce a indovinare una **buona sol. iniziale** per la verifica per induzione.

#### Problemi tipici non applicabili:

- Se la sol. dipende da una **sommatoria difficile da gestire**
- Se la verifica per induzione porta a passaggi algebrici **molto complessi** senza una chiara convergenza
- Se la ricorrenza ha un **comportamento irregolare** che rende difficile formulare un'ipotesi plausibile

### Metodo Principale

**Non applicabile** quando la ricorrenza **non è nella forma standard**  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

#### Problemi tipici non applicabili:

- Se  $a$  o  $b$  **non sono costanti** (dipendono da  $n$ )
- Se  $f(n)$  **non è una funz. polinomiale** o **non può essere confrontata direttamente con  $n^{\log_b(a)}$**
- Se la suddivisione del problema **non è uniforme** (es.  $T(n) = T(n-1) + T\left(\frac{n}{2}\right) + \Theta(1)$ )

Esistono **diversi metodi** per risolvere un'equazione di ricorrenza

### Metodo Iterativo

Sviluppa l'equazione e si **esprime come somma di termini dipendenti dall'input e dal caso base**, questo porterà inevitabilmente una maggiore quantità di calcoli algebrici rispetto ad altri metodi.

**Non applicabile** quando l'espansione della ricorrenza non porta a un pattern chiaro o una formula chiusa facilmente identificabile.

**Problemi tipici non applicabili:**

- Eq. con termini non facilmente esprimibili in funzione di n (es. dipendenze complesse da più val. precedenti)
- Ricorrenze **non omogenee** con termini variabili (es.  $T(n) = T(n-1) + T(n-2) + n$ )
- Se il num. di iter. richiesto per arrivare alla base è difficile da determinare

Esempio di applicazione:

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

Sviluppando  $T(n)$  come **somma dei sotto-termimi**:

- 1)  $T(n-1) + \Theta(1)$
- 2)  $(T(n-2) + \Theta(1)) + \Theta(1) = T(n-2) + 2 * \Theta(1)$
- 3)  $(T(n-3) + \Theta(1)) + \Theta(1) = T(n-3) + 3 * \Theta(1)$
- 4) ...
- 5)  $T(n-k) + k * \Theta(1)$

La ricorsione continua finché  $n-k = 1 \Rightarrow k = n-1$ , e l'eq. diventerà:  
 $T(n) = T(n-k) + k * \Theta(1) = T(n-n+1) + (n-1) * \Theta(1) = T(1) + (n-1) * \Theta(1) = \Theta(1) + \Theta(n) = \Theta(n)$

Altri esempi

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

Sviluppando  $T(n)$  otteniamo che

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + \Theta(1) = 2\left(2T\left(\frac{n}{4}\right) + \Theta(1)\right) + \Theta(1) = \\ &= 2\left(2\left(2T\left(\frac{n}{8}\right) + \Theta(1)\right) + \Theta(1)\right) + \Theta(1) = \dots \end{aligned}$$

Generalizzando l'equazione, otteniamo

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 2^i \cdot \Theta(1)$$

Sappiamo che il caso base è  $T(1)$  e che viene raggiunto quando

$$\frac{n}{2^k} = 1 \Rightarrow k = \log_2(n)$$

Dunque ne segue che

$$\begin{aligned} T(n) &= 2^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 2^i \cdot \Theta(1) = 2^{\log_2(n)} \cdot T(1) + \sum_{i=0}^{\log_2(n)-1} 2^i \cdot \Theta(1) = \\ &= \Theta(n) + \Theta\left(\frac{2^{\log_2(n)} - 1}{2 - 1}\right) = \Theta(n) \end{aligned}$$

## Metodo dell'Albero

Il **metodo dell'albero** è uguale al metodo iterativo nel calcolo ma si **rappresenta graficamente** lo sviluppo del costo dell'algoritmo usando un **albero binario** per valutarlo più facilmente.

Un albero completo di **altezza h** è un albero in cui tutti i nodi hanno

**due figli** tranne quelli nel livello h (foglie)

- Num. di foglie è  $2^h$
- Num di nodi interni è  $\sum_{i=0}^{h-1} 2^i = \frac{2^h - 1}{2 - 1} = 2^h - 1$
- Num tot. dei nodi è  $n = 2^{h+1} - 1$
- Di conseguenza,  $h = \log_2(\frac{n+1}{2})$

**Non applicabile** quando la struttura della ricorrenza non si presta ad una **suddivisione chiara** o quando il num. di chiamate ricorsive **non segue un modello semplice**.

**Problemi tipici non applicabili:**

- Se la funz. di ricorrenza **non si suddivide** in problemi di dim. costante
- Se il num. di sottoproblemi **cambia** in modo **non prevedibile**
- Se i costi delle chiamate ricorsive **non sono facilmente sommabili** in una forma chiusa

Esempi di applicazione:

$$T(n) = \begin{cases} T(n) = 2T\left(\frac{n}{4}\right) + \Theta(\sqrt{n}) \\ T(1) = \Theta(1) \end{cases}$$

- Ad ogni liv. i abbiamo  $2^i$  nodi, poiché ad ogni nodo avremmo 2 chiamate ricorsive
- Costo di un nodo al liv. i sarà  $\sqrt{\frac{n}{4^i}} = \Theta\left(\frac{\sqrt{n}}{2^i}\right)$
- Costo complessivo al liv. i sarà  $2^i * \frac{\sqrt{n}}{2^i} = \Theta(\sqrt{n})$
- Ci fermiamo quando  $\frac{n}{4^i} = 1 \Rightarrow i = \log_4(n)$ , quindi l'altezza dell'albero è  $\Theta(\log(n))$
- Costo complessivo: dato dalla somma dei costi di tutti i livelli. Poiché ogni liv. ha costo  $\Theta(\sqrt{n})$  e l'altezza dell'albero è  $\Theta(\log(n))$ , il costo è:

$$\sum_{i=0}^{\log_4(n)} \Theta(\sqrt{n}) = \Theta(\sqrt{n}) * \Theta\left(\sum_{i=0}^{\log_4(n)} 1\right) = \Theta(\sqrt{n}) * \Theta(\log(n)) = \Theta(\sqrt{n} * \log(n))$$

Ottavo potevamo scrivere: ad ogni liv. i ci sono  $2^i$  chiamate ricorsive, ciascuna con costo  $\Theta(\frac{\sqrt{n}}{2^i})$ :

$$\sum_{i=0}^{\log_4(n)} 2^i * \Theta(\sqrt{n}) * \Theta\left(\frac{1}{2^i}\right) = \Theta(\sqrt{n}) * \sum_{i=0}^{\log_4(n)} 2^i * \frac{1}{2^i} = \Theta(\sqrt{n}) * \sum_{i=0}^{\log_4(n)} 1 = \Theta(\sqrt{n}) * \Theta(\log(n)) = \Theta(\sqrt{n} * \log(n))$$

## Metodo di Sostituzione

Si ipotizza **intuitivamente** una soluzione, verificando poi per **induzione** se essa funziona. Questo metodo è tedioso in quanto è difficile trovare una funzione vicina alla vera soluzione, quindi non è molto utile nella pratica.

**Non applicabile** quando non si riesce a indovinare una **buona sol. iniziale** per la verifica per induzione.

**Problemi tipici non applicabili:**

- Se la sol. dipende da una **sommatoria difficile da gestire**

- Se la verifica per induzione porta a passaggi algebrici **molto complessi** senza una chiara convergenza
- Se la ricorrenza ha un **comportamento irregolare** che rende difficile formulare un'ipotesi plausibile

Esempio di applicazione:

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

**Ipotizziamo che  $T(n) = \Theta(n)$** , dobbiamo dimostrare che  $T(n)$  è sia  $O(n)$  che  $\Omega(n)$

**Dimostrazione  $O(n)$**

**Ipotizziamo che  $T(n) = O(n)$** , ossia che  $T(n) \leq k^*n$  per una costante  $k$ . Inoltre è importante **eliminare la notazione asintotica**, sostituendo  $\Theta(1)$  con **due costanti c e d** fissate

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

**Caso base:**  $n = 1 \Rightarrow T(1) \leq k^*1$ , essendo  $T(1) = d$  otteniamo che la disegualanza è vera se  $d \leq k$

**Passo induttivo:**

- per un  $n$  generico si ha  $T(n) \leq k^*n$
- sapendo che  $T(n) = T(n-1) + c$  otteniamo la disegualanza  $T(n-1) + c \leq k^*n$
- inoltre per **ipotesi induttiva**, essendo  $T(n) = O(n)$ , si ha che  $T(n-1) \leq k^*(n-1)$ , dunque:

$$k^*(n-1) + c \leq k^*n \Rightarrow k^*n - k + c \leq k^*n \Rightarrow -k + c \leq 0 \Rightarrow k \geq c$$

Poiché esiste sempre un val.  $k$  t.c.  $k \geq c$  e  $k \geq d$ , la sol. ipotizzata  $T(n) \leq k^*n$  è corretta quindi  $T(n) = O(n)$

**Dimostrazione  $\Omega(n)$**

**Ipotizziamo che  $T(n) = \Omega(n)$** , ossia che  $T(n) \geq h^*n$  per una costante  $h$ . Come prima, sostituiamo  $\Theta(1)$  con le costanti  $c$  e  $d$ .

**Caso base:**  $n = 1 \Rightarrow T(1) \geq h^*1$ , essendo  $T(1) = d$ , la disegualanza è vera se  $d \geq h$

**Passo induttivo:**

- Per **ipotesi induttiva** avremo anche  $T(n-1) \geq h^*(n-1)$ , dunque

$$T(n) = T(n-1) + c \geq h^*(n-1) + c \Rightarrow h^*n - h + c \geq h^*n \Rightarrow -h + c \geq 0 \Rightarrow c \geq h$$

Poiché esiste sempre un val.  $h$  t.c.  $h \leq c$  e  $h \leq d$ , la sol. ipotizzata  $T(n) \geq h^*n$  è corretta quindi  $T(n) = \Omega(n)$

Avendo dimostrato  $T(n) = \Omega(n)$  e  $T(n) = O(n)$ , possiamo dire che  $T(n) = \Theta(n)$

## Metodo Principale

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

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

Ha tre casi di risoluzione in base a  $f(n)$ :

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

con  $\epsilon > 0$  e  $c < 1$

In termini ssemplificati abbiamo **tre casi** che dipendono dal **confronto tra  $f(n)$  e  $n^{\log_b(a)}$**

- **Caso 1:** Se il **più grande dei due** è  $n^{\log_b(a)}$ , il costo è  $\Theta(n^{\log_b(a)})$
- **Caso 2:** Se sono **uguali**, si moltiplica  $f(n)$  per  $\log(n)$ , quindi il costo è  $\Theta(f(n) * \log(n)) = \Theta(n^{\log_b(a)} * \log(n))$
- **Caso 3:** Se il **più grande dei due** è  $f(n)$  e  $a^*f(n/b) \leq c^*f(n)$  con  $c < 1$ , il costo è  $\Theta(f(n))$

**Non applicabile** quando la ricorrenza non è nella forma standard  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

**Problemi tipici non applicabili:**

- Se  $a$  o  $b$  non sono costanti (dipendono da  $n$ )
- Se  $f(n)$  non è una funz. polinomiale o non può essere confrontata direttamente con  $n^{\log_b(a)}$
- Se la suddivisione del problema non è uniforme (es.  $T(n) = T(n-1) + T\left(\frac{n}{2}\right) + \Theta(1)$ )

## Metodo generale per trovare il val. massimo di $a$

Nel caso in cui conosciamo il costo di un algoritmo iterativo  $\Theta(g(n))$  e ci viene proposta una versione ricorsiva con equazione:

$$T = \begin{cases} T(n) = a * T\left(\frac{n}{b}\right) + f(n) \text{ per } n \geq x \\ T(n) = \Theta(1) \text{ altrimenti} \end{cases}$$

Dove  $a$  è un incognita costante intera positiva con  $a \geq y$ ,  $b$  e  $f(n)$  invece sono conosciute

Per determinare il **valore massimo** che la costante  $a$  può avere per fare in modo che l'algoritmo ricorsivo risulti asintoticamente **più efficiente** dell'algoritmo iterativo, dobbiamo:

- 1) Risolvere la ricorrenza usando  $y$  al posto di  $a$  per determinare il **termine dominante**:

Quindi confrontiamo il costo di  $f(n)$  con il costo di  $n^{\log_b(y)}$

- $T(n) = \Theta(n^{\log_b(y)})$  se  $f(n) = O(n^{\log_b(y)-\epsilon})$
- $T(n) = \Theta(n^{\log_b(y)} * \log(n))$  se  $f(n) = \Theta(n^{\log_b(y)})$
- $T(n) = \Theta(f(n))$  se  $f(n) = \Omega(n^{\log_b(y)+\epsilon})$  e  $a * f\left(\frac{n}{b}\right) \leq c * f(n)$  con  $c < 1$

Dobbiamo pertanto trovare il caso in cui ci troviamo e il costo della ricorsione in  $\Theta$

- 2) Riscriviamo la **disegualanza** per trovare il **val. massimo di  $a$**

Se  $g(n) = \Theta(n^p)$  possiamo calcolare il val di  $a$  per la ricorrenza iterativa con  $a = b^p$

Quindi se vogliamo trovare un algoritmo con costo minore dobbiamo avere  $a \leq b^p - 1$

## Esempio

Costo iterativo è  $\Theta(n^2)$  e l'equazione ricorsiva è:

$$T = \begin{cases} T(n) = a * T\left(\frac{n}{4}\right) + \Theta(1) \text{ per } n \geq 4 \\ T(n) = \Theta(1) \text{ altrimenti} \end{cases}$$

- Cominciamo col risolvere la ricorrenza inserendo  $y$  al posto di  $a$ :

Applicando il metodo principale abbiamo  $f(n) = \Theta(1)$  e  $n^{\log_4(a)} \geq n^{\log_4(2)} = n^{\frac{1}{2}}$  si ha quindi  $f(n) = O(n^{\log_4(a-\epsilon)})$   
Siamo quindi nel **primo caso** e il costo è  $\Theta(n^{\log_4(a)})$

- Troviamo il massimo di  $a$

Se  $a = 16 (4^2)$  la ricorrenza ha soluzione  $\Theta(n^{\log_4(16)}) = \Theta(n^2)$  quindi perché l'algoritmo ricorsivo abbia costo inferiore bisogna avere  $a \leq 15$

### Esempio di applicazione del primo caso

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

- $a = 9, b = 3$
- $f(n) = \Theta(n)$
- $n^{\log_b a} = n^{\log_3 9} = n^2$

Secondo il 1° caso, vediamo se:

- $\exists \epsilon > 0$  t.c.  $f(n) = O(n^{\log_b a - \epsilon})$
- Se  $\epsilon = 1$  allora  $f(n) = O(n^{2-1}) = O(n)$

Quindi  $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$

### Esempio di applicazione del secondo caso

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

- $a = 1, b = 3/2$
- $f(n) = \Theta(1)$
- $n^{\frac{\log_3 1}{2}} = n^0 = 1$

Secondo il 2° caso, vediamo se:

- $f(n) = \Theta(n^{\log_b a}) \Rightarrow \Theta(1) = \Theta(1)$

Quindi  $T(n) = \Theta(n^{\log_b a} * \log n)$ , cioè  $T(n) = \Theta(\log(n))$

### Esempio di applicazione del terzo caso

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

- $a = 3, b = 4$
- $f(n) = \Theta(n * \log(n))$
- $n^{\log_b a} = n^{\log_4 3} \approx n^{0.7}$

Secondo il 3° caso, vediamo se:

- $\exists \epsilon$  t.c.  $f(n) = \Omega(n^{\log_b a + \epsilon}) \Rightarrow \Theta(n * \log n) = \Omega(n^{0.7 + \epsilon})$
- Se  $\epsilon \leq 0.2 \Rightarrow \Theta(n * \log n) = \Omega(n^{0.9})$  poiché  $n * \log(n)$  domina  $n^{0.9}$
- $\exists c < 1$  t.c.  $a * f\left(\frac{n}{b}\right) \leq c * f(n) \Rightarrow 3 * f\left(\frac{n}{4}\right) \leq c * n * \log n \Rightarrow 3 * \Theta\left(\frac{n}{4} * \log \frac{n}{4}\right) \leq c * n * \log n$
- $3 * \Theta\left(\frac{n}{4} * \log \frac{n}{4}\right) \leq c * n * \log n \Rightarrow \frac{3}{4}(n * \log n - n * \log 4) \leq c * n * \log n$
- Vero se  $c = 3/4$

Quindi  $T(n) = \Theta(n * \log(n))$

### Esempio di non applicazione

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n * \log(n))$$

- $a = 2, b = 2$
- $f(n) = \Theta(n * \log(n))$
- $n^{\log_b a} = n^{\log_2 2} = n$

$f(n) = \Theta(n * \log(n))$  è asintoticamente più grande di  $n^{\log_2 2} = n$ , ma non polomialmente più grande.

Infatti,  $\log(n)$  è asintoticamente minore di  $n^\epsilon$  per qualunque  $\epsilon > 0$  (all'infinito cresce più lentamente di qualsiasi  $n^\epsilon$ ).  
Di conseguenza non possiamo applicare il metodo del teorema principale

# Ordinamento

sabato 8 febbraio 2025 16:20

## Riassunto Ordinamenti

### Insertion Sort

Inserisce ogni elem. nella pos. corretta rispetto ai precedenti, spostando a destra gli elementi **più grandi**

- **Costo Medio:**  $O(n^2)$
- **Costo Migliore:**  $O(n)$  (array già ordinato)
- **Costo Peggiori:**  $O(n^2)$  (array già ordinato al contrario)

#### Dimostrazione:

- Per ogni elem. si confronta con **tutti i precedenti**
- Quindi due cicli:  $O(n^2)$

Vantaggi: Semplice, adatto per piccoli array

Svantaggi: Lento su input grandi

Lo pseudocodice dell'algoritmo è il seguente.

```
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 = i - 1
        A[i+1] = x
```

### Selection Sort

Seleziona ripetutamente il **val. minimo** e lo scambia con l'elem. corrente

- **Costo Medio/Migliore/Peggiori:** sempre  $O(n^2)$

#### Dimostrazione:

- Sempre  $\frac{n*(n-1)}{2}$  confronti e  $O(n)$  scambi
- Due cicli for annidati

Vantaggi: Semplice, pochi scambi

Svantaggi: Lento, non stabile

```
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]
```

### Bubble Sort

Confronta **coppie adiacenti** e scambia se necessario, spostando il massimo alla fine

- **Costo Medio/Migliore/Peggiori:** sempre  $O(n^2)$

#### Dimostrazione:

- $\frac{n*(n-1)}{2}$  confronti nel caso peggiore
- Due cicli for annidati

Vantaggi: Semplice, stabile

Svantaggi: Lento, inefficiente

```
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]
```

### Merge Sort

Divide ricorsivamente l'**array a metà**, **ordina** le due **metà** e le **fonde**

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

- **Costo Medio/Migliore/Peggiori:**  $\Theta(n * \log(n))$  (2° caso Metodo Principale)

#### Dimostrazione:

- Divide e unisce in  $O(n)$
- Con profondità  $O(\log(n))$

Vantaggi: Stabile, efficiente

Svantaggi: Usa memoria extra (2° array in Fondi)

```
def merge_sort(A, primo_i, ultimo_i):
    # Divide ricorsivamente l'array in due sottoarray più piccoli fino al singolo elemento
    if primo_i < ultimo_i:
        medio_i = (primo_i + ultimo_i) // 2 # calcola l'indice medio dell'array
        merge_sort(A, primo_i, medio_i) # ordina il primo sottoarray
        merge_sort(A, medio_i + 1, ultimo_i) # ordina il secondo sottoarray
    return fondi(A, primo_i, medio_i, ultimo_i) # fonda i due sottoarray ordinati
```

```
def fondi(A, primo_i, medio_i, ultimo_i):
    i, j = primo_i, medio_i + 1 # i e j sono gli indici dei due sottoarray da fondere
    B = []
    while i <= medio_i and j <= ultimo_i: # finché nessuno dei due sottoarray è terminato
        if A[i] <= A[j]: # confronta gli elementi dei due sottoarray, se il minore è nel primo sottoarray
            B.append(A[i]) # aggiunge l'elemento minore al vettore B
            i += 1 # incrementa l'indice del primo sottoarray
        else: # se l'elemento minore è nel secondo sottoarray
            B.append(A[j]) # aggiunge l'elemento minore al vettore B
            j += 1 # incrementa l'indice del secondo sottoarray

    while i <= medio_i: # il primo sottoarray non è terminato quindi aggiunge gli elementi rimasti nel sottoarray a B
        B.append(A[i])
        i += 1
    while j <= ultimo_i: # il secondo sottoarray non è terminato quindi aggiunge gli elementi rimasti nel sottoarray a B
        B.append(A[j])
        j += 1
    for i in range(len(B)): # copia il vettore ordinato in A
        A[primo_i + i] = B[i]
    return A
```

### Quick Sort

Scelgono un pivot che separa gli elem. **minori e maggiori** del pivot,

per poi ordinare ricorsivamente le due parti

- **Costo Medio/Migliore:**  $\Theta(n * \log(n))$
- **Costo Peggiori:**  $\Theta(n^2)$

#### Dimostrazione:

- Se il pivot divide in due parti uguali
  - profondità  $O(\log(n))$
  - partizionamento  $\Theta(n)$

Vantaggi: Veloce, ordinamento "in loco"

Svantaggi: Instabile,  $O(n^2)$  nel peggioro dei casi

```
def quick_sort(A, primo_i, ultimo_i):
    if primo_i < ultimo_i: # se l'array ha più di un elemento
        medio_i = Partizione(A, primo_i, ultimo_i) # partiziona l'array e restituisce l'indice del pivot
        quick_sort(A, primo_i, medio_i - 1) # ordina il sottoarray sinistro
        quick_sort(A, medio_i + 1, ultimo_i) # ordina il sottoarray destro
    return A
```

```
def Partizione(A, primo_i, ultimo_i):
    pivot = A[primo_i] # il pivot è il primo elemento
    i = primo_i + 1 # i è l'indice del primo elemento successivo al pivot
    for j in range(primo_i + 1, ultimo_i + 1): # scorre tutti gli elementi
        if A[j] < pivot: # se l'elemento è minore di pivot
            A[i], A[j] = A[j], A[i] # scambia l'elemento minore di pivot con il primo elemento successivo al pivot
            i += 1 # incrementa l'indice del primo elemento successivo al pivot
    A[i-1], A[primo_i] = A[primo_i], A[i-1] # scambia il pivot con l'ultimo elemento minore di esso
    return i - 1 # restituisce l'indice del pivot
```

### Heap Sort

Costruisce un heap e successivamente in un ciclo estrae ripetutamente il massimo

e ripristina le proprietà dell'heap

- **Costo Medio/Migliore/Peggiori:**  $O(n * \log(n))$

#### Dimostrazione:

- Crea l'heap  $\Theta(n)$  (`Build_heap`)
- Ad ogni estrazione costa  $O(\log(n))$  (`Heapiify`) per n elementi:  $O(n * \log(n))$

```
def Build_heap(A):
    for i in reversed(range(len(A)//2)): # scorre tutti i nodi interni al contrario
        Heapify(A, i, len(A)) # chiama Heapify su ogni nodo interno
    return A
```

- Costo Medio/Migliore/Peggiori:  $O(n \log n)$

#### Dimostrazione:

- Crea l'heap  $\Theta(n)$  (Build\_heap)
- Ad ogni estrazione costa  $O(\log n)$  (Heapify) per  $n$  elementi:  $O(n \log n)$

Vantaggi: Efficiente, ordinamento "in loco", costo fisso

Svantaggi: Più lento di Quick Sort in media

def Heapsort (A):

```
Build_heap(A)
for x in reversed(range(1, len(A))): (n-1) iteraz.
    A[0], A[x] = A[x], A[0]
    Heapify(A, 0, x)
```

O(n)

Θ(1)

O(log n)

```
for i in reversed(range(len(A)//2)): # scorre tutti i nodi interni al contrario
    Heapify(A, i, len(A)) # chiama Heapify su ogni nodo interno
return A
```

## Counting Sort

Conta le occorrenze di ciascun valore e le riscrive in ordine

- Costo Medio/Migliore/Peggiori:  $O(n+k) = O(\max(n, k))$

#### Dimostrazione:

- Trova il val. max  $k$  ( $O(n)$ )
- Crea un array di grandezza  $k$  ( $O(k)$ )
- Contiamo le occorrenze dei val. ( $O(n)$ )
- Ricreiamo l'array ordinato ( $O(n)$ )

Vantaggi: Lineare se  $k = O(n)$ , stabile

Svantaggi: richiede spazio extra, non gestisce dati con chiavi non intere

## Bucket Sort

Divide gli elem. in bucket, ordina gli elem. nei bucket con un algoritmo ausiliario (spesso Insertion Sort) e poi li concatena

- Costo Medio:  $\Theta(n)$  (se distribuzione uniforme)
- Costo Migliore:  $\Theta(n)$
- Costo Peggiori:  $\Theta(n^2)$  (se tutti gli elem. finiscono in un solo bucket)

#### Dimostrazione:

- $\Theta(n)$  per assegnare gli elem. ai bucket
- $\Theta(n)$  per ordinare piccoli bucket

Vantaggi: Molto veloce su input uniformi

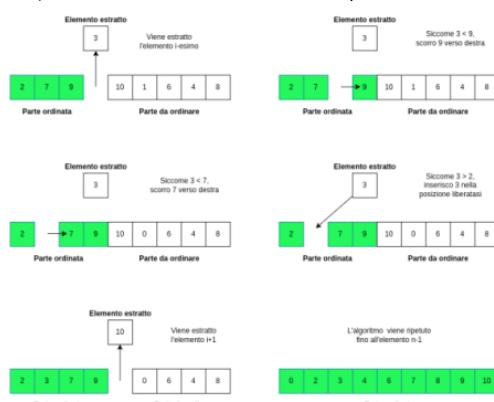
Svantaggi: Dipende dalla distribuzione, richiede memoria extra

## Ordinamenti semplici

Sono ordinamenti semplici che effettuano ordinamento in loco e hanno costo  $\Theta(n^2)$

### Insertion Sort

- Estraggo l'elem. in pos.  $i$
- Sposto verso destra tutti gli elem. maggiori
- Reinserisco l'elem. estratto nella posizione corretta



Lo pseudocodice dell'algoritmo è il seguente.

```
def Insertion_Sort(A)
for j in range(1, len(A)):
    (n-1) Θ(1) + Θ(1)
        x = A[j]
        i = j - 1
        while ((i >= 0) and (A[i]) > x) tjΘ(1) + Θ(1)
            A[i+1] = A[i]
            i = i - 1
        A[i+1] = x
                Θ(1)
```

Costo computazionale:

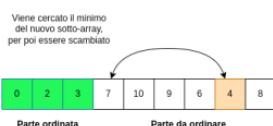
$$T(n) = \sum_{j=0}^{n-1} (\Theta(1) + t_j \Theta(1) + \Theta(1)) + \Theta(1)$$

- Caso migliore: l'elem. precedente a quello estratto è minore (una sola operazione) ( $j = 1$ )  
 $T(n) = (n - 1) * \Theta(1) = \Theta(n)$
- Caso peggiore: tutti gli elem. precedenti a quello estratto sono maggiori ( $j$  operazioni)  
 $T(n) = \sum_{j=0}^{n-1} (\Theta(1) + \Theta(j)) = \Theta(n) + \Theta(n^2) = \Theta(n^2)$

## Selection Sort

- Ricercò il minimo val.
- Il val. minimo viene scambiato con l'elem.  $i$
- Si ricerca il nuovo minimo nell'array restante da  $i$  fino a  $n-1$  e così via





```

for j in range(i+1, len(A)):   (n - i) Θ(1) + Θ(1)
    if (A[j] < A[m])           Θ(1)
        m = j                   Θ(1)
    A[m], A[i] = A[i], A[m]   Θ(1)

```

### Costo Computazionale:

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

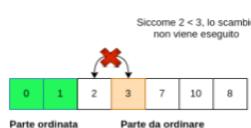
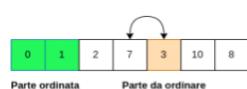
## Bubble Sort

- 1) Vengono analizzate tutte le **coppie di elem. adiacenti**, da destra verso sinistra
- 2) Se l'elem. più a destra è minore di quello precedente, vengono **scambiati**
- 3) L'elem. minore verrà trasportato nella sua posizione finale
- 4) L'algoritmo viene **ripetuto sul sotto-array** composto dagli elementi rimasti

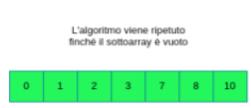
Siccome l'elemento analizzato è maggiore del suo precedente, lo scambio non avviene.



Anche questa volta lo scambio viene effettuato.



Girati alla fine del sotto-array, l'algoritmo verrà ripetuto sul sotto-array rimanente.



### Costo Computazionale:

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

```

def Bubble_Sort(A)
    for i in range(len(A)):   n Θ(1) + Θ(1)
        for j in range(len(A)-1, i, -1):   (n - i) Θ(1) + Θ(1)
            if (A[j] < A[j - 1]):           Θ(1)
                A[j], A[j - 1] = A[j-1], A[j]   Θ(1)

```

## Ordinamento più efficienti

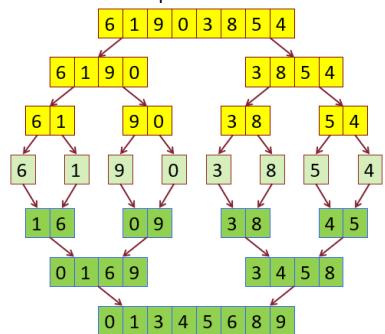
Ordinamenti basati sul **costo limite inferiore teorico di qualsiasi algoritmo basato sui confronti**, cioè  $\Omega(n \log n)$ .

### Merge Sort

Basato sulla tecnica **divide et impera**, dove il problema complessivo viene **diviso in sottoproblemi (divide)** che vengono **risolti ricorsivamente (impera)** e poi **ricomposti** per dare la **soluzione al problema originale (combina)**

Nel caso del **merge sort**, abbiamo:

- **Divide**: l'array viene diviso in **due sottosequenze di  $n/2$  elem.**
- **Impera**: le due sottosequenze vengono **ricorsivamente** divise di nuovo in  $n/2$  elem.
- **Caso base**: la ricorsione termina quando la sottosequenza ha un solo elem.
- **Combina**: qui avviene l'**ordinamento**; le sottosequenze vengono **fuse** in un'unica sequenza **ordinata** (usando l'algoritmo **Fondi**)



```

def merge_sort(A, primo_i, ultimo_i):
    # Divide ricorsivamente l'array in due sottoarray più piccoli fino al singolo elemento e infine fonde insieme i sottoarray ordinandoli
    if primo_i < ultimo_i:
        medio_i = (primo_i + ultimo_i) // 2 # calcola l'indice medio dell'array
        merge_sort(A, primo_i, medio_i) # ordina il primo sottoarray
        merge_sort(A, medio_i + 1, ultimo_i) # ordina il secondo sottoarray
        return fondi(A, primo_i, medio_i, ultimo_i) # fonde i due sottoarray ordinati

```

### Costo Computazionale:

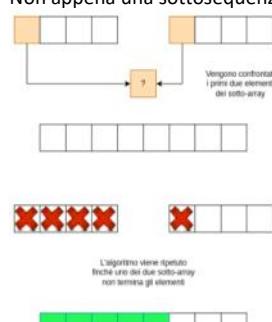
$$T(n) = \begin{cases} T(n) = \Theta(1) + 2T\left(\frac{n}{2}\right) + F(n) \\ T(1) = \Theta(1) \end{cases}$$

Dove  $F(n)$  è il costo di **Fondi()**

### Algoritmo Fondi

Confronta gli elem. di due sottosequenze già ordinate, inserendo e ordinando gli elementi in un array ausiliario.

Non appena una sottosequenza ha **terminato i suoi elem.**, gli elem. rimasti nell'altra sottosequenza vengono aggiunti in coda nell'array ausiliario.



```

def fondi(A, primo_i, medio_i, ultimo_i):
    i, j = primo_i, medio_i + 1 # i e j sono gli indici dei due sottoarray da fondere
    B = []
    while i <= medio_i and j <= ultimo_i: # finché nessuno dei due sottoarray è terminato
        if A[i] <= A[j]: # confronta gli elementi dei due sottoarray, se il minore è nel primo sottoarray
            B.append(A[i]) # aggiunge l'elemento minore al vettore B
            i += 1 # incrementa l'indice del primo sottoarray
        else: # se l'elemento minore è nel secondo sottoarray
            B.append(A[j]) # aggiunge l'elemento minore al vettore B
            j += 1 # incrementa l'indice del secondo sottoarray
    while i <= medio_i: # il primo sottoarray non è terminato quindi aggiunge gli elementi rimasti nel sottoarray a B
        B.append(A[i])
        i += 1
    while j <= ultimo_i: # il secondo sottoarray non è terminato quindi aggiunge gli elementi rimasti nel sottoarray a B
        B.append(A[j])
        j += 1
    return B

```

Algoritmo Merge Sort



```

while i <= medio_i: # il primo sottoarray non è terminato quindi aggiunge gli elementi rimasti nel sottoarray a B
    B.append(A[i])
    i += 1
while j <= ultimo_i: # il secondo sottoarray non è terminato quindi aggiunge gli elementi rimasti nel sottoarray a B
    B.append(A[j])
    j += 1
for i in range(len(B)):
    A[primo_i + i] = B[i]
return A

```

#### Valutazione costo di Fondi():

- Inizializzazione var:  $\Theta(1)$

- 1° ciclo while:

Incrementa di 1 l'indice i o j, il costo varia:

- Da un **minimo di  $n/2$**  (se tutti i minimi sono in una delle due sottosequenze):  $\Omega(n/2) = \Omega(n)$
- Ad un **massimo di  $n$**  (se le sottosequenze terminano in contemporanea):  $O(n)$

Costo  $\Theta(n)$

- 2° e 3° ciclo while:

Non vengono mai eseguiti entrambi, quindi vengono considerati come un unico ciclo. Il numero di iter. varia:

- Da un **minimo di 1** (se è rimasto un solo elem. nella sottosequenza):  $\Omega(1)$
- Ad un **massimo di  $n/2$**  (Se tutti i minimi sono in una delle due sottosequenze):  $O(n/2) = O(n)$

Costo  $O(n)$

- 4° ciclo while:

Copia gli elem. ordinati da B nella porzione di A:  $\Theta(n)$

Dunque  $F(n) = \Theta(1) + \Theta(n) + O(n) + \Theta(n) = \Theta(n)$

Quindi l'equazione finale del merge sort è:

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

Usando il **metodo principale** vediamo che il costo corrisponde a  $\Theta(n \log n)$

## Quick Sort

Riunisce i **vantaggi** del Selection Sort (**ordinamento in loco**) e del Merge Sort (**tempo di esecuzione ridotto**).

Nonostante abbia un costo nel **caso peggiore di  $O(n^2)$** , nella pratica è spesso la soluzione migliore per grandi val. di n per via dei suoi vantaggi.

Anche questo algoritmo utilizza la tecnica **divide et impera**:

- Divide:** nella sequenza di n. elem. si seleziona un **pivot**.  
La sequenza si divide in due sottosequenze: 1° con elem.  $<$  pivot, 2° con elem.  $>$  pivot.
- Impera:** le due sottosequenze sono **ordinate ricorsivamente**
- Caso base:** la ricorsione termina quando la sottosequenza ha un solo elem.

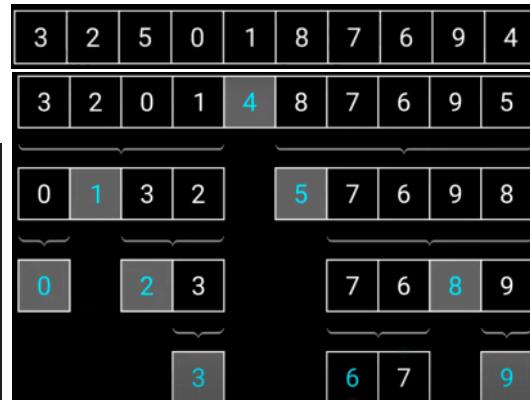
L'ordinamento viene svolto nella **fase di continua divisione ordinata ricorsiva**, ossia il **partizionamento**

```

def quick_sort(A, primo_i, ultimo_i):
    if primo_i < ultimo_i:
        medio_i = Partiziona(A, primo_i, ultimo_i) # partiziona l'array e restituisce l'indice del pivot
        quick_sort(A, primo_i, medio_i - 1) # ordina il sottoarray sinistro
        quick_sort(A, medio_i + 1, ultimo_i) # ordina il sottoarray destro
    return A

def Partiziona(A, primo_i, ultimo_i):
    pivot = A[primo_i] # il pivot è il primo elemento
    i = primo_i + 1 # i è l'indice del primo elemento successivo al pivot
    for j in range(primo_i + 1, ultimo_i + 1): # scorre tutti gli elementi
        if A[j] < pivot: # se l'elemento è minore di pivot
            A[i], A[j] = A[j], A[i] # scambia l'elemento minore di pivot con il primo elemento successivo al pivot
            i += 1 # incrementa l'indice del primo elemento successivo al pivot
    A[i-1], A[primo_i] = A[primo_i], A[i-1] # scambia il pivot con l'ultimo elemento minore di esso
    return i - 1 # restituisce l'indice del pivot

```



Costo di **Partiziona()**:

- Inizializzazione var:  $\Theta(1)$
- Ciclo for: scansione della sequenza, compiendo un num. di operazioni pari alla **dimensione dell'array:  $\Theta(n)$**
- Op. finale di spostamento del pivot nella sua pos. finale, ritornandone l'indice:  $\Theta(1)$

Quindi il costo è  $\Theta(n)$

Partiziona poi **divide la sequenza originale in due sottosequenze: 1° con k elem., 2° con n-k elem.**

L'equazione di ricorrenza sarà:

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

Calcoliamo il suo **caso peggiore, migliore e costo Medio**:

- Caso migliore:** ad ogni ricorsione, ogni sottoproblema viene **diviso a metà**, ossia  $k = \frac{n}{2}$ .

$$T(n) = T(k) + T(n - k) + \Theta(n) = T\left(\frac{n}{2}\right) + T\left(n - \frac{n}{2}\right) + \Theta(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

In modo simile al merge sort, ha soluzione  $T(n) = \Theta(n \log n)$

- Caso peggiore:** ad ogni ricorsione, la **dimensione di uno dei due sottoproblemi è 1 e l'altro  $n-1$** , ossia  $k = 1$ :

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

Applicando il **metodo iterativo** abbiamo:

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

Quindi il costo è

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

- Caso Costo Medio:** poiché il caso peggiore e migliore non coincidono, valutiamo il caso Costo Medio

Ipotizziamo che il pivot suddivida con **uguale probabilità**  $\frac{1}{n-1}$  la sequenza in due sottosequenze di dimensioni  $k$  e  $n-k$  **per tutti i valori di  $k$  tra 0 e  $n-1$**

Dunque l'equazione di ricorrenza sarà:

$$T(n) = \frac{1}{n-1} \left[ \sum_{k=0}^{n-1} T(k) - T(n-k) \right] + P(n)$$

Il cui **costo computazionale** è, usando il **metodo della sostituzione**,  $O(n \log(n))$

Dunque, poiché sappiamo che tale equazione è  $O(n \log(n))$  e che il **teorema della complessità di un algoritmo di ordinamento basato sui confronti** impone che esso sia anche  $\Omega(n \log(n))$ , possiamo dire che il **caso Costo Medio sia  $O(n \log(n))$**

Per fare in modo che l'**ipotesi di equiprobabilità** venga soddisfatta, possiamo rendere l'algoritmo **indipendente dall'input**:

- **Randomizzando la sequenza** prima di avviare l'algoritmo
- Scegliendo un **pivot a caso** nella sequenza

## Heap Sort

Similmente al Quick Sort, riunisce i vantaggi dell'**ordinamento in loco** e del **costo computazionale ridotto**, pero il costo nel caso peggiore è sempre  $O(n \log(n))$

Sfrutta una **struttura dati**, detta **Heap**, essenziale per il funzionamento dell'algoritmo.

Un **heap** è un **albero binario** quasi completo, con le seguenti proprietà

- tutti i livelli eccetto l'ultimo sono pieni
- i nodi sono addensati a sinistra
- proprietà principale dell'**ordinamento verticale**: la chiave di ogni nodo corrisponde ad un val. maggiore o uguale alla chiave dei suoi due figli

L'**Heap** deve rispettare alcune caratteristiche:

- Viene **implementato con un array**, con indici da 1 a **heap size** (num. nodi dell'Heap)
- L'array è riempito da sinistra e gli elem. **oltre heap size** non fanno parte dell'heap
- La **radice** è in  $A[0]$  ed è il **val. max** dell'Heap
- Considerato il nodo  $A[i]$ :
  - Il **figlio sinistro** è l'elem.  $A[2i+1]$
  - Il **figlio destro** è l'elem.  $A[2i+2]$
  - il **padre** è l'elem.  $A[\lfloor \frac{i-1}{2} \rfloor]$ :  $\text{parent}(i) = \lfloor \frac{i-1}{2} \rfloor$

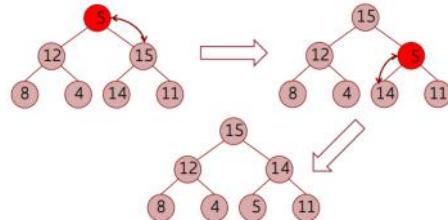
Ogni **livello** contiene  $2^h$  nodi, quindi  $h = O(\log(n))$

Per poter usare l'algoritmo di ordinamento Heap Sort, bisogna modificare l'array in modo che **rispetti le proprietà dell'Heap**. Per fare ciò usiamo due funzioni:

## Heapify

Ripristina le **proprietà dell'Heap**, confrontando il nodo con i figli e scambiandolo con il **figlio maggiore**, se necessario.

Si chiama ricorsivamente sul nodo figlio scambiato se necessario.



```
def Heapify(A, i, heap_size):
    L = 2*i+1 # indice del figlio sinistro
    R = 2*i+2 # indice del figlio destro
    max_i = i # indice del massimo tra i, L e R
    if L < heap_size and A[L] > A[i]: # se il figlio sinistro è maggiore del padre
        max_i = L # salva l'indice del figlio sinistro
    if R < heap_size and A[R] > A[max_i]: # se il figlio destro è maggiore del padre
        max_i = R # salva l'indice del figlio destro
    if max_i != i: # se il padre non è il massimo
        A[i], A[max_i] = A[max_i], A[i] # scambia il padre con il massimo
        Heapify(A, max_i, heap_size) # richiama ricorsivamente Heapify sul figlio massimo
    return A
```

L'equazione ricorsiva si può trovare in due modi:

- **In base al cammino di Heapify**: Heapify effettua un lavoro  $\Theta(1)$  su **ogni nodo** lungo un cammino con **lunghezza limitata da  $O(\log(n))$**

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

Poiché  $h = O(\log(n))$ , la soluzione nel **caso peggiore** è  $T(n) = O(\log(n))$

- **In base al massimo dei nodi di un sottoalbero**:

$$T(n) = \begin{cases} T(n) = T(n') + \Theta(1) \\ T(1) = \Theta(1) \end{cases}$$

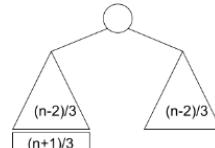
Dove  $n'$  è il **num. di nodi del sottoalbero più grande** tra i figli. Poiché ogni liv. contiene  $2^h$  nodi,

**n' non può essere più grande di  $\frac{2n}{3}$**  (situazione che accade quando l'ultimo livello è pieno **esattamente a metà**)

Dunque l'equazione di ricorrenza diventa:

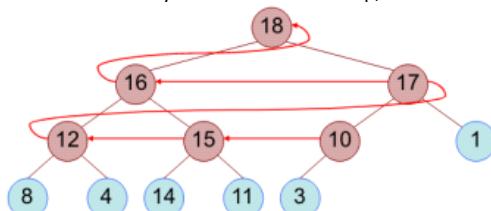
$$T(n) = T\left(\frac{2}{3}n\right) + \Theta(1)$$

Usando il **metodo principale**, vediamo che il costo è  $T(n) = O(\log(n))$  (non usiamo  $\Theta(\log(n))$  perché stiamo considerando il **caso peggiore**)



## Buildheap

Trasforma un array disordinato in un Heap, chiamando **Heapify** dal basso verso l'alto. Il num. di chiamate a Heapify è  $n/2$  dato che le **foglie sono già in Heap**.



```
def Build_heap(A):
    for i in reversed(range(len(A)//2)): # scorre tutti i nodi interni al contrario
        Heapify(A, i, len(A)) # chiama Heapify su ogni nodo interno
    return A
```

(Inserisco l'analisi del costo **meno accurata**, quella più estesa è nella pagina dell'Heap Sort negli Ordinamenti)

Il **ciclo** viene eseguito  $n/2$  volte, e al suo interno vi è solo la chiamata di **Heapify()**, che nel suo **caso peggiore** richiede tempo  $O(h) = O(\log(n))$ .

Quindi anche **Build\_heap** avrà costo  $O(n \log(n))$  nel **caso peggiore**.

Nell'analisi più accurata vedremmo che in realtà ha costo  $O(n)$ .

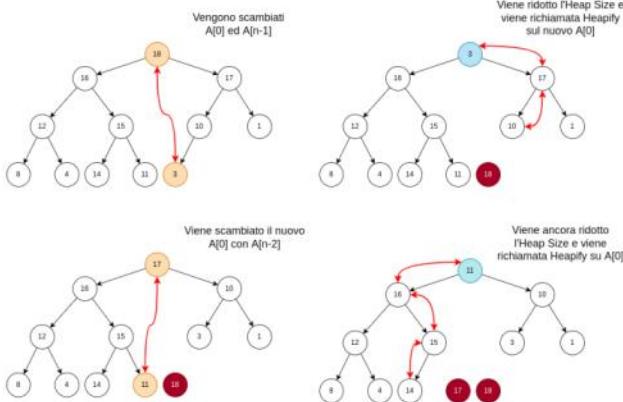
Però la differenza tra il costo  $O(n \log(n))$  (di questa analisi) e  $O(n)$  (dell'analisi estesa) **non influenza sul costo finale dell'algoritmo**.

## Heap Sort

- 1) Richiamo **Buildheap** per costruire l'Heap dall'array
- 2) Scambio il **val. massimo A[0]** con l'ultimo elem.  $A[n]$

3) Riduco heap size e applico Heapify sulla radice per mantenere le proprietà della struttura

4) Ripeto fino a ordinare l'intero array



def Heapsort (A):

Build\_heap(A)

for x in reversed(range(1, len(A))): (n-1) iteraz.

A[0], A[x] = A[x], A[0]

Heapify(A, 0, x)

O(n)

Θ(1)

O(log n)

T(n) = O(n) + (n-1)(Θ(1)) + O(log(n)) = O(n) + Θ(n) + O(n\*log(n)) = O(n\*log(n))  
Poiché O(n\*log(n)) cresce più velocemente di Θ(n) e O(n).

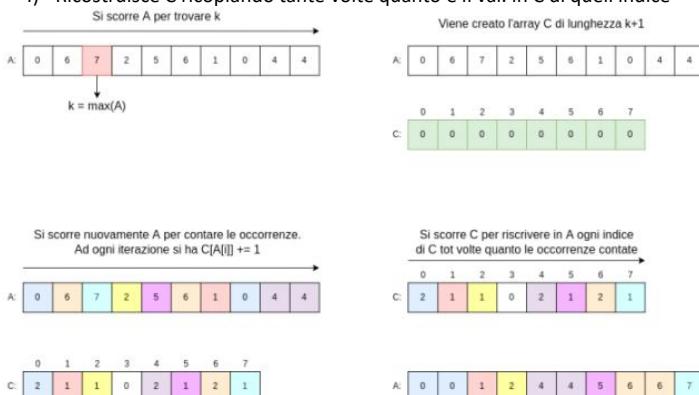
```
def heap_sort(A):
    Build_heap(A) # costruisce un heap massimo
    for i in reversed(range(1, len(A))): # scorre tutti gli elementi al contrario tranne l'ultimo
        A[0], A[i] = A[i], A[0] # scambia il primo elemento con l'elemento corrente
        Heapify(A, 0, i) # richiama Heapify sul primo elemento
    return A
```

## Ordinamenti non basati sui confronti

Gli algoritmi che **operano per confronti** hanno costo computazionale  $\Omega(n \log n)$ . Possiamo avere algoritmi con costo minore poiché **non basati sui confronti**

### Counting Sort

- 1) Trova il **val. max** di A (ossia k) in  $\Theta(n)$
- 2) Crea un **secondo array** di dimensioni  $k+1$
- 3) Conta le **concorrenze** di ogni elem. (per ogni elem.  $A[i]$  incrementa  $C[A[i]]$ )
- 4) Ricostruisce C ricopriando tante volte quanto è il val. in C di quell'indice



def Counting\_sort (A):

k=max(A)

n=len(A)

C[0]\* (k+1)

for j in range(n):

C[A[j]] +=1

//C[i] ora contiene il numero di elementi uguali a i  
j = 0

for i in range(k):

while (C[i] > 0)

A[j]=i

j+=1

C[i]-=1

Θ(n)

Θ(1)

Θ(k)

n volte

$\Theta(1)$

$\sum_{i=0}^k$

$C[i]$  volte

$\Theta(1)$

$\Theta(1)$

$\Theta(1)$

```
def counting_sort(A):
    k = max(A) # trova il massimo elemento
    n = len(A) # lunghezza dell'array
    C[0]* (k + 1) # crea un array di contatori inizializzati a 0
    for j in range(n):
        C[A[j]] += 1 # incrementa il contatore dell'elemento corrente
    j = 0 # indice per l'array ordinato
    for i in range(k): # scorre tutti gli elementi
        while C[i] > 0: # finché il contatore dell'elemento corrente è maggiore di 0
            A[j] = i # inserisce l'elemento corrente nell'array ordinato
            j += 1 # incrementa l'indice per l'array ordinato
            C[i] -= 1 # decrementa il contatore dell'elemento corrente
    return A
```

Costo:

$$T(n) = \Theta(n) + \Theta(k) + \Theta(1) + n * \Theta(1) + \sum_{i=0}^k C[i] * \Theta(1) = \Theta(n) + \Theta(k) + \Theta(n) + n * \Theta(1) = \Theta(n) + \Theta(k) = \Theta(k+n) = \Theta(\max(k, n))$$

Se:

- $k \leq n \rightarrow T(n) = \Theta(n)$
- $k > n \rightarrow T(n) = \Theta(k)$

Counting Sort con Dati Satellite omesso dal riassunto

## Bucket Sort

- 1) **Divido** l'intervallo  $[1, k]$  in  $n$  **sottointervalli** di dimensione  $\frac{k}{n}$ (bucket)

- 2) **Distribuisco** gli elementi nei **bucket** rispettivi

- 3) **Ordino** i singoli bucket con un altro ordinamento

- 4) **Ricombino** gli elementi nei bucket

Poiché i val. sono **uniformemente distribuiti**, ogni bucket contiene in media pochi elementi

$n=8, k=16$

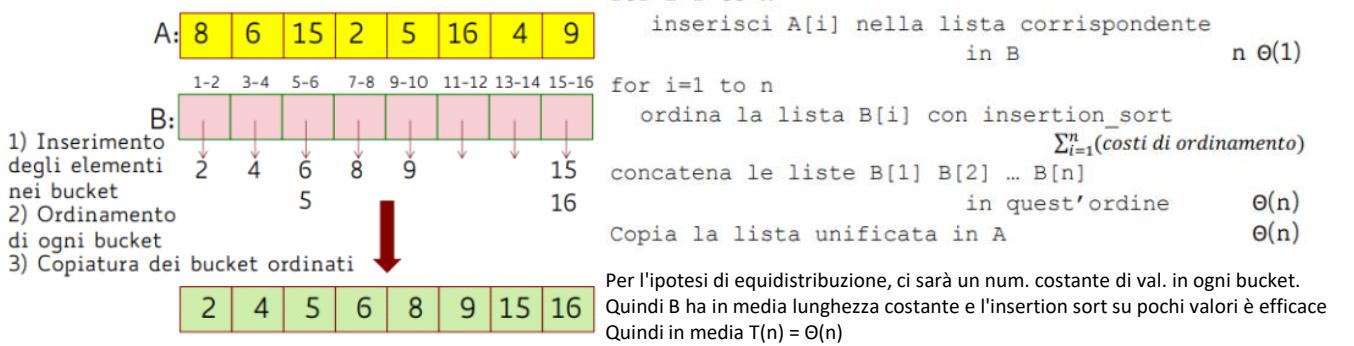


Funzione Bucket\_Sort (A; k, n: interi)

for i=1 to n

inserisci A[i] nella lista corrispondente  
in B

$n \Theta(1)$



# Strutture Dati

sabato 8 febbraio 2025 17:23

## Liste Puntate

Un **puntatore** fa riferimento (punta) all'**indirizzo di mem.** che contiene il valore

Una **lista puntata** è una struttura dati in cui gli elem. sono **collegati tramite puntatore**. Gli elem. (nodi) contengono:

- Key: val. dell'elem.
- Next: puntatore all'elem. successivo (o **None** per l'ultimo elem.)

**Vantaggi** rispetto agli array:

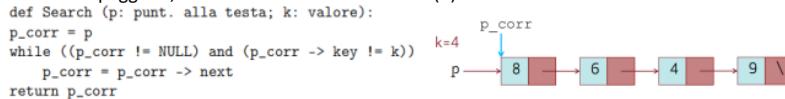
- Inserimenti e eliminazioni **più flessibili**

**Svantaggi** rispetto agli array:

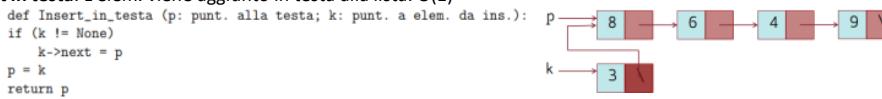
- Necessario l'**accesso sequenziale** agli elem. ( $O(n)$  nel caso peggiore)

Costi operazioni:

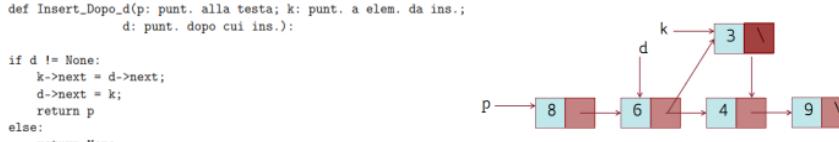
- **Search**: Nel caso peggiore, l'elem. sta in fondo alla lista:  $O(n)$



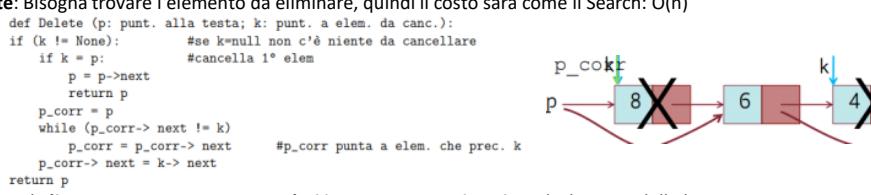
- **Insert in testa**: L'elem. viene aggiunto in testa alla lista:  $\Theta(1)$



- **Insert dopo un elemento**: Nel caso peggiore, il puntatore dopo cui inserire l'elem. sta in fondo alla lista:  $O(n)$

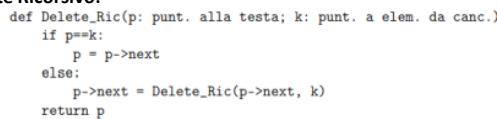


- **Delete**: Bisogna trovare l'elemento da eliminare, quindi il costo sarà come il Search:  $O(n)$



Siccome le **liste puntate sono strutture dati inerentemente ricorsive**, gli algoritmi delle liste puntate possono essere anche in **versione ricorsiva**

- **Delete Ricorsivo**:



La **lista circolare** è una lista puntata in cui il **primo e ultimo elem. sono collegati**,

## Pile e Code

La **pila** è una struttura dati **LIFO (Last In, First Out)**: l'ultimo elem. inserito è il primo a essere estratto

Si possono eseguire solo due op.:

- **Push**: inserisce un elem. in cima, costo  $\Theta(1)$
- **Pop**: estrae l'elem. dalla cima, costo  $\Theta(1)$

La **coda** è una struttura dati **FIFO (First In, First Out)**: il primo entrato è il primo a uscire

Anche qui si possono eseguire solo due op.:

- **Enqueue**: inserisce un elem. in coda, costo  $\Theta(1)$
- **Dequeue**: estrae l'elem. dalla testa, costo  $\Theta(1)$

La **coda con priorità** è una variante della coda in cui gli elem. sono **ordinati** in base ad una **priorità** e non all'ordine di inserimento.

Però si può presentare un problema di **Starvation**, dove gli elem. con **priorità bassa** non vengono **mai estratti** se arrivano continuamente elem. con **priorità più alta**

## Alberi

Un **albero** è una struttura dati estremamente **versatile**.

Essi sono dei tipi di **grafi**:

- **Aciclici**: non vi sono cicli tra i collegamenti dei nodi
- **Connnessi**: non ci sono nodi sconnessi

Gli alberi che vedremo sono alberi **radicati** in cui vi è un nodo speciale detto **radice** da cui parte la **struttura gerarchica**.

I nodi sono organizzati in **livelli** da 0 a h, dove h è il cammino più lungo dalla radice alla foglia.

Un nodo può avere 0 o più figli. Se un nodo ha **0 figli** esso è una **foglia** dell'albero

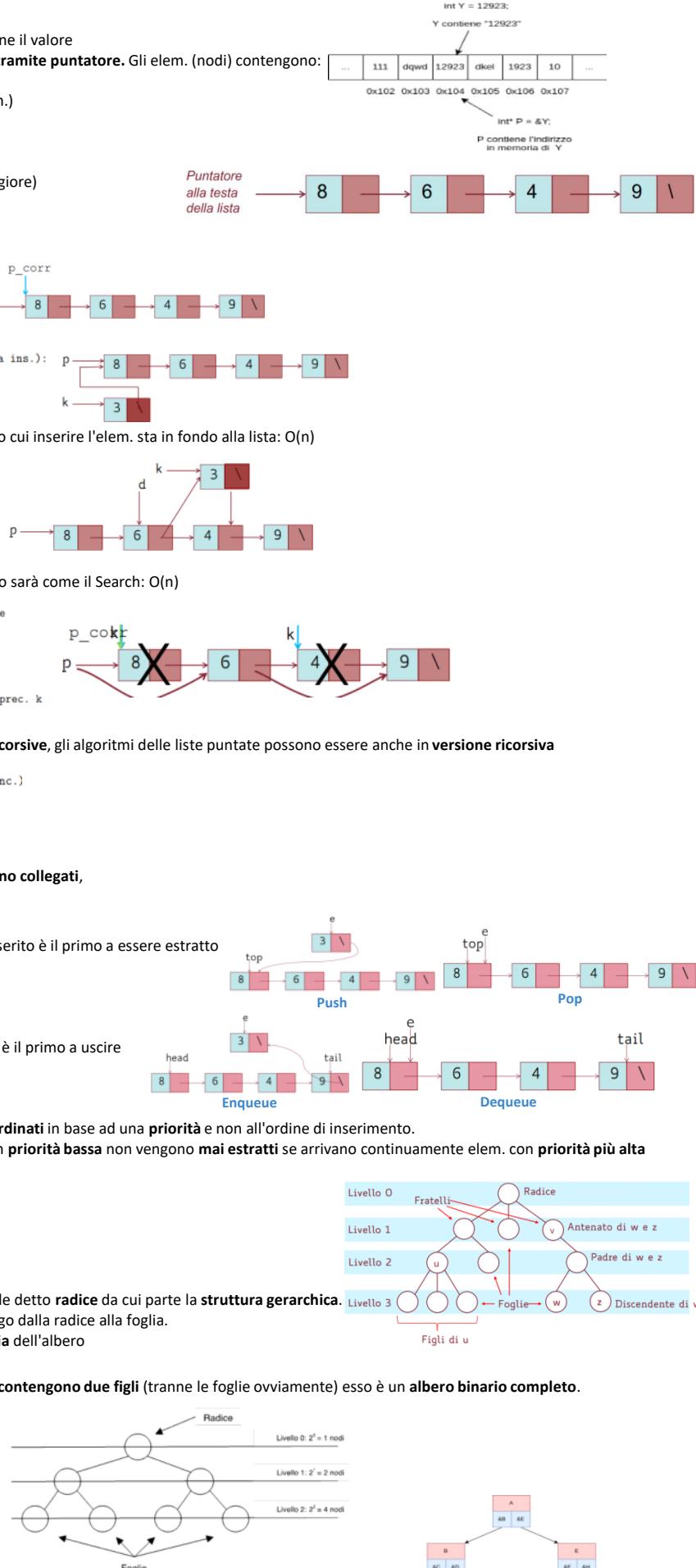
## Alberi Binari

In un **albero binario** ogni nodo ha **massimo due figli**. Se tutti i livelli **contengono due figli** (tranne le foglie ovviamente) esso è un **albero binario completo**.

- L'**altezza** dell'albero è h
- Il **num. di nodi ad ogni livello** è  $2^i$ , dove i è il liv.
- Il **num. di foglie** è  $2^h$
- Il **num. di nodi interni** (nodi non foglie) è  $2^h - 1$
- Il **num. totale di nodi** è  $n = 2^{h+1} - 1$
- L'altezza di un albero in base al num. di nodi è  $h = \log\left(\frac{n+1}{2}\right)$

Quindi l'**altezza di un albero completo** è  $h = \Theta(\log(n))$

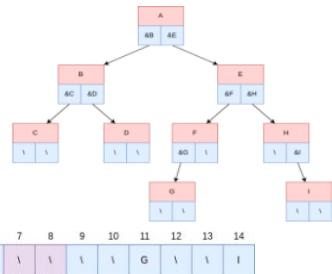
Nel caso di un **albero non completo** è  $h = O(n)$



- L'altezza di un albero in base al num. di nodi è  $h = \log\left(\frac{n+1}{2}\right)$
- Quindi l'**altezza di un albero completo** è  $h = \Theta(\log(n))$   
Nel caso di un **albero non completo** è  $h = O(n)$



Albero completo di altezza 2 (tre livelli)



Poiché l'indice di C è 3, i suoi figli saranno in posizione  $2 \times 3 + 1$  e  $2 \times 3 + 2$

Un albero si può rappresentare in tre modi:

- Tramite puntatori:** Ogni nodo è un record con
  - Key** (informazioni del nodo stesso)
  - Un puntatore al figlio sinistro e uno al destro** (o None se non ci sono)**Vantaggi:** flessibile per inserimenti e modifiche  
**Svantaggi:** per trovare un nodo bisogna scendere partendo dalla radice
- Tramite indici posizionali:** similmente all'Heap, i figli del nodo  $i$  sono in  $2*i + 1$  e  $2*i + 2$ .  
**Vantaggi:** Accesso rapido ai figli  
**Svantaggi:** richiede di conoscere in anticipo l'**altezza massima** e vi è uno **spreco di mem.** se l'albero non è completo
- Tramite vettore dei padri:** si usano **due array**:  $A[i]$  contiene il **val.** e  $P[i]$  l'**indice del padre del nodo  $i$ .**  
**Vantaggi:** facile trovare il padre  
**Svantaggi:** complesso trovare i figli

### Confronto tra rappresentazioni

- Trovare il padre di un nodo**
  - Puntatori:** non siamo in grado di farlo senza implementare un algoritmo complesso (**visita dell'albero**)
  - Posizionale:** il padre del nodo  $i$  è in pos  $\left[\frac{i-1}{2}\right]$ :  $\Theta(1)$
  - Vettore:** l'indice del padre del nodo  $i$  è memorizzato in  $P[i]$ :  $\Theta(1)$
- Determinare se un nodo abbia 0, 1 o 2 figli**
  - Puntatori:** controllo se i campi **left** e **right** sono settati a **None**:  $\Theta(1)$
  - Posizionale:** controllo se gli elementi  $2i+1$  e  $2i+2$  sono settati a ' $-1$ ':  $\Theta(1)$
  - Vettore:** scorro l'array  $P$  e si conta il num. di occorrenze di  $i$ :  $\Theta(n)$
- Determinare la distanza di un nodo dalla radice**
  - Puntatori:** non si può fare senza implementare un algoritmo complesso (**visita dell'albero**)
  - Posizionale:** il liv. del nodo  $i$  (e quindi la sua distanza dalla radice) è  $\lfloor \log(i+1) \rfloor$ :  $\Theta(1)$
  - Vettore:** a partire da  $i$  risaliamo di padre in padre passando per  $P[i], P[P[i]], P[P[P[i]]], \dots$  fino alla radice:  $\Theta(h)$

Operazione	Puntatori	Posizionale	Vettore
Trovare padre	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Verifica figli	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$
Distanza da radice	$\Theta(n)$	$\Theta(1)$	$\Theta(h)$

### Visite di un Albero

L'accesso progressivo a tutti i nodi dell'albero si chiama **visita dell'albero**, e si può effettuare in tre modi:

- Visita preordine:** si accede al nodo **prima** della visita nei suoi **sottoalberi**
- Visita inordine:** si accede al nodo **dopo** la visita del **sottoalbero sinistro** e **prima** la visita del **sottoalbero destro**
- Visita postordine:** si accede al nodo **dopo** la visita nei suoi **sottoalberi**

```
def visita_preordine(p):
    access(p)
    visita_preordine(p.left)
    visita_preordine(p.right)

def visita_inordine(p):
    visita_inordine(p.left)
    access(p)
    visita_inordine(p.right)

def visita_postordine(p):
    visita_postordine(p.left)
    visita_postordine(p.right)
    access(p)
```

### Costo computazionale

Il **costo delle visite** è lo stesso e varia in base alla struttura dell'albero.

Nel caso di record e puntatori, abbiamo  $k = \text{num. nodi sottoalbero sinistro} + n-k-1 = \text{num. nodi sottoalbero destro}$

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

- Caso migliore (albero completo):** ogni nodo ha esattamente due figli,  $n = 2^{h+1} - 1$  nodi

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

Ricadiamo nel **caso 1 del teorema principale**, quindi costo  $\Theta(n)$

- Caso peggiore (albero sbilanciato):** tutti i nodi sono in un **unico ramo**,  $n-k-1 = 0$  (se sono in quello sinistro) o  $k = 0$  (se sono in quello destro). In entrambi i casi si ha:  $T(n) = T(n-1) + \Theta(1)$   
Usando il **metodo iterativo**, il costo è  $n * \Theta(1) = \Theta(n)$
- Caso medio:** poiché in entrambi i casi il costo è  $\Theta(n)$ , possiamo ipotizzare che il suo costo è  $\Theta(n)$ , dato che ogni nodo viene visitato una sola volta e  $n$  è il num. di nodi

Dobbiamo usare il **metodo della sostituzione** per verificare con certezza che il costo è  $\Theta(n)$

### Applicazione delle visite

- Conteggio dei nodi (visita postordine):** Somma i nodi dei due sottoalberi e li aggiunge alla radice

```
def Calcola_n(p):
    if p != None:
        num_l = Calcola_n(p.left) # nodo sinistro
        num_r = Calcola_n(p.right) # nodo destro
        num = num_l + num_r + 1 # accesso al nodo
    return num
return 0
```

compresso

```
def Calcola_n(p):
    if p != None:
        return Calcola_n(p.left) + Calcola_n(p.right) + 1
    return 0
```

- Ricerca di un nodo (visita preordine):** Controlla il nodo attuale prima di scendere nei sottoalberi

```
def Cerca(p, k):
    if p != None:
        if p.info == k: return True
        else:
            if Cerca(p.left, k) == True:
                return True
            else:
                return Cerca(p.right, k)
    return False
```

compresso

```
def Cerca(p, k):
    if p != None:
        if p.info == k: return True
        else:
            return Cerca(p.left, k) or Cerca(p.right, k)
    return False
```

- Calcolo dell'altezza (visita postordine):** Trova l'altezza massima dei sottoalberi e aggiunge 1

```
def Calcola_h(p):
    if p == None:
        return -1
    if p.left == None and p.right == None:
        return 0
    h = max(Calcola_h(p.left), Calcola_h(p.right)) + 1
    return h
```

compresso

```
def Calcola_h(p):
    if p != None:
        return max(Calcola_h(p.left), Calcola_h(p.right)) + 1
    return -1
```

```

        if p.left == None and p.right == None:
            return 0
        h = max(Calcola_h(p.left), Calcola_h(p.right)) + 1
        return h
    
```

```

        return max(calcola_h(p.left), calcola_h(p.right)) + 1
    return -1
    
```

## Alberi Binari di Ricerca

Un **albero binario di ricerca (ABR)** è una struttura in cui vengono mantenute queste **priorità**:

- Il **sottoalbero sinistro** contiene solo val. **minori** della chiave del nodo
- Il **sottoalbero destro** contiene solo val. **maggiori** della chiave del nodo
- Il **minimo** si trova nel **nodo più a sinistra**, il **massimo** nel **nodo più a destra**

Se l'ABR è:

- Bilanciato**, l'altezza è  $O(\log(n))$ , garantendo ricerca e op. efficienti
- Sbilanciato**, l'altezza è  $O(n)$ , rendendo l'op. inefficienti

## Operazioni sugli ABR

- Ordinamento dei nodi:** Per stampare i nodi in **ordine crescente** possiamo eseguire una **visita inordine** ( $\Theta(n)$ )

- Ricerca di un nodo:** Similmente alla ricerca binaria, si segue un percorso **guidato dai criteri di ricerca ricorsiva**

```

def ABR_search_ricorsiva(p, k):
    if p == None or p.key == k:
        return p
    if k < p.key:
        return ABR_search_ricorsiva(p.left, k)
    else:
        return ABR_search_ricorsiva(p.right, k)
    
```

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

Col **metodo iterativo**: costo  $\Theta(h)$

```

def ABR_search_iterativa(p, k):
    while p != None and p.key != k:
        if k < p.key:
            p = p.left
        else:
            p = p.right
    return p
    
```

### Ricerca Iterativa

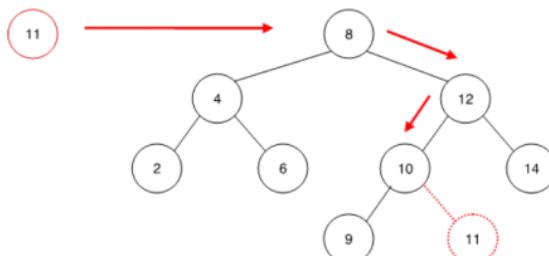
Si esegue al massimo **h volte**.  
Costo  $O(h)$

- Inserimento di un nodo:** Come nella ricerca, si scende in **base ai confronti** fino a trovare un **puntatore None** e si inserisce il nuovo nodo

```

def ABR_insert(p, z):
    x, y = p, None # x è la radice dell'albero, y è il padre di x
    while x != None: # discesa alla prima pos. disponibile
        y = x # y punta sempre al padre di x
        if z.key < x.key: # se z è nel sottoalbero SX di x
            x = x.left # x diventa il figlio SX di x
        else: # se z è nel sottoalbero DX di x
            x = x.right # x diventa il figlio DX di x
        z.parent = y # collegiamo al padre il nodo da inserire
    if y == None: # l'albero è vuoto
        p = z # z è la radice
    elif z.key < y.key: # z va a SX
        y.left = z # z è il figlio SX di y
    else: # z va a destra
        y.right = z # z è il figlio DX di y
    return p # restituisce la radice dell'albero
    
```

Ciclo eseguito **h volte**  $\rightarrow O(h)$

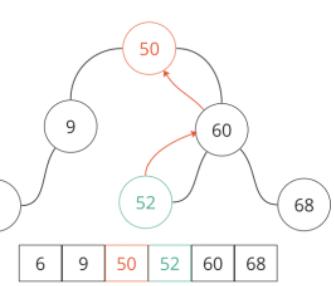
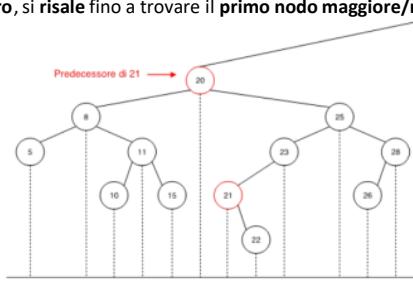


- Ricerca di Minimo/Massimo:** Si scende a **sinistra/destra** fino a trovare un nodo senza figlio corrispondente

- Ricerca di Predecessore/Successore:** Si possono verificare 2 casi:

- Se il nodo ha un **sottoalbero sinistro/destro**, il **predecessore/successore** è il **massimo/minimo** di quel sottoalbero
- Se il nodo non ha un **sottoalbero sinistro/destro**, si risale fino a trovare il **primo nodo maggiore/minore**

Costo  $O(h)$



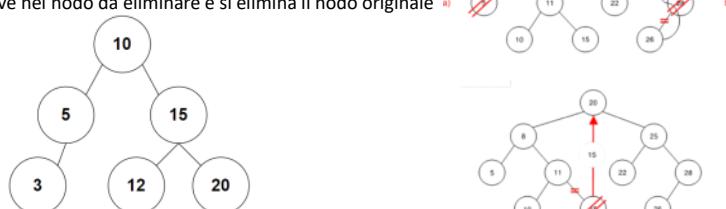
- Eliminazione di un nodo:** Si possono verificare 3 casi:

- Nodo senza figli:** viene rimosso il nodo ponendo **None** nel campo del suo **nodo padre**
- Nodo con un solo figlio:** Si "cortocircuita", collegando il figlio al **padre** del nodo eliminato
- Nodo con due figli:** Si trova il **successore o predecessore**, si copia la chiave nel nodo da eliminare e si elimina il nodo originale

Costo  $O(h)$  (visita + sostituzione)

```

def ABR_delete(p, z):
    # CASO 1 o 2
    if z.left == None or z.right == None: # z non ha figli o ne ha uno
        y = z # z è il nodo da eliminare
    # CASO 3
    else: # z ha due figli
        y = ABR_successore(z) # il nodo da eliminare è il successore di z
        # Ora y punta al nodo da eliminare
        if y.left != None: # y ha un figlio SX
            x = y.left # x è il figlio SX di y
        else: # y non ha figli SX
            x = y.right # x è il figlio DX di y
        # y non può avere due figli, x punterà al figlio o sarà None
        # CASO 2: Se y ha un figlio, lo collegiamo al padre di y
        if x != None: # se x esiste, quindi y ha un figlio
            x.parent = y.parent # il padre di x diventa il padre di y
        if y.parent == None: # y è la radice
            p = x # si: x diventa la radice
        else: # y non è la radice
            if y == y.parent.left: # se y era il figlio SX
                y.parent.left = x
            else: # se y era il figlio DX
                y.parent.right = x
    
```



### Esempio di Cancellazione:

- Caso 1: il nodo 12 non ha figli, basta **rimuoverlo** e aggiornare il puntatore del padre (15) a **None**
- Caso 2: il nodo 5 ha un solo figlio, 5 viene **eliminato** e suo figlio (3) viene **collegato al padre** di 5 (10)
- Caso 3: il nodo 15 ha due figli, si trova il **successore** (20), si **copia 20 in 15** e si **elimina il nodo 20**.

Dove si trovano i casi nel codice?

- Caso 1: y viene semplicemente **scollegato** dall'ARB
- Caso 2: il nodo viene "cortocircuitato" sostituendolo col figlio
- Caso 3: si **elimina il successore** e si **copia la sua chiave nel nodo da eliminare**

```

if y.parent == None: # y è la radice?
    p = x # si: x diventa la radice
else: # y non è la radice
    if y == y.parent.left: # se y era il figlio SX
        y.parent.left = x # x prende il posto di y
    else: # se y era il figlio DX
        y.parent.right = x # x prende il posto di y
# CASO 3: Se y è il successore di z, copiamo la chiave di y in z
if y != z: # y è il successore di z
    z.key = y.key # copia la chiave di y in z
return p

```

Dove si trovano i casi nel codice?

- **Caso 1:** y viene semplicemente **scollegato** dall'ARB
- **Caso 2:** il nodo viene "**cortocircuitato**" sostituendolo col figlio
- **Caso 3:** si elimina il **successore** e si copia la sua chiave nel nodo da eliminare

# Riassunto Eq. principali

lunedì 17 febbraio 2025 17:31

## Ricerca Bin

- **Caso migliore:** v è al centro ( $\Omega(1)$ )
- **Caso peggiore:** l'array viene dimezzato fino a un singolo elem. ( $O(\log(n))$ )
- **Caso medio:** simile al caso peggiore, poiché il num. di iter. è legato al dimezzamento progressivo dell'array ( $\Theta(\log(n))$ )

## Ricerca Binaria Sequenziale

Ad ogni iter. di ricerca l'intervallo di ricerca viene dimezzato quindi il num. max di iter. è  $\frac{n}{2^k} \Rightarrow \log_2(n)$

## Ricerca Binaria Ricorsiva

Ad ogni iter. abbiamo due casi per la chiamata:

- Se il val. da cercare è maggiore del val. a metà allora dobbiamo eseguire la chiamata ricorsiva sulla metà superiore dell'array
- Invece se il val. da cercare è minore del val. a metà allora dobbiamo eseguire la chiamata ricorsiva sulla metà inferiore dell'array.

In ogni iter. quindi eseguiamo la chiamata una sola volta e dimezziamo l'input della chiamata, quindi:

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

Usando il metodo principale o l'iterativo possiamo vedere che il costo è  $\Theta(\log(n))$

## Riassunto Costo Ordinamenti Principali

- **Merge Sort:**
  - **Costo**  $\Theta(n * \log(n))$
  - $T = \begin{cases} T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \\ T(1) = \Theta(1) \end{cases}$
  - **Fondi:**  $\Theta(n)$
  - Due chiamate ricorsive sulla metà dell'array
- **Quick Sort:**
  - **Costo migliore e medio:**  $\Theta(n * \log(n))$
  - **Costo peggiore:**  $\Theta(n^2)$
  - $T = \begin{cases} T(n) = T(k) + T(n - k) + \Theta(n) \\ T(1) = \Theta(1) \end{cases}$  la 1° seq. ha k elem., la 2° ne ha n-k
  - **Partiziona:**  $\Theta(n)$
  - Due chiamate ricorsive sulla 1° e 2° sequenza
- **Heap Sort:**
  - **Costo:**  $\Theta(n * \log(n))$
  - **Build Heap:**  $O(n)$
  - **Heapify:**  $O(\log(n))$
  - Richiama Heapify sugli n elem. dell'Heap
- **Counting Sort:**
  - **Costo:**  $O(n+k)$  dove k è il val. massimo dell'array  
Utile solo se k è lineare su n
- **Bucket Sort:**
  - **Costo migliore e medio:**  $\Theta(n)$
  - **Costo peggiore:**  $\Theta(n^2)$

# Logaritmi

martedì 11 marzo 2025 22:27

$$\log_b(x) = y \Leftrightarrow b^y = x$$

$$a^b = 2^{b * \log_2(a)}$$

$$\log_b(x) = \frac{\log_k(x)}{\log_k(b)} \text{ (utile per convertire tra basi diverse, es: } \log_2(x) = \frac{\ln(x)}{\ln(2)})$$

$$\log_b(xy) = \log_b(x) + \log_b(y)$$

$$\log_b\left(\frac{x}{y}\right) = \log_b(x) - \log_b(y)$$

$$\log_b(\sqrt[n]{x}) = \frac{1}{n} * \log_b(x)$$

$$\log_b(x^y) = y * \log_b(x)$$

$$2^{\log_b(n)} = n^{\log_b(2)}$$

$$\log_a(b) = \begin{cases} > 1 & \text{se } a < b \\ 1 & \text{se } a = b \\ < 1 & \text{se } a > b \mid \text{base} < \end{cases}$$

# Sommatorie Importanti

giovedì 24 ottobre 2024 18:38

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

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

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

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

$$\sum_{i=0}^n i2^i = \theta(n2^n)$$

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

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

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

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

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

$$\sum_{i=0}^n r^i = \frac{1-r^{n+1}}{1-r}$$

$$\sum_{i=0}^n \frac{1}{2^i} = 2 - \frac{1}{2^n}$$

quando  $n$  cresce  $\frac{1}{2^n} \rightarrow 0$ , dunque  
 $S(n) = 2 = \Theta(1)$

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

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

$$\sum_{i=a}^n c = c * n - a + 1$$

# Algoritmi utili

giovedì 13 febbraio 2025 16:26

## Ricerca Binaria

```
def Ricerca_Binaria(A, v):
    a = 0 # il primo indice di A
    b = len(A) - 1 # l'ultimo indice di A
    m = (a+b)//2 # l'indice a metà di A
    while A[m] != v: # finché non trovo v
        if A[m] > v: # se v è minore di A[m]
            b = m - 1 # prendo la metà inferiore
        else: # se v è maggiore di A[m]
            a = m + 1 # prendo la metà superiore
        if a > b: # se v non è in A
            return -1
        m = (a+b)//2 # ricalcolo il nuovo valore di m
    return m
```

```
def Ricerca_Binaria_seq(A, v):
    if len(A) == 0: # se A è vuoto
        return -1
    m = len(A)//2 # calcolo l'indice a metà di A
    if A[m] == v: # se v sta a metà
        return m
    if v < A[m]: # se v si trova nella metà inferiore
        return Ricerca_Binaria_seq(A[:m], v)
    else: # se v si trova nella metà superiore
        return Ricerca_Binaria_seq(A[m+1:], v)
```

## Confronto tra Elementi di Array Ordinati con n Interi Distinti

Se si hanno due array A e B **ordinati** e con **n interi distinti** e si deve eseguire un confronto sugli elem. dei due array per controllare una certa proprietà (es. intersezione, unione, differenza), l'algoritmo più efficiente è quello che utilizza **due indici** per **scorrere simultaneamente entrambi gli array**, confrontando gli elem. ed eseguendo le op. desiderate in tempo lineare.

### Struttura dell'Algoritmo Generale:

- 1) **Inizializzazione:** imposta due indici a = 0 e b = 0 per i rispettivi array A e B
- 2) **Scorrimento simultaneo:**
  - Mentre entrambi gli indici non hanno raggiunto la fine degli array (while a < n and b < n)
    - **Confronto:** Se A[a] e B[b] soddisfano le proprietà richieste, elabora l'elem.
    - **Avanzamento:**
      - Se A[a] < B[b]: incrementa a
      - Se A[a] > B[b]: incrementa b
- 3) **Terminazione:** il ciclo termina quando almeno uno dei due array è stato completamente scandito

Poiché ogni elem. viene esaminato al massimo una volta, ogni iter è costante e il costo complessivo è O(n)

Questo metodo si chiama **two pointer technique** e si può usare anche su anche tre array o anche su un array solo, magari per scambiare gli elem. dividendo in due parti l'array in base a una certa proprietà oppure controllare se esiste una coppia con una certa proprietà

## Ricerca val. max in un array diviso in due parti

Abbiamo un array ordinato con n interi distinti ed è diviso in due parti in cui la prima parte è crescente e la seconda è decrescente (es. [5, 7, 9, 4, 2] oppure i primi k elem. sono ruotati a destra (es. [2, 4, 6, 7, 9] → [6, 7, 9, 2, 4]).

Per trovare il val. massimo ci basta trovare il punto in cui le due parti si incontrano (es. nel primo esempio tra 9 e 4), possiamo usare una ricerca binaria così il costo sarà O(log(n)).

```
def Bin_max_iter(A):
    s, d = 0, len(A) - 1
    while True:
        m = (s+d)//2
        if A[m-1] > A[m]: # se l'elem prec. è più grande del corrente
            # allora il max è m-1
            return m-1
        if A[s] < A[m]: # dobbiamo ancora trovare il max
            # il max è a destra
            s = m+1 # ricontrolliamo da [m+1, d]
        else: # abbiamo superato il max
            # il max è a sinistra
            d = m # ricontrolliamo da [s, m]
```

## Scorrimento di una Lista di Puntatori

```
def visita_lista_ric(n):
    if n is None: # Caso base: fine della lista
        return
    print(n.val) # Visita il nodo corrente
    visita_lista_ric(n.next) # Visita il nodo successivo
```

```
def visita_lista_iter(n):
    while n is not None:
        print(n.val) # Visita il nodo corrente
        n = n.next # Passa al nodo successivo
```

## Algoritmi per l'ordinamento

Se l'esercizio richiede:

- **Tempo**  $\Theta(n)$  e gli elem. sono lineari su n allora possiamo usare direttamente **CountingSort()**.  
Se invece abbiamo un range max del val. degli elem. che dobbiamo usare (ad es. 100) possiamo usare una **versione modificata** del **counting sort**, in cui:
  - Creiamo un array C di x elem. (nel nostro caso 100), dove x deve essere una costante
  - Scorriamo l'array originale e se un elem.  $A[i] < x$  allora:
    - Se A ha elem. distinti: possiamo direttamente salvare nell'array C gli elem. ( $C[A[i]] = A[i]$ )

Esempio:

Dato un array A di n interi non negativi distinti, si vuole determinare se esistono almeno tre numeri consecutivi di valore inferiore a 100. Ad esempio, se A = [101, 5, 9, 31, 33, 10, 100, 4, 8, 32, 500, 11, 99], gli elementi 8, 9 e 10 così come gli elementi 31, 32 e 33 rispettano la proprietà a mentre 99, 100 e 101 no.

```
def Exam2_20_3_23(A):
    C[0]*100 # inizializziamo le 100 pos di C (che va da 0 a 99)
    for i in range(len(A)):
        if A[i] < 100: # inseriamo solo i val. minori di 100
            C[A[i]] += 1 # aumentiamo l'elem. di C in pos A[i] di 1
    for i in reversed(range(1, 98)):
        if C[i-1] == 1 and C[i] == 1 and C[i+1] == 1: # se la terna è valida
            return A[i] # ritorniamo l'elem. centrale
    return -1 # altrimenti se non abbiamo trovato una terna valida ritorniamo -1
```

# Altri argomenti importanti

giovedì 13 febbraio 2025 11:06

Argomenti basati su tutti gli esami dal 2017 al 2025 dando priorità a quelli più nuovi

## 1. Algoritmi di Ricorrenza e loro Risoluzione

Molti esercizi richiedono di risolvere equazioni di ricorrenza utilizzando vari metodi:

- **Metodo iterativo**
- **Metodo dell'albero di ricorsione**
- **Metodo di sostituzione**
- **Teorema principale (Master Theorem)**

È utile saper riconoscere il tipo di ricorrenza e applicare il metodo più adatto.

## 2. Algoritmi di Ordinamento

Molti esami richiedono di implementare, analizzare o confrontare algoritmi di ordinamento:

- **Merge Sort** (spesso con variazioni, come divisione in 4 sottovettori)
- **Quick Sort** (inclusa la funzione di partizione, tipo la modifica del pivot)
- **Selection Sort** (specialmente applicato a liste concatenate)
- **Heap Sort** (con costruzione e operazioni su heap)

## 3. Strutture Dati Fondamentali

Gli esercizi coinvolgono frequentemente:

- **Heap massimo** e operazioni come inserimento, eliminazione della radice e fusione di due heap.
- **Alberi binari di ricerca (ABR)**, con operazioni come inserimento, cancellazione, ricerca, successore e predecessore.
- **Liste concatenate**, con implementazioni di funzioni su liste come ordinamento, ricerca, somma di elementi e gestione di cicli (rilevamento di liste circolari o lassos).

## 4. Algoritmi su Matrici

- **Trova il massimo locale in una matrice.**

Un **massimo locale** in una matrice è un elemento che è strettamente maggiore di tutti i suoi vicini immediati (su, giù, sinistra, destra).

**Approccio naïve O(nm):**

- a. Scansiona tutta la matrice e per ogni elemento  $A[i][j]$ , confrontalo con i suoi 4 vicini.
- b. Se è maggiore di tutti i vicini, segna la posizione come massimo locale.

**Approccio più efficiente con ricerca binaria O( $n^*log(m)$ ):**

- o Se la matrice è **montuosa** (elementi che crescono e poi decrescono), si può usare **ricerca binaria sulle colonne** per trovare il massimo locale più velocemente.

- **Trova punti panoramici (massimi in riga e colonna).**

Un **punto panoramico** in una matrice è un elemento che è il massimo nella sua riga e nella sua colonna.

**Algoritmo O(nm):**

1. Per ogni riga, trova il massimo.
2. Per ogni colonna, trova il massimo.
3. Interseca i risultati per trovare eventuali punti in comune.

**Ottimizzazione O(n+m):**

- Se si precomputano gli array dei massimi per ogni riga e colonna, si può ridurre la complessità.

- **Calcolo delle medie mobili di periodo k** in una matrice o in un array.

La media mobile di periodo k è la media dei valori in una finestra scorrevole di lunghezza k

**Algoritmo per array O(n) (Sliding Window con somma accumulata):**

1. Calcola la somma dei primi k elementi.
2. Per ogni elemento successivo, aggiorna la somma sottraendo il primo valore della finestra precedente e aggiungendo il nuovo valore.

**Algoritmo per matrici O(nm):**

- Estendi lo stesso approccio alle righe e colonne, usando somme cumulative 2D.

- **Visibilità dei grattacieli in una matrice** (calcolo degli edifici visibili da ogni lato).

Se la matrice rappresenta un insieme di grattacieli, possiamo calcolare quanti sono visibili da ciascun lato (sinistra, destra, sopra, sotto).

**Approccio O(nm):**

1. Per ogni riga, trova i massimi visibili da sinistra e da destra.
2. Per ogni colonna, trova i massimi visibili dall'alto e dal basso.
3. Somma i valori ottenuti.

**Utilizzo di stack monotoni per ridurre la complessità a O(nm)**

- Mantieni uno stack crescente/decrecente per calcolare i punti di visibilità più rapidamente.

## 5. Algoritmi di Selezione e Ricerca

- **Trovare il minimo intero non presente in un array** con vincoli di spazio.

Dato un array di nnn numeri interi positivi, trovare il **più piccolo intero positivo che non compare** nell'array.

**Approccio naïve O( $n*log(n)$ ):**

- Ordina l'array e scorri fino a trovare il primo buco.

**Approccio O(n) senza memoria extra:**

- Mappa ogni valore all'indice corretto (bucket sort modificato) e trova il primo indice errato.

- **Selezione del suffisso lungo k di una lista concatenata.**

Dato un numero k, trovare gli ultimi k elementi di una lista concatenata.

**Approccio naïve O(n):**

1. Conta il numero di nodi nella lista.
2. Scorri la lista fino al nodo n-k.

**Approccio con due puntatori O(n):**

1. Sposta un puntatore k passi avanti.
2. Muovi entrambi i puntatori finché il primo raggiunge la fine.
3. Il secondo puntatore sarà all'inizio del suffisso.

- **Ricerca del valore maggioritario in un array** (versione quadratica e  $O(n \log n)$ ).

Un valore maggioritario è un valore che compare più di  $n/2$  volte in un array.

#### **Approccio $O(n \log n)$ con ordinamento:**

1. Ordina l'array.
2. Controlla la frequenza dell'elemento in posizione  $n/2$ .

#### **Approccio $O(n)$ con l'algoritmo di Boyer-Moore:**

1. Usa un contatore per eliminare coppie di elementi differenti.
2. Verifica se il candidato trovato è effettivamente maggioritario.

## 6. Tecniche di Programmazione

- **Divide et Impera**, applicato a problemi come QuickSort modificato o ricerca binaria su array speciali.
- **Backtracking**, per problemi di ricerca in alberi o grafi.

- **Ricerca binaria su array speciali**

Se un array ha proprietà particolari, possiamo applicare la **ricerca binaria modificata**:

- **Trova il massimo in un array montuoso**: cerca il punto in cui  $A[i-1] < A[i] > A[i+1] A[i-1] < A[i] > A[i+1] A[i-1] < A[i] > A[i+1]$ .
- **Ricerca in un array ruotato**: dividi in due parti e determina in quale metà si trova il valore cercato.
- Ricerca massimo in una matrice, possiamo trovare il massimo tempo  $O(n \log m)$  invece di  $O(nm)$

## 7. Gestione della Memoria

Molti esercizi richiedono:

- Allocazione dinamica e gestione di puntatori.
- Modifica di strutture dati senza allocare memoria aggiuntiva.
- Implementazione di funzioni che non alterano le strutture dati originali.

# Introduzione

Friday, October 4, 2024 8:58 PM

Un **algoritmo** è una "sequenza di comandi elementari ed univoci che terminano in un **tempo finito** e operano su **strutture dati**"

Un comando è "**elementare**" quando non può essere scomposto in comandi più semplici.

La scelta della **struttura dati** è fondamentale per la risoluzione del problema stesso.

Così come la scelta dell'**algoritmo** poiché **cruciale per l'efficienza** del programma in termini di operazioni elementari e quantità di spazio di mem. usato in funzione della dimensione dell'input

## Random Access Machine (RAM)

Le caratteristiche HW dell'elaboratore **influiscono sulle performance dell'algoritmo**. Per poter valutare la vera efficienza di un algoritmo, è necessario valutarlo come se venisse eseguito da una **macchina astratta**.

La **Random Access Machine (RAM)** è una macchina astratta, la cui validità e potenza concettuale risiede nel fatto che **non diventa obsoleta**.

Le caratteristiche del modello RAM sono:

- Un **singolo processore** che esegue le operazioni **sequenzialmente**
- Esistono **solo operazioni elementari** e l'esecuzione di ciascuna delle quali richiede per definizione un **tempo costante** (es. op. aritmetiche, letture, scritture, salto condizionato, ecc.)
- Esiste un **limite alla dimensione** di ogni valore memorizzato ed al numero complessivo di valori utilizzati (il max valore rappresentabile in mem. non può superare 2 elevato al num. di bit della word(32 o 64))

## Misura di costo uniforme

Se è soddisfatta l'ipotesi che ogni dato in input sia minore di  $2^k$  dove k è il **n. bit della word in mem.**, ciascuna operazione elementare sui dati del problema verrà eseguita in un tempo costante. In tal caso si parla di **misura di costo uniforme**.

Se però un dato del problema non rispetta tale ipotesi, esso dovrà comunque essere memorizzato e sarà necessario usare **più parole di memoria** e quindi anche le operazioni elementari su di esso dovranno essere reiterate per tutte le parole di mem. che lo contengono, richiedendo un tempo non più costante: Calcolo scientifico e misura di costo logaritmico.

Esempio costo uniforme

```
def PotenzaDi2(n):  
    x = 1  
    for i in range(n):  
        x = x*2  
    return x
```

Il tempo di esecuzione totale è **proporzionale a n**:

- **Ciclo eseguito n volte**
- Ad ogni iterazione si compiono due op., ciascuna con **costo unitario**:
  - l'incremento del contatore
  - Il calcolo del nuovo valore di x

# Notazione O grande

giovedì 24 ottobre 2024 18:25

## Notazione Asintotica

Per confrontare le efficienze degli algoritmi che risolvono lo stesso problema dobbiamo valutare il loro **costo computazionale** (il tempo di esecuzione di un algoritmo e delle sue necessità in termini di mem.)

In matematica la **notazione asintotica** permette di confrontare il **tasso di crescita** (comportamento asintotico) di una funzione nei confronti di un'altra.

In informatica, il calcolo asintotico è usato per analizzare il **costo di un algoritmo**, in particolar modo per stimare **quanto aumenta il tempo di esecuzione al crescere delle dimensioni n dell'input**.

- **Notazione asintotica O grande:** limite superiore asintotico
- **Notazione asintotica Ω (Omega):** limite inferiore asintotico
- **Notazione asintotica Θ (Teta):** limite asintotico stretto

## Notazione O grande

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

se esistono due costanti  $c$  ed  $n_0$  t.c.

$$0 \leq f(n) \leq c \cdot g(n) \quad \forall n \geq n_0$$

È importante che  $f(n)$  da un certo punto in poi sia sotto  $g(n)$ , quindi  $f(n)$  è in  $O(g(n))$

È importante trovare la **più piccola  $g(n)$**  per determinare O

In  $O(g(n))$  troviamo **tutte** le funzioni che risultano "dominate" da  $g(n)$

La notazione **O grande**, definisce quello che è il **limite superiore asintotico** di  $f(n)$ :

Una volta superato un valore  $n_0$  (dove  $n \rightarrow \infty$ ) l'andamento di  $f(n)$  viene **limitato** da  $c \cdot g(n)$ , rimanendo sempre **al di sotto di essa** (viene "dominata" da essa)

Esempio:

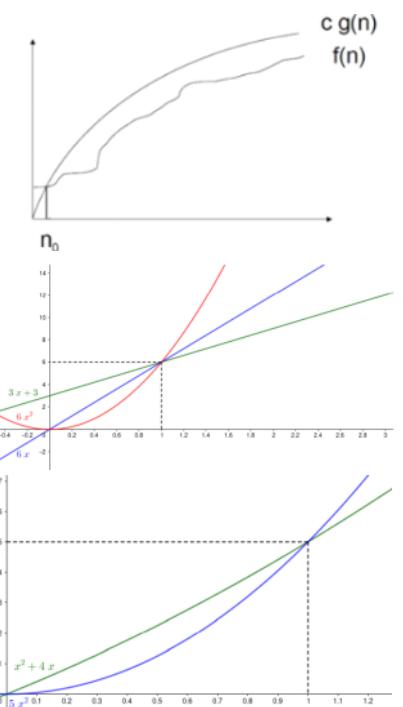
- $f(n) = 3n+3$   
 $f(n)$  è in  $O(n^2)$  in quanto, posto  $c=6$ :  
 $cn^2 \geq 3n+3 \quad \forall n \geq 1 \quad (6 \cdot 1^2 \geq 3+3)$   
Ma  $f(n)$  è anche in  $O(n)$  in quanto:  
 $Cn \geq 3n+3 \quad \forall n \geq 1 \text{ se } c \geq 6, \quad (6 \cdot 1 \geq 3+3)$   
oppure  $\forall n \geq 3$  se  $c \geq 4$  ( $4 \cdot 3 \geq 3 \cdot 3 + 3$ )  
data  $f(n)$ , esistono **infinte funzioni  $g(n)$**  per cui  $f(n)$  risulta in  $O(g(n))$ .  
•  $f(n) = n^2 + 4n$   
 $f(n)$  è in  $O(n^2)$  in quanto:  
 $cn^2 \geq n^2 + 4n \quad \forall n \geq 1 \text{ se } c \geq 5 \quad (5 \cdot 1 \geq 1+4 \cdot 1)$   
•  $\log_2^{a,b}(n)$  è in  $O(n^{1/b})$   $\forall a, b \geq 1$   
•  $f(n) = n^{1/a}$  è in  $O(n)$   $\forall a \geq 2$   
•  $f(n) = n^2$  è in  $O(b^n)$   $\forall b \geq 2$

Nel primo esempio vediamo che un **polinomio di primo grado** sia in  $O(n)$ , mentre un **polinomio di secondo grado** sia in  $O(n^2)$ . possiamo generalizzare la cosa nel teorema:

Sia  $f(n)$  un **polinomio** di grado  $m$ , definito come:

$$f(n) = \sum_{i=0}^m a_i n^i = a_0 + a_1 n + a_2 n^2 + \dots + a_m n^m$$

possiamo concludere che  $f(n)$  è in  $O(n^m)$



# Notazione Omega

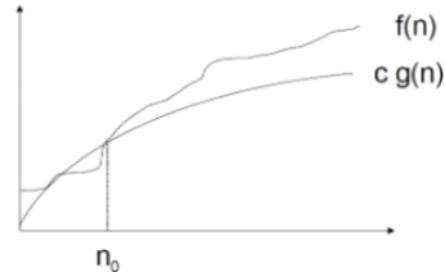
giovedì 24 ottobre 2024 18:28

## Notazione $\Omega$

Date due funzioni  $f(n), g(n) \geq 0$  si dice che  $f(n)$  è in  $\Omega(g(n))$  se esistono due costanti  $c$  e  $n_0$  t.c.

$$f(n) \geq c*g(n) \quad \forall n \geq n_0$$

È importante che  $f(n)$  da un certo punto in poi sia sopra  $g(n)$ , quindi  $f(n)$  è in  $O(g(n))$   
È importante trovare la più grande  $g(n)$  per determinare  $\Omega$



In  $\Omega(g(n))$  troviamo tutte le funzioni che "dominano" la funzione  $g(n)$

La notazione **Omega** definisce il **limite inferiore asintotico** di  $f(n)$ :

una volta superato un valore  $n_0$  (dove  $n \rightarrow \infty$ ) l'andamento di  $f(n)$  viene **limitato** da  $c*g(n)$ , rimanendo sempre **al di sopra essa** ("domina" essa)

Esempio:

- $f(n) = 2n^2 + 3, f(n) = \Omega(n)$  in quanto

$$2n^2 + 3 \geq c*n, \forall n \geq n_0, c=1$$

Ma  $f(n)$  è anche in  $\Omega(n^2)$  in quanto

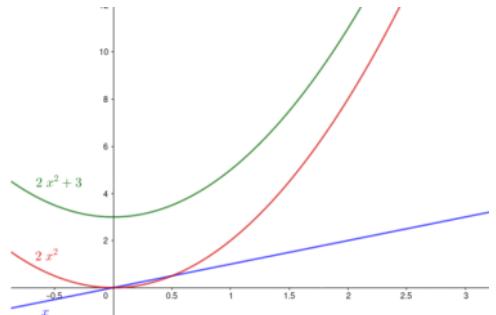
$$2n^2 + 3 \geq c*n^2, \forall n, \text{ se } c \leq 2$$

Similmente a O grande, possiamo formulare il seguente teorema:

Sia  $f(n)$  è un **polinomio** di grado  $m$ , definito come:

$$f(n) = \sum_{i=0}^m a_i n^i = a_0 + a_1 n + a_2 n^2 + \dots + a_m n^m$$

possiamo concludere che  $f(n)$  è in  $\Omega(n^m)$

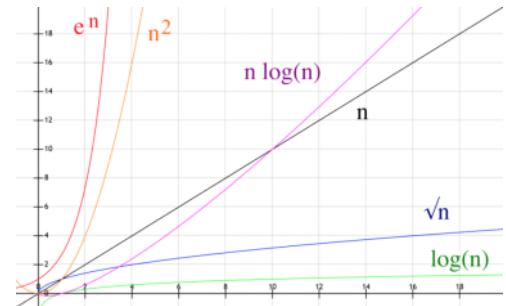


# Ordini grandezza

giovedì 24 ottobre 2024 22:34

## Ordini di grandezza O grande e Omega

- Sia  $f(n) = \log(n)$ . Allora  $f(n)$  è in  $O(\sqrt{n})$  e in  $\Omega(1)$   
Più in generale, abbiamo che:
  - $\log^a(n) = O(n^b)$   $\forall a, b \geq 1$
  - $z^a(n) = \Omega(1) \forall a$
  - Dunque, un poli-logaritmo è dominato da qualunque radice e un poli-logaritmo è dominato da qualunque costante
- Sia  $f(n) = \sqrt[n]{n}$ . Allora  $f(n)$  è in  $O(n)$  e in  $\Omega(\log(n))$   
Più in generale abbiamo che:
  - $\sqrt[n]{n} = O(n^b)$   $\forall a, b \geq 1$
  - $\sqrt[n]{n} = \Omega(\log_b(n)) \forall a, b \geq 1$
  - Dunque, una radice è dominata da qualunque polinomio e una radice domina qualunque poli-logaritmo
- Sia  $f(n) = n^a$ . Allora  $f(n)$  è in  $O(2^n)$  e in  $\Omega(b\sqrt{n})$ 
  - $n^a = O(b^n)$   $\forall a \geq 1, b \geq 2$
  - $n^a = \Omega(b\sqrt{n}) \forall a, b \geq 1$
  - Dunque, un polinomio è dominato da qualunque esponenziale e un polinomio domina qualunque radice



Quindi la **scala di O grandi e Omega** segue la **scala degli ordini di grandezza** delle successioni numeriche (per  $n \rightarrow \infty$ )

$$1 < \log^a(n) < \sqrt[n]{n} (n^{1/b}) < n^c < k^n < n! < n^n (< = precede)$$

# Notazione Teta

giovedì 24 ottobre 2024 18:28

## Notazione Teta

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

se esistono tre costanti  $c_1, c_2$  e  $n_0$  t.c

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n) \quad \forall n \geq n_0$$

Cioè se  $f(n)$  è sia in  $O(g(n))$  e sia in  $\Omega(g(n))$ , allora è anche in  $\Theta(g(n))$ .

Quindi **Teta** rappresenta il **limite stretto asintotico** della funzione:  
una volta superata una certa  $n$ , la funzione  $f(n)$  si comporta come  $g(n)$

Esempio:

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

$f(n)$  è in  $\Theta(n)$  ponendo ad esempio:

$$c_1 = 3, c_2 = 4, n_0 = 3$$

infatti:

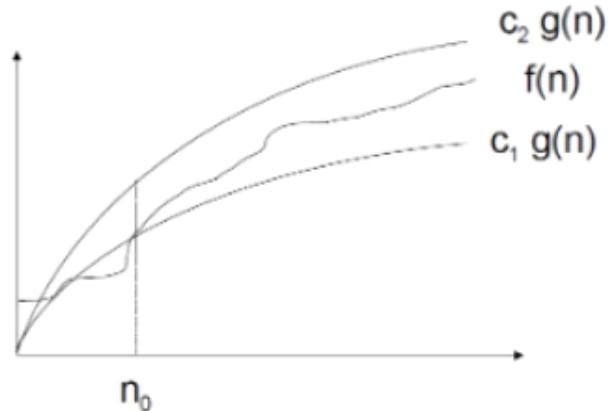
$$3n \leq 3n + 3 \leq 4n \text{ per } n \geq 3$$

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

Basta usare la formula per il cambio di base dei logaritmi:

$$\log_a(n) = \log_b(n) * \log_a(b) = c * \log_b(n)$$

Il cambio di base è quindi asintoticamente irrilevante e nella **notazione asintotica** la base del logaritmo viene spesso **omessa**



# Algebra Asintotica

giovedì 24 ottobre 2024 18:28

## Calcolo delle Notazioni Asintotiche con i Limiti

possiamo formulare le seguenti regole basandoci sulla definizione di limite per  $n \rightarrow \infty$ :

$$\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = k > 0 \text{ allora } f(n) = \Theta(g(n))$$

$$\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = +\infty \text{ allora } f(n) = \Omega(g(n)) \text{ ma } f(n) \neq \Theta(g(n))$$

$$\lim_{n \rightarrow +\infty} \frac{f(n)}{g(n)} = 0 \text{ allora } f(n) = O(g(n)) \text{ ma } f(n) \neq \Theta(g(n))$$

quando il limite non esiste, è necessario procedere diversamente.

## Algebra della notazione Asintotica

Per semplificare il **calcolo del costo computazionale** tramite limite asintotico degli algoritmi, si possono usare **tre regole algebriche**

### Regola delle costanti moltiplicative

- $\forall k > 0$  e  $\forall f(n) \geq 0$ , se  $f(n)$  è in  $O(g(n))$  allora anche  $k*f(n)$  è in  $O(g(n))$
- $\forall k > 0$  e  $\forall f(n) \geq 0$ , se  $f(n)$  è in  $\Omega(g(n))$  allora anche  $k*f(n)$  è in  $\Omega(g(n))$
- $\forall k > 0$  e  $\forall f(n) \geq 0$ , se  $f(n)$  è in  $\Theta(g(n))$  allora anche  $k*f(n)$  è in  $\Theta(g(n))$

In modo informale, si può affermare che **le costanti moltiplicative si possono ignorare**.

ATTENZIONE: è necessario che la costante moltiplicativa **non sia all'esponente** della funzione (es: in  $f(n) = 2^{k*n}$  non possiamo ignorare  $k$ )

### Regola della Commutatività con Somma

Sia  $f(n) = p(n) + q(n)$ .  $\forall p(n), q(n) > 0$  abbiamo che:

- Se  $p(n)$  è in  $O(g(n))$  e  $q(n)$  è in  $O(h(n))$  allora  $f(n)$  è in  $O(g(n) + h(n)) = O(\max(g(n), h(n)))$
- Se  $p(n)$  è in  $\Omega(g(n))$  e  $q(n)$  è in  $\Omega(h(n))$  allora  $f(n)$  è in  $\Omega(g(n) + h(n)) = \Omega(\max(g(n), h(n)))$
- Se  $p(n)$  è in  $\Theta(g(n))$  e  $q(n)$  è in  $\Theta(h(n))$  allora  $f(n)$  è in  $\Theta(g(n) + h(n)) = \Theta(\max(g(n), h(n)))$

Possiamo affermare che dato  $f(n) = p(n) + q(n)$ , allora un qualsiasi limite asintotico di  $f(n)$  è uguale al **massimo tra il limite asintotico di  $p(n)$  e  $q(n)$**

### Regola delle Commutatività con Prodotto

Sia  $f(n) = p(n) * q(n)$ .  $\forall p(n), q(n) > 0$  abbiamo che:

- Se  $p(n)$  è in  $O(g(n))$  e  $q(n)$  è in  $O(h(n))$  allora  $f(n)$  è in  $O(g(n) * h(n))$
- Se  $p(n)$  è in  $\Omega(g(n))$  e  $q(n)$  è in  $\Omega(h(n))$  allora  $f(n)$  è in  $\Omega(g(n) * h(n))$
- Se  $p(n)$  è in  $\Theta(g(n))$  e  $q(n)$  è in  $\Theta(h(n))$  allora  $f(n)$  è in  $\Theta(g(n) * h(n))$

Possiamo affermare che dato  $f(n) = p(n) * q(n)$ , allora un qualsiasi limite asintotico di  $f(n)$  è uguale al **prodotto tra il limite asintotico di  $p(n)$  e  $q(n)$**

# Esercizi applicazione regole

giovedì 24 ottobre 2024 18:29

## Esercizi di applicazione delle regole

- 1) Trova il lim. asint. stretto di  $f(n) = 3n^2 + 7$

$$f(n) = 3n^2 + 7 = \Theta(n^2) + \Theta(1) = \Theta(n^2)$$

- 1) Trova il lim. asint. stretto di  $f(n) = 3n2^n + 4n^4$

$$f(n) = 3n2^n + 4n^4 = \Theta(n) * \Theta(2^n) + \Theta(n^4) = \Theta(n2^n)$$

- 2) Trova il lim. asint. stretto di  $f(n) = 2^{n+1}$

$$f(n) = 2^{n+1} = 2 * 2^n = \Theta(1) * \Theta(2^n) = \Theta(2^n)$$

- 3) Trova il lim. asint. stretto di  $f(n) = 2^{2n}$

$$f(n) = 2^{2n} = 2^n * 2^n = \Theta(2^n) * \Theta(2^n) = \Theta(2^{2n})$$

- 4) Trova il lim. asint. stretto di  $f(n) = 5*2^{\log(n)} + 13$

$$f(n) = 5*2^{\log(n)} + 13 = 5n + 13 = \Theta(n) + \Theta(1) = \Theta(n) \quad (2 \text{ e il log con base } 2 \text{ si annullano quindi } 2^{\log(n)} = n)$$

- 1) Trova il lim. asint. stretto di  $f(n) = \log_2(n) + 8*2^{n*\log_2(n)} + 3$

$$f(n) = \log_2(n) + 8*2^{n*\log_2(n)} + 3 = \log_2(n) + 8*n^n + 3 = \Theta(\log_2(n)) + \Theta(n^n) + \Theta(1) = \Theta(n^n) \quad (\text{stesso motivo di prima, } 2^{n*\log_2(n)} = 2^{\log_2(n)*n} = n^n)$$

- 2) Trova il lim. asint. stretto di  $f(n) = n^2 * \log(n)$

$$f(n) = n^2 * \log(n) = \Theta(n^2) * \Theta(\log(n)) = \Theta(n^2 * \log(n))$$

- 3) Trova il lim. asint. stretto di  $f(n) = 3n*\log(n) + 2n^2$

$$f(n) = 3n*\log(n) + 2n^2 = \Theta(n)*\Theta(\log(n)) + \Theta(n^2) = \Theta(n^2)$$

- 1) Trova il lim. asint. stretto di  $f(n) = 2^{\log(n)/2} + 5n$

$$f(n) = 2^{\log(n)/2} + 5n = (2^{\log(n)})^{1/2} + 5n = \sqrt{n} + 5n = \Theta(\sqrt{n}) + \Theta(n) = \Theta(n)$$

- 2) Trova il lim. asint. stretto di  $f(n) = 4^{\log(n)}$

$$f(n) = 4^{\log(n)} = 2^{\log(n)} * 2^{\log(n)} = n * n = n^2 = \Theta(n^2)$$

oppure

$$f(n) = 4^{\log(n)} = (2^2)^{\log(n)} = (2^{\log(n)})^2 = n^2 = \Theta(n^2)$$

- 3) Trova il lim. asint. stretto di  $f(n) = (\sqrt{2})^{\log(n)}$

$$f(n) = (2^{1/2})^{\log(n)} = (2^{\log(n)})^{1/2} = \sqrt{(2^{\log(n)})} = \sqrt{n} = \Theta(\sqrt{n})$$

# Costo Computazionale

giovedì 24 ottobre 2024 18:33

Il calcolo del **costo computazionale** (tempo di esecuzione di un algoritmo è una **funzione monotona non decrescente**) è strettamente dipendente dalla **quantità di dati in input**, quindi bisogna prima trovare il **parametro** corrispondente ad esso:

- In un **algoritmo di ordinamento** esso sarà il num. di dati da ordinare
- In un **algoritmo che lavora su una matrice** sarà il num. di righe e colonne
- In un **algoritmo che opera su alberi** sarà il num. di nodi dell'albero
- In altri casi l'individuazione del parametro non è facile

E poiché esistono algoritmi che si comportano diversamente con un input piccolo rispetto ad un input maggiore, è opportuno considerare solo input grandi per calcolarne la vera efficienza

## Costo delle istruzioni

Ci sono **tre categorie di istruzioni**:

- **Istruzioni elementari**: istr. con **tempo di esecuzione costante** (non dipendono dalla dimensione dell'input). Es: op. aritmetiche, lettura/scrittura var., valutazione logica. Poiché il loro tempo di esecuzione è costante, hanno un costo di  $\Theta(1)$

```
var = 10          Θ(1)
var += 10 * 10    Θ(1) + Θ(1) + Θ(1) = Θ(1)
print("Il valore di var è: ", var)   Θ(1)
```

- **Blocchi if/else**: hanno un costo pari alla somma tra il costo della verifica della condizione (di solito  $\Theta(1)$ , ma NON sempre) e il max tra i costi complessivi dei blocchi

```
if(a > b):           condizione = Θ(1)
    a += b
    print("Il valore di a+b è", a)  blocco if = Θ(1) + Θ(1) = Θ(1)
else:
    print("Il valore di a è", a)  blocco else = Θ(1)
```

costo totale = condizione + max(Costolf, CostoElse) =  $\Theta(1) + \max(\Theta(1), \Theta(1)) = \Theta(1)$

- **Blocchi iterativi**: hanno un costo pari alla **somma effettiva (non asintotica)** dei **costi di ciascuna iterazione**, compreso il costo di **verifica della condizione**. Se tutte le iterazioni hanno lo stesso costo, il costo dell'iterazione è **costo singola iterazione \* num. iterazioni**.

Inoltre, la condizione viene valutata una volta in più rispetto al num. di iterazioni, perché l'ultima valutazione è quella che fa terminare l'iterazione (ma se il suo costo è costante, possiamo ignorarlo)

```
for i in range(len(A)):
    sum += A[i]          #n iter. + Θ(1) dell'ultima verifica
                           Θ(1)
```

costo totale = NumIter \* CostoIter + Ultimalter =  $n*\Theta(1) + \Theta(1) = \Theta(n)$

Il **costo dell'algoritmo nel suo complesso** è pari alla **somma dei costi delle istr. che lo compongono**. Però un algoritmo potrebbe avere tempi di esecuzione diversi a seconda dell'input (**caso migliore e peggiore in base all'input**)

Quindi si preferisce conoscere il suo comportamento nel **caso peggiore**, ma per essere più precisi utilizziamo la **notazione Teta (Θ)**. Laddove non sia possibile, **approssimeremo** il costo per difetto ( $\Omega$ ) o per eccesso ( $O$  grande).

# Esempi

giovedì 24 ottobre 2024 18:33

## Esempio 1: Calcolo del max di un array disordinato con n valori

```
def Trova_Max(A):
    n = len(A)          Θ(1)
    max = A[1]          Θ(1)
    for i in range(1, n): #n-1 iter. + Θ(1)
        if A[i] > max:   Θ(1)
            max = A[i]   Θ(1)
    return max           Θ(1)
```

**costo totale:**  $T(n) = \Theta(1) + \Theta(1) + [(n-1) * \Theta(1) + \Theta(1) + \Theta(1)] + \Theta(1) + \Theta(1) = \Theta(1) + [\Theta(n - 1) + \Theta(1)] + \Theta(1) = \Theta(1) + \Theta(n) + \Theta(1) = \Theta(n)$   
ricordiamo che  $\Theta(n - 1) = \Theta(n)$  perché facciamo  $\max(n, 1)$

## Esempio 2: Calcolo della somma dei primi n interi

```
def Calcola_Somma(n):
    somma = 0          Θ(1)
    for i in range(1, n+1): #n iter. + Θ(1)
        somma += 1      Θ(1)
    return somma        Θ(1)
```

**costo totale:**  $T(n) = \Theta(1) + [n * \Theta(1) + \Theta(1) + \Theta(1)] + \Theta(1) = \Theta(1) + \Theta(n) + \Theta(1) = \Theta(n)$

Tuttavia, lo stesso problema può essere risolto in modo ben più efficiente:

```
def Calcola_Somma(n):
    somma = n*(n+1)/2   Θ(1)
    return somma         Θ(1)
```

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

$T(n) = \Theta(1) + \Theta(1) = \Theta(1)$ . Decisamente meglio rispetto al  $\Theta(n)$  precedente.

Spesso è possibile **ottimizzare** pezzi di algoritmo che si occupano di **calcolo matematico** con delle **formule dirette** per ridurre notevolmente il costo dell'algoritmo.

## Esempio 3: Valutazione polinomio nel punto x

$$\sum_{k=0}^n a_i x^i$$

ogni  $a_i$  corrisponde ad un elemento di un array dato in input assieme al val. assunto dalla var. x  
Ad esempio, il seguente polinomio verrà espresso nell'array in questa forma

$$x^2 - 4x + 5 = [a_0, a_1, a_2] = [5, -4, 1]$$

```
def Calcola_Polinomio(A, x):
    somma = 0          Θ(1)
    for i in range(len(A)): #n iter. + Θ(1)
        potenza = 1      Θ(1)
        for j in range(i): #i iter. + Θ(1)
            potenza = x*potenza   Θ(1)
        somma = somma + A[i]*potenza   Θ(1)
    return somma        Θ(1)
```

$T(n) = \Theta(1) + \sum_{i=1}^n (\Theta(1) + \Theta(i) + \Theta(1)) + \Theta(1) =$   
 $= \Theta(1) + \sum_{i=1}^n (\Theta(1) + \Theta(i)) =$   
 $= \Theta(1) + \sum_{i=1}^n \Theta(1) + \sum_{i=1}^n \Theta(i) = \Theta(1) + \Theta(n) + \Theta(n^2)$   
Le **sommatorie**, nell'ambito della notazione asintotica, sono **intercambiabili con la notazione usata**:

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

Tuttavia, invece che ricalcolare la potenza corrispondente ad ogni termine del polinomio, possiamo **conservare** la potenza calcolata per il termine precedente, riducendo la necessità di dover utilizzare il secondo ciclo for:

```
def Calcola_Polinomio(A, x):
    somma = 0          Θ(1)
    potenza = 1        Θ(1)
    for i in range(len(A)): #n iter. + Θ(1)
        potenza = x*potenza   Θ(1)
        somma = somma + A[i]*potenza   Θ(1)
    return somma        Θ(1)
```

$T(n) = \Theta(1) + n*\Theta(1) + \Theta(1) = \Theta(n)$ . Decisamente meglio rispetto al  $\Theta(n^2)$  precedente

#### Esempio 4: Analisi del caso migliore e peggiore

```
def es4(n):
    if n<0:
        n = -n
    while(n):
        if n%2 == 1:
            return 1
        n -= 2
    return 0
```

##### Analisi del ciclo while:

- Se  $n$  è **dispari**, allora la condizione  $if n \% 2 == 1$  restituirà **True**, eseguendo l'istr. **return** e terminando il ciclo  
Dunque abbiamo un costo pari a  $\Theta(1)$
- Se  $n$  è **pari**, allora il comportamento del ciclo sarà

n. iter	1	2	3	4	...	k
val. di n	$n - 2$	$n - 4$	$n - 6$	$n - 8$	...	$n - 2k$

Finché  $n - 2k = 0$  (condizione necessaria a terminare il while)

Dunque il costo sarà  $\Theta(n)$

$$\begin{aligned} n - 2k &= 0 \\ n &= 2k \\ k &= n/2 \\ 1/2 * \Theta(n) &= \Theta(n) \end{aligned}$$

Possiamo quindi dire che

$$\begin{aligned} \Theta(1) &\text{ se } n \text{ è dispari (caso migliore)} \\ T(n) = \Theta(1) &\text{ se } n \text{ è pari (caso peggiore)} \end{aligned}$$

#### Esempio 5: Iterazione con radice

esempio in cui il num. di iter. del ciclo descritto corrisponde ad una radice:

```
def es5(n):
    n = abs(n)
    x = r = 0
    while x*x < n:
        x += 1
        r *= 3*x
    return r
```

##### Analisi del ciclo while:

- Comportamento del ciclo:

n. iter	1	2	3	4	...	k
val. di x	1	2	3	4	...	k

Finchè  $x*x = n$

Dunque, il num. di iter sarà

$$\begin{aligned} x * x &= n \\ x^2 &= n \\ x &= \sqrt{n} \end{aligned}$$

Il costo finale è:

$$T(n) = \Theta(1) + \sqrt{n} * \Theta(1) + \Theta(1) = \Theta(\sqrt{n})$$

#### Esempio 6: Iterazioni logaritmiche

```
def es6(n):
    n = abs(n)
    x = r = 0
    while n > 1:
        r += 2
        n = n//3
    return r
```

##### Analisi del ciclo while:

- Comportamento del ciclo:

n. iter	1	2	3	4	...	k
val. di n	$n / 3$	$n / 3^2$	$n / 3^3$	$n / 3^4$	...	$n / 3^k$

Finchè  $n / 3^k = 1$

Dunque, il num di iter sarà

$$\begin{aligned} n / 3^k &= 1 \\ n &= 3^k \\ k &= \log_3(n) \end{aligned}$$

Il costo finale è:

$$T(n) = \Theta(1) + \log_3(n) * \Theta(1) + \Theta(1) = \Theta(\log(n))$$

### Esempio 7: Iterazioni esponenziali

```
def es7(n):
    n = abs(n)
    x = t = 1
    for i in range(n):
        t = 3*t
    t -= 1
    while t ≥ x:
        x += 2
        t -= 2
    return x
```

Analisi del ciclo for:

- Il ciclo viene eseguito  $n$  volte, ad ogni iter. la var.  $t$  viene moltiplicata per 3. Alla fine del ciclo avremo  $t = 3^n$

Prima del ciclo  $t = 3^n - 1$

Analisi del ciclo while:

- Comportamento del ciclo:

n. iter	1	2	3	4	...	k
val. di x	3	5	7	9	...	$2k + 1$
val. di t	$3^n - 3$	$3^n - 5$	$3^n - 7$	$3^n - 9$	...	$3^n - (2k + 1)$

$$\text{Finch\`e } t = x \Rightarrow 3^n - (2k + 1) = 2k + 1$$

$$3^n - (2k + 1) = 2k + 1$$

$$3^n - 2k - 1 = 2k + 1$$

$$4k = 3^n - 2$$

$$k = (3^n - 2) / 4$$

Il costo finale è:

$$T(n) = \Theta(1) + n * \Theta(1) + ((3^n - 2) / 4) * \Theta(1) + \Theta(1) = \Theta(n) + \Theta(3^n) = \Theta(3^n)$$

### Esempio 8: Doppio logaritmo

```
def es8(n):
    n = abs(n)
    p = 2
    while n ≥ p:
        p = p*p
    return p
```

Analisi del ciclo while:

- Comportamento del ciclo:

n. iter	1	2	3	4	...	k
val. di p	$2^2$	$2^4$	$2^6$	$2^8$	...	$2^{2^k}$

$$\text{Finch\`e } 2^{2^k} = n + 1$$

$$2^{2^k} = n + 1$$

$$2^k = \log_2(n + 1)$$

$$k = \log_2(\log_2(n + 1))$$

Il costo finale è:

$$T(n) = \Theta(1) + \log_2(\log_2(n + 1)) * \Theta(1) + \Theta(1) = \Theta(\log_2(\log_2(n + 1)))$$

### Esempio 9: ciclo annidato

```
def es9(n):
    n = abs(n)
    i, j, t, s = 1
    while i*i ≤ n:
        for j in range(t):
            s += 1
        i += 1
        t += 1
    return s
```

Analisi del ciclo while:

- Comportamento del ciclo:

n. iter	1	2	3	4	...	k
val. di i	2	3	4	5	...	$k + 1$
val. di t	2	3	4	5	...	$k + 1$

Finchè  $i^*i = n + 1 \Rightarrow (k + 1)^2 = n + 1$

$$(k + 1)^2 = n + 1$$

$$k + 1 = \sqrt{n + 1}$$

$$k = \sqrt{n + 1} - 1$$

Analisi del ciclo for annidato:

- Ad ogni iter. del ciclo while, il ciclo for viene eseguito  $t$  volte. Tuttavia il valore di  $t$  aumenta di 1 ad ogni iter. del while, quindi il num. di iter sarà:

$$\sum_{t=1}^{\sqrt{n+1}-1} t = \frac{(\sqrt{n+1} - 1) \cdot \sqrt{n+1}}{2} = \frac{n+1 - \sqrt{n+1}}{2}$$

Il costo finale è:

$$T(n) = \Theta(1) + (n+1 - \sqrt{n+1})/2 * \Theta(1) + \Theta(1) = \Theta(n)$$

### Esempio 10

```
def es10:  
    n = abs(n)  
    s = n  
    p = 2  
    i, r = 1  
    while s ≥ 1:  
        s = s//5  
        p += 2  
    p = p*p  
    while i*i*i < n:  
        for j in range(p):  
            r += 1  
        i += 1  
    return r
```

Analisi del primo while:

- Comportamento del ciclo

n. iter	1	2	3	4	...	k
val. di s	$n/5$	$n/5^2$	$n/5^3$	$n/5^4$	...	$n/5^k$
val. di p	4	6	8	10	...	$2 + 2k$

Finchè  $n/5^k = 1$

$$n/5^k = 1$$

$$n = 5^k$$

$$k = \log_5(n)$$

Poiché il ciclo è eseguito  $\log_5(n)$  volte, anche  $p += 2$  è eseguito  $\log_5(n)$  volte. Quindi alla fine del ciclo  $p = 2 + 2 * \log_5(n)$

Poi viene eseguito  $p = p*p$  quindi  $p = (2 + 2 * \log_5(n))^2$

Analisi del secondo while:

- Comportamento del ciclo

n. iter	1	2	3	4	...	k
val. di i	1	2	3	4	...	k
val. di $i^3$	$2^3$	$3^3$	$4^3$	$5^3$	...	$(k+1)^3$

Finchè  $(k+1)^3 = n$

$$(k+1)^3 = n$$

$$k+1 = \sqrt[3]{n}$$

$$k = \sqrt[3]{n} - 1$$

Analisi ciclo for annidato:

- Il ciclo vengono eseguite  $p$  iter.  $(2 + 2 * \log_5(n))^2$  volte) ad ogni iterazione del ciclo while ( $\sqrt[3]{n} - 1$  volte)

Il costo finale è:

$$\begin{aligned} T(n) &= \Theta(1) + \log_5(n)*\Theta(1) + (\sqrt[3]{n})*((2 + 2 * \log_5(n))^2 * \Theta(1) + \Theta(1)) + \Theta(1) = \\ &= \Theta(\log(n)) + (\sqrt[3]{n})*(\Theta(\log^2(n)) + \Theta(1) + \Theta(1) = \Theta(\log(n)) + \Theta(\sqrt[3]{n} * \log^2(n)) + \Theta(\sqrt[3]{n})) = \\ &= \Theta(\sqrt[3]{n} * \log^2(n)) \end{aligned}$$

# Ricerca Sequenziale

giovedì 24 ottobre 2024 18:33

Uno dei problemi più ricorrenti in informatica è la **ricerca di un elemento in un insieme di dati** (es. num, cognomi, ecc)

Tali problemi consistono in:

- **Input:** un **array** A di  $n$  elementi ed un **valore** v da cercare al suo interno
- **Output:** l'**indice** corrispondente alla posizione dell'elemento v trovato all'interno dell'array, oppure **Null** o -1 se l'elemento non viene trovato

## Ricerca sequenziale

La prima tipologia di **algoritmo di ricerca** è composta da tre passaggi:

- Presi in input il val. v da cercare e l'array, quest'ultimo viene analizzato sequenzialmente **elemento per elemento**
- Ogni elemento viene **confrontato** con v. Se **coincide** con v allora viene restituito l'**indice** dell'elemento, altrimenti si procede con il **prossimo elemento**
- Se sono finiti gli elementi da controllare allora v non è nell'array e viene restituito -1

anche senza dover analizzare il codice possiamo vedere che il **caso migliore** corrisponde al caso in cui l'elemento da cercare sia in **prima posizione** (indice 0), mentre il **caso peggiore** corrisponde al caso in cui l'elemento **non sia presente** nella lista (dovrà comunque essere analizzata tutta la lista)

Dunque:

- **Caso migliore:** Se  $v = A[0]$ , allora  $\Theta(1)$
- **Caso peggiore:** Se  $v \notin A$ , allora  $\Theta(n)$

Non avendo trovato una stima del costo che sia valida per tutti i casi, diremo che il costo computazionale dell'algoritmo è un  **$O(n)$** , per evidenziare che ci sono input in cui questo caso viene raggiunto, ma altri casi in cui il costo è minore.

```
def Ricerca_Sequenziale(A, v):
    for i in range(len(A)):
        if A[i] == v:
            return i
    return -1
```

Poiché il caso migliore e peggiore sono differenti, non è possibile determinare il costo asintotico stretto di tale algoritmo.  
è necessario effettuare una stima del **costo medio** dell'algoritmo, ossia quello che si verifica con **più probabilità**.

Ipotizziamo di avere un array in cui ogni posizione ha la **stessa probabilità** di contenere il val. v da cercare.

Quindi la **probabilità che v sia in i-esima posizione** è

$$P = \frac{1}{n}$$

Applicando la probabilità al **num. totale di iter.** otteniamo:

$$P * \sum_{i=1}^n i = \frac{1}{n} * \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Dunque **in media** il ciclo viene eseguito  $(n+1)/2$  volte, quindi  $\Theta(n)$ . In questo caso, il **caso medio** si avvicina di più al **caso peggiore** rispetto a quello migliore

In alternativa, il **costo medio** può essere trovato utilizzando il calcolo delle **permutazioni** (possibili combinazioni di n elem.).

Le permutazioni di una lista di  $n$  elementi corrisponde a  $n!$ , quindi le **permutazioni totali di A** sono  $P_{tot} = n!$

All'interno di questo insieme di permutazioni, vi sono anche i seguenti sotto-insiemi:

- Permutazioni in cui v è in **prima posizione**
- Permutazioni in cui v è in **seconda posizione**
- Permutazioni in cui v è in **terza posizione**
- ...
- Permutazioni in cui v è in **ultima posizione**

Ognuno di tali sotto-insiemi corrisponde ad una **permutazione di n-1 elementi**, ossia  $P_i = (n-1)!$ . Dunque il num. medio di iterazioni del ciclo è:

$$\sum_{i=1}^n i * \frac{n \cdot \text{permutazioni in cui } v \text{ è in pos } i}{n \cdot \text{tot di permutazioni}} = \sum_{i=1}^n (i * \frac{P_i}{P_{tot}}) = \sum_{i=1}^n (i * \frac{(n-1)!}{n!}) = \sum_{i=1}^n (i * \frac{1}{n}) = \frac{1}{n} * \sum_{i=1}^n i = \frac{1}{n} * \frac{n(n+1)}{2} = \frac{n+1}{2}$$

Anche in questo caso il **numero medio di iterazioni del ciclo** è  $(n+1)/2$  quindi  $\Theta(n)$

ATTENZIONE:

l'operatore **in** di Python usa la **ricerca sequenziale** quindi ha costo  $O(n)$ . es. "if v in A" sembra con costo pari a  $\Theta(1)$  ma in realtà è  $\Theta(n)$

# Ricerca Binaria

giovedì 24 ottobre 2024

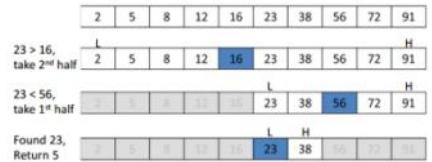
18:35

L'algoritmo della **ricerca binaria** funziona come segue:

- 1) Viene ispezionato l'elemento centrale dell'array (che chiameremo  $m$ ):
  - o Se  $v = m$ , allora abbiamo trovato l'elemento che stavamo cercando e viene restituito l'indice della posizione trovata
  - o Se  $v < m$ , allora  $v$  sarà nella **metà inferiore della lista**, dunque l'algoritmo verrà ripetuto su di essa
  - o Se  $v > m$ , allora  $v$  sarà nella **metà superiore della lista**, e l'algoritmo verrà ripetuto su di essa
- 2) **Ripetiamo** il primo passaggio finché la lista non sarà ridotta ad un **singolo elemento**
  - o Se l'**unico elemento rimasto** dopo la riduzione della lista effettuate è  $v$ , allora verrà restituito l'indice trovato
  - o Altrimenti, verrà restituito -1 poiché  $v$  non è nella lista

L'unico **requisito necessario** per poter applicare questo algoritmo è che **l'array deve essere obbligatoriamente ordinato in modo crescente.**

```
def Ricerca_Binaria(A, v):  
    a = 0 # il primo indice di A  
    b = len(A) - 1 # l'ultimo indice di A  
    m = (a+b)//2 # l'indice a metà di A  
    while A[m] != v: # finchè non trovo v  
        if A[m] > v: # se v è minore di A[m]  
            b = m - 1 # prendo la metà inferiore  
        else: # se v è maggiore di A[m]  
            a = m + 1 # prendo la metà superiore  
        if a > b: # se v non è in A  
            return -1  
        m = (a+b)//2 # ricalcolo il nuovo valore di m  
    return m
```



Un **esempio grafico** dell'algoritmo in cui viene ricercato il valore 23

Analisi del ciclo while:

- **Ipotesi:** supponiamo che l'el.  $v$  venga trovato alla  $k$ -esima iter. del ciclo
- **Comportamento del ciclo:**

n. iter	1	2	3	4	...	$k$
Len(A)	$n/2$	$n/2^2$	$n/2^3$	$n/2^4$	...	$n/2^k$

Finchè  $n/2^k = 1$  (nel caso peggiore solo  $v$  rimane nell'array)

$$\begin{aligned} n/2^k &= 1 \\ n &= 2^k \\ k &= \log_2(n) \end{aligned}$$

Costo finale nel caso peggiore:

$$T(n)_{\text{peggiore}} = \Theta(1) + \log_2(n) * \Theta(1) + \Theta(1) = \Theta(\log_2(n))$$

Il costo della ricerca binaria ( $\Theta(\log_2(n))$ ) è nettamente meglio di quello della ricerca sequenziale ( $\Theta(n)$ ).

Nel **caso migliore** corrisponde al caso in cui, appena avviato l'algoritmo, si verifica che  $v = A[m]$  ( $v$  è in mezzo all'array) quindi abbiamo  $\Theta(1)$

Come per la ricerca sequenziale, il caso peggiore e migliore discordano, quindi dobbiamo calcolare il **caso medio**

**Assunzioni:**

- Il num. di elementi dell'array è una **potenza di 2** (per semplicità di calcolo)
- $v$  è **presente nell'array** (dunque non si ricade nel caso peggiore)
- Tutte le posizioni dell'array hanno la **stessa probabilità** di contenere  $v$  ( $p = \frac{1}{n}$ )

**Analisi delle posizioni raggiungibili**

- 1) Alla 1° iter., le pos. raggiungibili sono solo **una**, quella centrale
- 2) Alla 2° iter. le pos. raggiungibili sono **due**:
  - a. Quella al centro della **metà inferiore**
  - b. Quella al centro della **metà superiore**
- 3) Alla 3° iter. le pos. raggiungibili sono **quattro**:
  - a. Quella al centro della metà inferiore della **prima metà inferiore**
  - b. Quella al centro della metà superiore della **prima metà inferiore**
  - c. Quella al centro della metà inferiore della **prima metà superiore**
  - d. Quella al centro della metà superiore della **prima metà superiore**
- 4) E così via

Concludiamo che le **posizioni raggiungibili** da ogni  $k$ -esima iter. sono  $n(k) = 2^{k-1}$

Poiché abbiamo assunto che ogni posizione è **equiprobabile** la probabilità di ogni pos è  $\frac{1}{n} * n(k) = \frac{2^{k-1}}{n}$

Il numero medio di iterazioni sarà

$$\sum_{k=1}^{\log(n)} (k * \frac{2^{k-1}}{n}) = \frac{1}{n} * \sum_{k=1}^{\log(n)} k * 2^{k-1}$$

Ricordando che  $\sum_{i=1}^n i * 2^{i-1} = (n - 1) * 2^n + 1$ , abbiamo:

$$\frac{(\log(n) - 1) * 2^{\log(n)} + 1}{n} = \frac{(\log(n) - 1) * n}{n} + \frac{1}{n} = \log(n) - 1 + \frac{1}{n} = \Theta(\log(n))$$

# Ricorsione

giovedì 24 ottobre 2024 18:35

In matematica una funzione è detta **ricorsiva** quando la sua definizione è espressa in termini di se stessa (es. il **fattoriale**)  
Nel campo degli algoritmi vi è un concetto del tutto analogo, quello degli **algoritmi ricorsivi**:

$$n! = \begin{cases} n \cdot (n-1)! & \text{se } n > 0 \\ 1 & \text{se } n = 0 \end{cases}$$

Un algoritmo è detto ricorsivo quando è espresso in **termini di se stesso**.

Un algoritmo ricorsivo ha sempre queste proprietà:

- la **soluzione** del problema complessivo è **costruita risolvendo (ricorsivamente) uno o più sottoproblemi** di dimensione minore e **combinando poi queste soluzioni**
- La successione dei **sottoproblemi**, che sono sempre più piccoli, **dove sempre convergere ad un sottoproblema** che costituisca un **caso base** la cui **ricorsione termina**

Esempio ricerca binaria (dettagli volutamente tralasciati):

```
def Ricerca_Binaria_seq(A, v):  
    if len(A) == 0: # se A è vuoto  
        return -1  
    m = len(A)//2 # calcolo l'indice a metà di A  
    if A[m] == v: # se v sta a metà  
        return m  
    if v < A[m]: # se v si trova nella metà inferiore  
        return Ricerca_Binaria_seq(A[:m], v)  
    else: # se v si trova nella metà superiore  
        return Ricerca_Binaria_seq(A[m+1:], v)
```

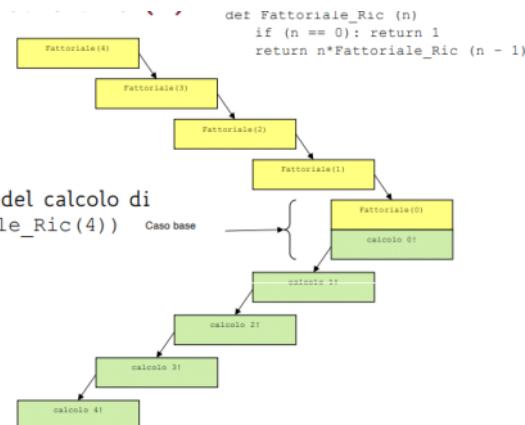
Altri esempi di codice ricorsivo

- **Fattoriale di n**  

```
def fattoriale(n): # n = int. non negativo  
    if n==0: return 1  
    return n*fattoriale(n - 1)
```
- **Ricerca sequenziale**  

```
def ricerca_seq(A, v, n=len(A)-1):  
    if A[n] == v:  
        return n  
    if n == 0:  
        return -1  
    else:  
        return ricerca_seq(A, v, n - 1)
```
- **Ricerca binaria**  

```
def ricerca_bin(A, v, i_min=0, i_max=len(A))  
    if i_min > i_max:  
        return -1  
    m = (i_min + i_max)//2  
    if A[m] == v:  
        return m  
    elif A[m] > v:  
        return ricerca_bin(A, v, i_min, m-1)  
    else:  
        return ricerca_bin(A, v, m+1, i_max)
```



(sviluppo del calcolo di  
Fattoriale\_Ric(4))

Caso base

Anche nella formula ricorsiva della ricerca binaria ogni nuova chiamata ricorsiva riceve un sotto-problema la cui dimensione è circa la metà di quello originario quindi, come nella versione iterativa, si arriva rapidamente al caso base con costo computazionale di **O(log(n))**

## Iterazione vs Ricorsione

Qualsiasi problema risolvibile in modalità ricorsiva può essere risolto anche in modalità iterativa

Conviene usare allora:

Algoritmo ricorsivo:	Algoritmo iterativo
Quando si può formulare la soluzione è inerentemente ricorsiva (es. fattoriale), mentre la soluzione iterativa è più complicata o non evidente	<ul style="list-style-type: none"><li>• Esiste una soluzione iterativa semplice e chiara rispetto alla versione ricorsiva</li><li>• L'efficienza è un requisito primario</li></ul>

Infatti in genere le funzioni ricorsive richiamano se stesse un numero elevato di volte richiedendo **maggiori esigenze in termini di memoria**.

Si preferisce allora l'iterazione, a meno che il problema non sia intuitivamente risolvibile come algoritmo ricorsivo.

Es:

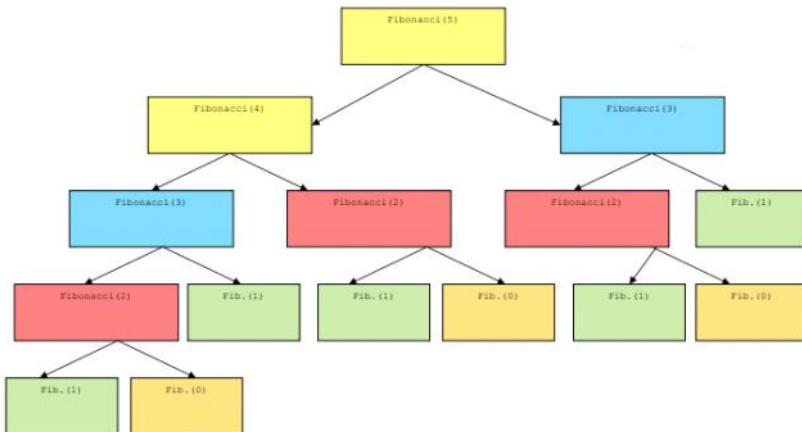
- Calcolo dell'n-esimo numero di Fibonacci, definito come:
  - $F(0) = 0$
  - $F(1) = 1$
  - $F(n) = F(n-1) + F(n-2)$  se  $n > 1$

In questo caso viene naturale pensare all'algoritmo ricorsivo poiché l'n-esimo numero viene calcolato tramite altri due numeri di Fibonacci.

Il codice ricorsivo sarebbe:

```
def Fibonacci_ric(n):  
if n ≤ 1:          θ(1)  
    return n          θ(1)  
return (Fibonacci_ric(n-1) + Fibonacci_ric(n+2))   T(n-1) + T(n-2)
```

Poiché ci sono due chiamate ricorsive e non una, l'articolazione delle chiamate di funzione diventa più complessa:



Il suo costo pari a

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

Per calcolare il costo effettivo è necessario introdurre il concetto di **equazione di ricorrenza**

Invece la sua versione iterativa è:

```
def Fibonacci_iter(n):
    if n ≤ 1: return n
    fib_prec_prec = 0
    fib_prec = 1
    for i in range(2, n+1):
        fib_prec_prec, fib_prec = fib_prec, fib_prec+fib_prec_prec
    return fib_prec
```

Il suo costo è pari a  $\Theta(n)$

#### Osservazioni:

- Molti calcoli vengono **ripetuti** più volte
- L'occupazione di memoria è **molto alta**:
  - Il numero totale delle chiamate effettuate cresce molto velocemente con  $n$
  - Quando si arriva a ciascun caso base vi è una catena di chiamate "aperte" lungo il percorso per tornare alla chiamata iniziale
- Per risolvere un problema di dim.  $n$  si devono risolvere due sottoproblemi di dim. ben poco inferiori ( $n-1$  e  $n-2$ )

## Esempi

giovedì 24 ottobre 2024 18:35

### Esercizio 1 - potenza k-esima di n

Dati in input n e k interi, calcolare la potenza k-esima di n  
Possiamo basarci sul fatto che:  $n^k = n * n^{k-1}$  e  $n^0 = 1$

```
def n_alla_k(n, k):
    if k == 0: return 1
    return n*n_alla_k(n, k-1)
```

Costo:

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

n è un parametro inutile per il costo computazionale

### Esercizio 2 - somma elementi array

Dato in input un array di n interi, calcolare la somma dei suoi elementi  
La somma è pari al val. di un elemento più la somma ricorsivamente sul resto dell'array

```
def SommaArray(A, ult_ind=len(A)-1):
    if ult_ind == 0: return A[ult_ind]
    return (A[ult_ind] + SommaArray(A, ult_ind-1))
```

Costo:

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

### Esercizio 3 - minimo array

Dato in input un array di n interi, trovare il minimo  
Il minimo è pari al minore fra il val. di un elemento e il min trovato ricorsivamente sul resto dell'array

```
def MinArray(A, ult_ind=len(A)-1):
    if ult_ind == 0:
        return A[ult_ind]
    return min(A[ult_ind], MinArray(A, ult_ind-1))
```

Costo:

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

### Esercizio 4 - palindromo

Dato in input un array di n interi, verificare se è palindromo  
Un array è palindromo se i suoi estremi sono uguali e il resto dell'array è anch'esso palindromo  
!!! se n è dispari c'è un elem. centrale, altrimenti no; questa differenza va gestita

```
def Palindromo(A, ind_min=0, ind_max=len(A)-1):
    if ind_min ≥ ind_max: # caso base
        return True
    if A[ind_min] != A[ind_max]:
        return False
    return Palindromo(A, ind_min+1, ind_max-1)
```

!!! dim dell'input n = len(A) = ind\_max - ind\_min + 1

Costo:

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

Esempio di array palindromo:

1	5	4	8	4	5	1
---	---	---	---	---	---	---

### Esercizio 5 - stampa array ordine inverso

Dato in input un array di n interi, stampare le chiavi dall'ultima alla prima, ossia nell'ordine:  
A[n-1], A[n-2], A[n-3], ..., A[2], A[1], A[0]

Stampiamo l'ultima chiave e chiamiamo ricorsivamente la funzione sul resto dell'array, gestendo opportunamente un secondo parametro.

```
def StampaInv(A, ind_ult=len(A)-1):
    print(A[ind_ult])
    if ind_ult > 0:
        StampaInv(A, ind_ult-1)
```

Costo:

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

### Esercizio 6 - stampa array ordine normale

Se vogliamo stampare le chiavi dalla prima all'ultima?

Basta stampare durante la risalita dalle chiamate ricorsive, dopo aver incontrato il caso base per  $n=1$ .

Basta scambiare l'ordine della stampa rispetto alla chiamata ricorsiva

```
def StampaArr(A, ui=len(A)-1):
    if ui > 0:
        StampaArr(A, ui-1)
    print(A[ui])
```

Costo:

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

# Metodo Iterativo

giovedì 24 ottobre 2024 18:37

La funzione matematica ricorsiva che esprime il costo è anche detta **equazione di ricorrenza**.

Riprendiamo l'esempio del calcolo fattoriale

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

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

La parte generale dell'equazione di ricorrenza che definisce  $T(n)$  deve essere sempre costituita dalla **somma di almeno due addendi**, di cui **almeno uno contiene la parte ricorsiva** (nell'esempio  $T(n-1)$ ), mentre uno rappresenta il costo computazionale di tutto ciò che viene eseguito al di fuori della chiamata ricorsiva (in questo caso il  $\Theta(1)$ ).

Inoltre anche nell'equazione di ricorrenza **dove sempre essere presente un caso base** (in questo caso  $T(0)$ )

## Metodo Iterativo

L'idea del **metodo iterativo** è di sviluppare l'equazione di ricorrenza ed esprimere come **somma di termini dipendenti da n e dal caso base**

**Difficoltà:** Maggiore quantità di calcoli algebrici rispetto agli altri metodi

**Non applicabile** quando l'espansione della ricorrenza **non porta a un pattern chiaro** o una **formula chiusa facilmente identificabile**.

**Problemi tipici non applicabili:**

- Eq. con termini non facilmente esprimibili in funzione di  $n$  (es. dipendenze complesse da più val. precedenti)
- Ricorrenze **non omogenee** con termini variabili (es.  $T(n) = T(n-1) + T(n-2) + n$ )
- Se il num. di iter. richiesto per arrivare alla base è difficile da determinare

Es:

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

Sviluppiamo  $T(n)$  come **somma dei suoi sotto-termeni**:

Se  $T(n)$  è definito come  $T(n-1)+\Theta(1)$ , allora  $T(n-1)$  sarà definito come  $T(n-2)*\Theta(1)$  e così via

1. Per definizione abbiamo

$$T(n) = T(n-1) + \underbrace{\Theta(1)}_{\text{Una volta}} = T(n-2) + 1 \cdot \Theta(1)$$

2. Sviluppando  $T(n-1)$ , otteniamo che

$$T(n) = T(n-2) + \underbrace{\Theta(1) + \Theta(1)}_{\text{Due volte}} = T(n-2) + 2 \cdot \Theta(1)$$

3. Sviluppando  $T(n-2)$ , otteniamo che

$$T(n) = T(n-3) + \underbrace{\Theta(1) + \Theta(1) + \Theta(1)}_{\text{Tre volte}} = T(n-3) + 3 \cdot \Theta(1)$$

4. Sviluppando  $T(n-(k-1))$ , otteniamo che

$$T(n) = T(n-k) + \underbrace{\Theta(1) + \Theta(1) + \dots + \Theta(1) + \Theta(1)}_{k \text{ volte}} = T(n-k) + k \cdot \Theta(1)$$

Abbiamo ottenuto una **forma generalizzata** del caso  $T(n)$  sviluppando i suoi sotto-termeni  $k$ -volte.

Dopo un **determinato num. di ricorsioni**, l'equazione raggiungerà il suo **caso base** ( $T(1) = \Theta(1)$ )

La catena continuerà quindi finche  $n - k = 1$  così che  $T(n-k) = T(1)$ , Dunque:

$$n - k = 1 \Rightarrow k = n - 1$$

Sostituendo il val. di  $k$  nell'equazione per trovare il **costo computazionale generico** avremo

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

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

Ulteriori esempi (preso uno dall'exsys e uno dagli esercizi del prof):

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

Sviluppando  $T(n)$  otteniamo che

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1) = T\left(\frac{n}{2^2}\right) + \Theta(1) + \Theta(1) = T\left(\frac{n}{2^3}\right) + \Theta(1) + \Theta(1) + \Theta(1) = \dots$$

Generalizzando l'equazione, otteniamo

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

Sappiamo che il caso base è  $T(1)$  e che viene raggiunto quando

$$\frac{n}{2^k} = 1 \implies k = \log_2(n)$$

Dunque ne segue che

$$T(n) = T\left(\frac{n}{2^k}\right) + k \cdot \Theta(1) = T(1) + \log_2(n) \cdot \Theta(1) = \Theta(1) + \Theta(\log(n)) = \Theta(\log(n))$$

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + \Theta(n^2) \\ &= T\left(\frac{n}{2^2}\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right) + \Theta(n^2) \\ &= T\left(\frac{n}{2^3}\right) + \Theta\left(\left(\frac{n}{2^2}\right)^2\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right) + \Theta(n^2) \\ &\dots \\ &= T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} \Theta\left(\left(\frac{n}{2^i}\right)^2\right) \end{aligned}$$

Continuiamo ad iterare fino al raggiungimento del caso base, cioè fino a quando  $n/2^k = 1$ , il che avviene se e solo se  $k = \log_2(n)$ .

L'equazione diventa così:

$$T(n) = T(1) + \sum_{i=0}^{\log_2(n)-1} \Theta\left(\left(\frac{n}{2^i}\right)^2\right) = \Theta(1) + n^2 \sum_{i=0}^{\log_2(n)-1} \Theta\left(\frac{1}{2^i}\right).$$

Ricordando che  $\sum_{i=0}^{\log_2(n)-1} 2^i = \frac{2^{\log_2(n)}}{2} - 1$  otteniamo infine,

$$T(n) = T\left(\frac{n}{2^k}\right) + k \cdot \Theta(1) = T(1) + \log_2(n) \cdot \Theta(1) = \Theta(1) + \Theta(\log(n)) = \Theta(\log(n))$$

$$T(n) = T(1) + \sum_{i=0}^{\log n - 1} \Theta\left(\left(\frac{n}{2^i}\right)^2\right) = \Theta(1) + n^2 \sum_{i=0}^{\log n - 1} \Theta\left(\frac{1}{4^i}\right).$$

Esistono dei casi particolari, vediamo ad esempio il calcolo dell' $n$ -esimo numero di Fibonacci

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

- Sviluppando  $T(n)$  otteniamo che:

$$T(n) = T(n-1) + T(n-2) + \Theta(1) = T(n-2) + 2T(n-3) + T(n-4) + 3 * \Theta(1) = T(n-3) + 3 * T(n-4) + 3 * T(n-5) + T(n-6) + 6 * \Theta(1) = \dots$$

In un caso del genere è complicato **generalizzare il problema**, in una situazione del genere possiamo usare le **maggiorazioni** e le **minorazioni**, cercando di calcolare il **costo O grande** e il **costo Ω**:

Calcolo O grande:

- poiché  $T(n) = T(n-1) + T(n-2) + \Theta(1)$ , possiamo dire che:  
 $T(n-1) + T(n-2) + \Theta(1) \leq T_1(n) = T(n-1) + T(n-1) + \Theta(1)$

Dunque, poiché l'equazione originale è **minore** a questa nuova equazione di ricorrenza, calcolando il **costo Θ della nuova equazione** otteniamo anche il **costo O grande dell'equazione iniziale**:

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

- Sviluppando  $T_1(n)$  otteniamo

$$T_1(n) = 2T_1(n-1) + \Theta(1) = 2[2T_1(n-2) + \Theta(1)] + \Theta(1) = 2[2[2T_1(n-3) + \Theta(1)] + \Theta(1)] + \Theta(1) = \dots$$

- Generalizzando l'equazione a

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

Sappiamo che il caso base è  $T(1)$  e che viene raggiunto quando

$$n - k = 1 \Rightarrow k = n - 1$$

Dunque

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

Quindi, poiché  $T(n) \leq T_1(n)$ , ne segue che

$$T(n) \leq T_1(n)$$

$$T(n) \leq \Theta(2^n)$$

$$T(n) = O(2^n)$$

Calcolo Ω:

- poiché  $T(n) = T(n-1) + T(n-2) + \Theta(1)$ , possiamo dire che:  
 $T(n-1) + T(n-2) + \Theta(1) \geq T(n) = T(n-2) + T(n-2) + \Theta(1)$

Rispetto a prima, l'equazione originale è **maggior** rispetto a questa nuova equazione, quindi calcolando il **costo Θ della nuova equazione** otteniamo anche il **costo Ω dell'equazione iniziale**:

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

- Sviluppando  $T_2(n)$  otteniamo che:

$$T_2(n) = 2T_2(n-2) + \Theta(1) = 2[2T_2(n-4) + \Theta(1)] + \Theta(1) = 2[2[2T_2(n-6) + \Theta(1)] + \Theta(1)] + \Theta(1) = \dots$$

- Generalizzando l'equazione a

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

Il procedimento si ferma quando  $n - 2k = 0$  (se  $n$  è pari; se è dispari il procedimento differisce per alcuni dettagli ma il comportamento asintotico non cambia), ossia quando  $k = n/2$

Dunque ne segue che

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

Quindi, poiché  $T(n) \geq T_2(n)$ , ne segue che

$$T(n) \geq T_2(n)$$

$$T(n) \geq \Theta(\sqrt{2^n})$$

$$T(n) = \Omega(\sqrt{2^n})$$

Poiché il **limite inferiore** e il **limite superiore** differiscono, non possiamo trovare una funzione asintotica stretta, però concludiamo che il calcolo dei numeri di Fibonacci con tecnica ricorsiva richiede un tempo esponenziale in  $n$ , visto che per opportune costanti  $c_1$  e  $c_2$ , otteniamo

$$c_1 \cdot \sqrt{2^n} \leq T(n) \leq c_2 \cdot 2^n$$

# Metodo dell'albero

giovedì 24 ottobre 2024 18:37

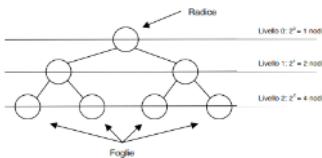
Il metodo dell'albero è la **rappresentazione grafica del metodo iterativo**.

**Idea:** Rappresentare graficamente lo sviluppo del costo computazionale per valutare più facilmente il costo

**Dificoltà:** come il metodo iterativo

In un albero binario completo di **altezza  $h$** :

- Il num. di foglie è  $2^h$
  - Il num. di nodi interni è
- $$\sum_{i=0}^{h-1} 2^i = \frac{2^h - 1}{2 - 1} = 2^h - 1$$
- Il num. tot. di nodi è  $2^h + 2^h - 1 = 2^{h+1} - 1$



## Punti di attenzione

- Quanti nodi ci sono al liv.  $i$ ?
- Qual è il costo di un nodo al liv.  $i$ ?
- Qual è il contributo (costo complessivo dei nodi) al livello  $i$ ?
- Qual è # di livelli? quando ci fermiamo?
- Qual è il costo complessivo di  $T(n)$ , dato dalla somma dei contributi di tutti i livelli

**Non applicabile** quando la struttura della ricorrenza **non si presta** ad una **suddivisione chiara** o quando il num. di chiamate ricorsive **non segue un modello semplice**.

**Problemi tipici non applicabili:**

- Se la funz. di ricorrenza **non si suddivide** in problemi di dim. costante
- Se il num. di sottoproblemi **cambia** in modo **non prevedibile**
- Se i costi delle chiamate ricorsive **non sono facilmente sommabili** in una forma chiusa

**Esempio:**

$$T(n) = \begin{cases} T(n) = 2T\left(\frac{n}{4}\right) + \Theta(\sqrt{n}) \\ T(1) = \Theta(1) \end{cases}$$

- Ad ogni liv.  $i$  abbiamo  $2^i$  nodi, poiché ad ogni nodo avremmo 2 chiamate ricorsive
- Costo di un nodo al liv.  $i$  sarà  $\frac{\sqrt{n}}{4^i} = \frac{\sqrt{n}}{2^{2i}} = \Theta\left(\frac{\sqrt{n}}{2^i}\right)$
- Costo complessivo al liv.  $i$  sarà  $2^i * \frac{\sqrt{n}}{2^i} = \Theta(\sqrt{n})$
- Ci fermiamo quando  $\frac{n}{4^i} = 1 \Rightarrow i = \log_4(n)$ , quindi l'altezza dell'albero è  $\Theta(\log(n))$
- Costo complessivo: dato dalla somma dei costi di tutti i livelli. Poiché ogni liv. ha costo  $\Theta(\sqrt{n})$  e l'altezza dell'albero è  $\Theta(\log(n))$ , il costo è:

$$\sum_{i=0}^{\log_4(n)} \Theta(\sqrt{n}) = \Theta(\sqrt{n}) * \Theta\left(\sum_{i=0}^{\log_4(n)} 1\right) = \Theta(\sqrt{n}) * \Theta(\log(n)) = \Theta(\sqrt{n} * \log(n))$$

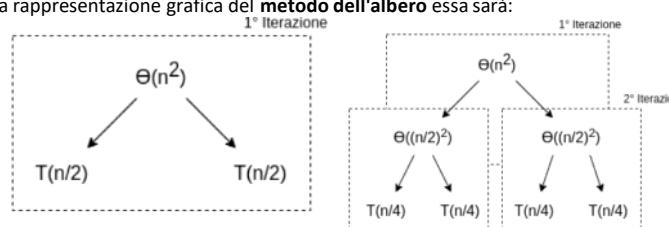
Oppure potevamo scrivere: ad ogni liv.  $i$  ci sono  $2^i$  chiamate ricorsive, ciascuna con costo  $\Theta\left(\frac{\sqrt{n}}{2^i}\right)$ :

$$\sum_{i=0}^{\log_4(n)} 2^i * \Theta(\sqrt{n}) * \Theta\left(\frac{1}{2^i}\right) = \Theta(\sqrt{n}) * \sum_{i=0}^{\log_4(n)} 2^i * \frac{1}{2^i} = \Theta(\sqrt{n}) * \sum_{i=0}^{\log_4(n)} 1 = \Theta(\sqrt{n}) * \Theta(\log(n)) = \Theta(\sqrt{n} * \log(n))$$

**Esempio differenza tra metodo iterativo e dell'albero:**

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

- Col **metodo iterativo**, rappresentiamo  $T(n)$  nella forma:  
 $T(n) = 2T(n/2) + \Theta(n^2)$
- Con la rappresentazione grafica del **metodo dell'albero** essa sarà:



- Sviluppando la seconda iterazione, otteniamo

$$T(n) = 2\left(2T\left(\frac{n}{2}\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right)\right) + \Theta(n^2) = 4T\left(\frac{n}{2}\right) + 2\Theta\left(\left(\frac{n}{2}\right)^2\right) + \Theta(n^2)$$

- Sviluppando la terza iterazione, otteniamo

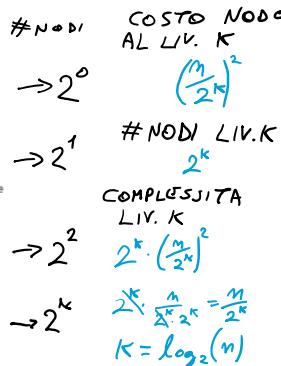
$$T(n) = 2\left(2\left(2T\left(\frac{n}{2^3}\right) + \Theta(n^2)\right) + \Theta(n^2)\right) + \Theta(n^2) = 8T\left(\frac{n}{2^3}\right) + 4\Theta\left(\left(\frac{n}{2^2}\right)^2\right) + 2\Theta\left(\left(\frac{n}{2}\right)^2\right) + \Theta(n^2)$$

- A questo punto effettuiamo il passaggio di **generalizzazione** del metodo iterativo della  $k$ -esima iterazione

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + \left(\sum_{i=0}^k 2^i * \Theta\left(\left(\frac{n}{2^i}\right)^2\right)\right) = 2^k T\left(\frac{n}{2^k}\right) + \left(\sum_{i=0}^k \Theta\left(\frac{n^2}{2^i}\right)\right) = 2^k T\left(\frac{n}{2^k}\right) + \Theta(n^2) * \sum_{i=0}^k \frac{1}{2^i}$$

Le iter. vengono eseguite finché

$$\frac{n}{2^k} = 1 \Rightarrow k = \log_2(n)$$



Sostituendo k otteniamo

$$T(n) = 2^{\log_2(n)-1} * \Theta(1) + \Theta(n^2) * \sum_{i=0}^{\log_2(n)} \frac{1}{2^i} = n * \Theta(1) + \Theta(n^2) * \left(2 - \frac{1}{2^{\log_2(n)}}\right) = \Theta(n^2) * \left(2 - \frac{1}{n}\right) = \Theta(n^2) * \Theta(1) = \Theta(n^2)$$



$= \Theta(1)$  per n grande

Analizziamo ora cosa accade ad ogni **livello dell'albero** che abbiamo creato

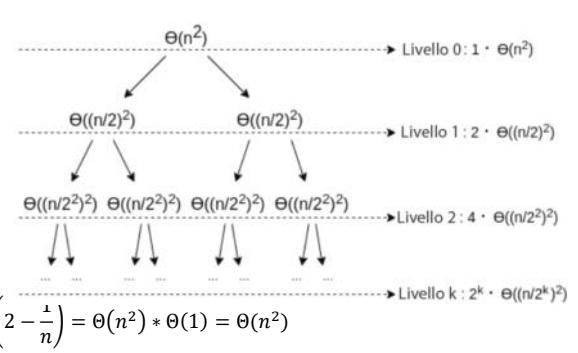
- Ad ogni liv. i abbiamo  $2^i$  nodi, poiché ad ogni nodo avremmo 2 chiamate ricorsive
- Costo di un nodo al liv. i sarà  $\binom{n}{2^i}^2$
- Costo complessivo al liv. i sarà  $2^i \binom{n}{2^i}^2 = 2^i * \frac{n^2}{2^{2i}} = \frac{n^2}{2^i}$
- Ci fermiamo quando  $\frac{n}{2^i} = 1 \Rightarrow i = \log_2(n)$ , quindi l'altezza dell'albero è  $\Theta(\log(n))$
- Costo complessivo: dato dalla somma dei costi di tutti i livelli. Poiché ogni liv. ha costo  $\Theta(\frac{n^2}{2^i})$  e l'altezza dell'albero è  $\Theta(\log(n))$ , il costo è:

$$\sum_{i=0}^{\log_2(n)} \frac{n^2}{2^i} = \Theta(n^2) * \sum_{i=0}^{\log_2(n)} \frac{1}{2^i} = \Theta(n^2) * \left(2 - \frac{1}{2^{\log_2(n)}}\right) = \Theta(n^2) * \Theta(1) = \Theta(n^2)$$

Otture possiamo scrivere:

- Sommando tutti i livelli, ottieniamo
- Il num di iter. totale trovato col metodo iterativo  
(ossia trovando il val. di k, che sappiamo essere  $k = \log(n)$ )

$$T(n) = \sum_{i=0}^k 2^i * \Theta\left(\binom{n}{2^i}^2\right) = \Theta(n^2) * \sum_{i=0}^k \frac{n^2}{2^i}$$



Notiamo che abbiamo ottenuto lo stesso risultato tramite gli **stessi calcoli**. Il metodo dell'albero non è altro che una rappresentazione grafica del metodo iterativo.

# Metodo Sostituzione

giovedì 24 ottobre 2024 18:37

## Idea:

- Si ipotizza una soluzione per l'equazione di ricorrenza
- Si verifica (per induzione) se "funziona"

**Difficoltà:** Si deve trovare la funzione più vicina alla vera soluzione, perché tutte le funzioni **più grandi** (se cerchiamo  $O$  grande) o **più piccole** (se cerchiamo  $\Omega$ ) funzionano

Questo metodo serve soprattutto nelle dimostrazioni e si sconsiglia nella pratica

**Non applicabile** quando non si riesce a indovinare una **buona sol. iniziale** per la verifica per induzione.

## Problemi tipici non applicabili:

- Se la sol. dipende da una **sommatoria difficile da gestire**
- Se la verifica per induzione porta a passaggi algebrici **molto complessi** senza una chiara convergenza
- Se la ricorrenza ha un **comportamento irregolare** che rende difficile formulare un'ipotesi plausibile

Esempio:

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

Ipotizziamo le soluzioni per **maggioranza e minoranza**

1) **Eliminiamo la notazione asintotica** dall'equazione, sostituendo i  $\Theta(1)$  con due costanti **c** e **d** fissate

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

2) **Ipotizziamo la soluzione**  $T(n) = O(n)$ , ossia che  $T(n) \leq k^*n$  per una certa costante  $k$  indeterminata

3) **Analizziamo il caso base**, ossia quando  $n = 1$

$$T(1) \leq k^*1$$

Poiché sappiamo che  $T(1) = d$ , otteniamo che la diseguaglianza è vera se e solo se

$$d \leq k$$

4) **Consideriamo il passo induttivo**, per un **n** generico

$$T(n) \leq kn \Rightarrow T(n - 1) + c \leq kn$$

Per **ipotesi**, sappiamo che anche  $T(n - 1) \leq k(n - 1)$  quindi

$$k(n - 1) + c \leq kn$$

$$kn - k + c \leq kn$$

$$c \leq k$$

Dunque la diseguaglianza è vera se e solo se  $c \leq k$

5) Poiché esiste sempre un  $k$  t.c.  $k \geq c$  e  $k \geq d$ , la **soluzione ipotizzata**  $T(n) \leq kn$  è **vera**, dunque  $T(n) = O(n)$

Potevamo anche ipotizzare che  $T(n) \leq kn^2$  pero a noi interessa stimare  $T(n)$  asintoticamente tramite la funzione più piccola

6) **Ipotizziamo la soluzione**  $T(n) = \Omega(n)$ , ossia che  $T(n) \geq h^*n$  per una certa costante  $h$  indeterminata

7) **Analizziamo il caso base**, quando  $n = 1$

$$T(1) \geq h^*1$$

Poiché sappiamo che  $T(1) = d$ , la diseguaglianza è vera se e solo se

$$d \geq h$$

8) **Consideriamo il passo induttivo**, per un **n** generico

$$T(n) \geq hn \Rightarrow T(n - 1) + c \geq hn$$

Per **ipotesi**, sappiamo che anche  $T(n - 1) \geq h(n - 1)$  quindi

$$h(n - 1) + c \geq hn$$

$$hn - h + c \geq hn$$

$$c \geq h$$

Dunque la diseguaglianza è vera se e solo se  $c \geq h$

9) Poiché esiste sempre un  $h$  t.c.  $h \leq c$  e  $h \leq d$ , la **soluzione ipotizzata**  $T(n) \geq hn$  è **vera**, dunque  $T(n) = \Omega(n)$

10) Poiché  $T(n)$  è sia in  $O(n)$  che in  $\Omega(n)$ , possiamo dire che  $T(n) = \Theta(n)$

Potevamo anche **ipotizzare** le soluzioni  $O(n^2)$ ,  $O(2^n)$  e  $\Omega(\sqrt{n})$ ,  $\Omega(\log(n))$  poiché esistono più soluzioni, però l'obiettivo è **stimare i costi asintotici il più stretti possibile**.

### Altro esempio

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

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

- Eliminiamo la nozione asintotica

$$T(n) = 2T\left(\frac{n}{2}\right) + c$$

$$T(1) = d$$

- Ipotizziamo la soluzione  $T(n) \leq kn$

- Caso base:  $d \leq k$

- Passo induttivo:  $T(n) \leq kn \Rightarrow 2T\left(\frac{n}{2}\right) + c \leq kn$

Per ipotesi, sappiamo che  $2T\left(\frac{n}{2}\right) \leq 2k\left(\frac{n}{2}\right)$ , quindi

$$2k\left(\frac{n}{2}\right) + c \leq kn$$

$$kn + c \leq kn$$

Dunque la diseguaglianza è **sempre FALSA**

- Ipotizziamo una nuova soluzione  $T(n) \leq kn - h$

- Caso base:  $d \leq k - h$

- Passo induttivo:  $T(n) \leq kn - h \Rightarrow 2T\left(\frac{n}{2}\right) + c \leq kn - h$

$$2(k\left(\frac{n}{2}\right) - h) + c \leq kn - h$$

$$kn - 2h + c \leq kn - h$$

$$c \leq h$$

- Deduciamo che  $T(n) = O(n)$

- CONTINUARE PER ESERCIZIO DIMOSTRAZIONE  $T(n) = \Omega(n)$

# Metodo dell'Esperto (Principale)

giovedì 24 ottobre 2024 18:38

**Idea:** avere una **formula** per risolvere un'equazione di ricorrenza

**Difficoltà:** funziona solo quando l'equazione è in forma

$$T(n) = a * \Theta\left(\frac{n}{b}\right) + f(n)$$

Dove  $T(1) = \Theta(1)$  e  $f(n)$  può avere qualsiasi costo  $\Theta$

## Enunciato teorema principale

Dove:

- a: num. di sottoproblemi in cui viene diviso il problema
- b: dimensione di ogni sottoproblema
- f(n): costo combinato per dividere e ricombinare i sottoproblemi

Dati  $\alpha \geq 1, \beta > 1$ , una funzione asintoticamente positiva  $f(n)$  ed un'equazione di ricorrenza di forma

$$T = \begin{cases} T(n) = \alpha \cdot T\left(\frac{n}{\beta}\right) + f(n) \\ T(1) = \Theta(1) \end{cases}$$

abbiamo che:

- Se per  $f(n)$  vale che

$$f(n) = O(n^{\log_\beta(\alpha)-\varepsilon})$$

per qualche costante  $\varepsilon > 0$ , allora

$$T(n) = \Theta(n^{\log_\beta(\alpha)})$$

- Se per  $f(n)$  vale che

$$f(n) = \Theta(n^{\log_\beta(\alpha)})$$

allora

$$T(n) = \Theta(n^{\log_\beta(\alpha)} \cdot \log(n))$$

- Se per  $f(n)$  vale che

$$f(n) = \Omega(n^{\log_\beta(\alpha)+\varepsilon})$$

per qualche costante  $\varepsilon > 0$  e se

$$\alpha \cdot f\left(\frac{n}{\beta}\right) \leq c \cdot f(n)$$

per qualche costante  $0 < c < 1$  e per  $n$  abbastanza grande, allora

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

In termini semplici, si possono verificare **tre casi possibili** che dipendono dal confronto tra  $f(n)$  e  $n^{\log_b(a)}$ :

- **Caso 1:** Se il più grande dei due è  $n^{\log_b(a)}$ , il costo è  $\Theta(n^{\log_b(a)})$
- **Caso 2:** Se sono uguali, allora si moltiplica  $f(n)$  per un  $\log(n)$ , il costo è  $\Theta(f(n) * \log(n)) = \Theta(n^{\log_b(a)} * \log(n))$
- **Caso 3:** Se il più grande dei due è  $f(n)$  e  $a * f(n/b) \leq c * f(n)$  con  $c < 1$ , allora il costo è  $\Theta(f(n))$

ATTENZIONE: con "più grande" si intende **polinomialmente più grande**, data la presenza di  $\varepsilon$ .

Quindi  $f(n)$  deve essere asintoticamente più grande rispetto a  $n^{\log_b(a)}$  di un fattore  $n^\varepsilon$  per qualche  $\varepsilon > 0$

**Non applicabile** quando la ricorrenza **non è nella forma standard**  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

**Problemi tipici non applicabili:**

- Se a o b non sono costanti (dipendono da n)
- Se f(n) non è una funz. polinomiale o non può essere confrontata direttamente con  $n^{\log_b(a)}$
- Se la suddivisione del problema non è uniforme (es.  $T(n) = T(n-1) + T\left(\frac{n}{2}\right) + \Theta(1)$ )

## Metodo generale per trovare il val. massimo di a

Nel caso in cui conosciamo il costo di un algoritmo iterativo  $\Theta(g(n))$  e ci viene proposta una versione ricorsiva con equazione:

$$T = \begin{cases} T(n) = a * T\left(\frac{n}{b}\right) + f(n) \text{ per } n \geq x \\ T(n) = \Theta(1) \text{ altrimenti} \end{cases}$$

Dove a è un incognita costante intera positiva con  $a \geq y$ . b e f(n) invece sono conosciute

Per determinare il **valore massimo** che la costante a può avere per fare in modo che l'algoritmo ricorsivo risulti asintoticamente **più efficiente** dell'algoritmo iterativo, dobbiamo:

- 1) Risolvere la ricorrenza usando y al posto di a per determinare il **termine dominante**:

Quindi confrontiamo il costo di f(n) con il costo di  $n^{\log_b(y)}$

- $T(n) = \Theta(n^{\log_b(y)})$  se  $f(n) = O(n^{\log_b(y)-\varepsilon})$
- $T(n) = \Theta(n^{\log_b(y)} * \log(n))$  se  $f(n) = \Theta(n^{\log_b(y)})$
- $T(n) = \Theta(f(n))$  se  $f(n) = \Omega(n^{\log_b(y)+\varepsilon})$  e  $a * f\left(\frac{n}{b}\right) \leq c * f(n)$  con  $c < 1$

Dobbiamo pertanto trovare il caso in cui ci troviamo e il costo della ricorsione in  $\Theta$

## 2) Riscriviamo la disegualanza per trovare il val. massimo di a

Se  $g(n) = \Theta(n^p)$  possiamo calcolare il val di a per la ricorrenza iterativa con  $a = b^p$

Quindi se vogliamo trovare un algoritmo con costo minore dobbiamo avere  $a \leq b^p - 1$

### Esempio

Costo iterativo è  $\Theta(n^2)$  e l'equazione ricorsiva è:

$$T = \begin{cases} T(n) = a * T\left(\frac{n}{4}\right) + \Theta(1) & \text{per } n \geq 4 \\ T(n) = \Theta(1) & \text{altrimenti} \end{cases}$$

- Cominciamo col risolvere la ricorrenza inserendo y al posto di a:

Applicando il metodo principale abbiamo  $f(n) = \Theta(1)$  e  $n^{\log_4(a)} \geq n^{\log_4(2)} = n^{\frac{1}{2}}$  si ha quindi  $f(n) = O(n^{\log_4(a-\varepsilon)})$   
Siamo quindi nel **primo caso** e il costo è  $\Theta(n^{\log_4(a)})$

- Troviamo il massimo di a

Se  $a = 16$  ( $4^2$ ) la ricorrenza ha soluzione  $\Theta(n^{\log_4(16)}) = \Theta(n^2)$  quindi perché l'algoritmo ricorsivo abbia costo inferiore bisogna avere  $a \leq 15$

### Esempio 1: Caso 1

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

- $a = 9, b = 3$
- $f(n) = \Theta(n)$
- $n^{\log_b(a)} = n^{\log_3(9)} = n^2$
- Vediamo facilmente che ci troviamo nel **caso 1** poiché ponendo  $\varepsilon = 1$  otteniamo  $f(n) = O(n^{\log_3(9)-1}) = O(n)$
- Dunque il costo sarà

$$T(n) = \Theta(n^{\log_b(a)}) = \Theta(n^2)$$

### Esempio 2: Caso 2

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

- $a = 1, b = 3/2$
- $f(n) = \Theta(1)$
- $n^{\log_b(a)} = n^{\frac{\log_3(1)}{\frac{3}{2}}} = n^0 = 1$
- Vediamo che ci troviamo nel **caso 2** poiché  $f(n) = n^{\log_b(a)}$   
 $\Theta(1) = \Theta(1)$
- Dunque il costo sarà

$$T(n) = \Theta(f(n)*\log(n)) = \Theta(\log(n))$$

### Esempio 3: Caso 3

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

- $a = 3, b = 4$
- $f(n) = \Theta(n * \log(n))$
- $n^{\log_b(a)} = n^{\log_4(3)} \approx n^{0.7}$
- Vediamo che ci troviamo nel **caso 3** poiché ponendo  $\varepsilon = 0.1$   
 $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$   
 $f(n) = \Omega(n^{\log_b(a)+0.1}) \approx n^{0.8}$
- Tuttavia dobbiamo prima **verificare**  
 $a * f(n/b) \leq c * f(n)$   
 $3 * (n/4) * \log(n/4) \leq c * n * \log(n)$  per qualche  $c < 1$  e n abbastanza grande
- Ponendo  $c = 3/4$  otteniamo  
 $3 * (n/4) * \log(n/4) \leq (3/4) * n * \log(n)$   
 $(3n/4) * \log(n/4) \leq (3n/4) * \log(n)$   
 $\log(n/4) \leq \log(n)$
- Che è vera  $\forall n$
- Dunque il costo sarà

$$T(n) = \Theta(f(n)) = \Theta(n * \log(n))$$

### Esempio 4

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n * \log(n))$$

- $a = 2, b = 2$
- $f(n) = \Theta(n \log(n))$
- $n^{\log_b(a)} = n^{\log_2(2)} = n$
- In questo caso non rientriamo in **nessuno dei tre casi**, poiché  $f(n) \neq n^{\log_b(a)}$  (dunque non è il caso 2) e non esiste un  $\epsilon$  che possa verificare il caso 1 e 3. Di conseguenza, **non possiamo applicare il metodo principale**, e siamo costretti a usare un altro metodo.  
 $f(n) = \Theta(n \log(n))$  è asintoticamente più grande di  $n^{\log_b(a)} = n$ , ma non polinomialmente più grande. Infatti,  $\log(n)$  è asintoticamente minore di  $n^\epsilon$  per qualunque valore di  $\epsilon > 0$

# Insertion Sort

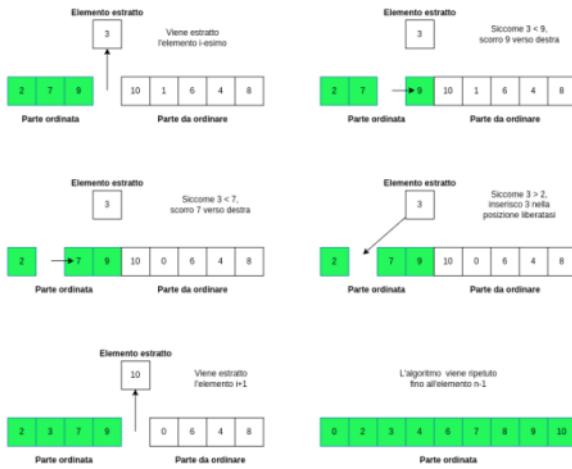
lunedì 28 ottobre 2024 11:38

## Insertion Sort

L'**Insertion Sort** è simile all'**ordinamento di un mazzo di carte**: scorrendo il mazzo analizziamo ogni carta e la mettiamo nella sua posizione corretta, inserendola in mezzo al mazzo di carte già ordinate

Step dell'algoritmo:

- 1) Gli elementi da ordinare sono contenuti inizialmente in un array. Per ogni  $i=0, \dots, n-1$
- 2) Si estrae l'ultimo elemento della posizione  $i$ , così da liberare la sua posizione corrente.
- 3) Si spostano tutti gli elementi alla sua sinistra (già ordinati) che sono maggiori di esso di una posizione verso destra, finché non viene trovato un elemento minore (o uguale) all'elemento estratto
- 4) Si inserisce l'elemento nella posizione che si è liberata
- 5) **INVARIANTE**: ad ogni passo  $i$ , gli elementi con indice  $<i$  (a sinistra) sono già ordinati, quelli con indice  $> i$  (a destra) sono ancora da processare



Costo computazionale:

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

Il num. di iter. del ciclo for esterno è  $n - 1$ , mentre il while interno,  $t_j$ , può andare:

- **Caso migliore**: Se il primo elemento precedente a quello estratto è minore di esso (una sola operazione) (minimo di 1 per ogni  $j$  (se ogni  $x > A[j-1]$ ))
 
$$T(n) = (n - 1) \cdot \Theta(1) = \Theta(n)$$
- **Caso peggiore**: Se ogni elemento precedente a quello estratto è maggiore di esso ( $j$  operazioni) (massimo di  $j$  per ogni  $j$  (se ogni  $x < A[1]$ ))
 
$$T(n) = \sum_{j=0}^{n-1} [\Theta(1) + \Theta(j)] = \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

Lo pseudocodice dell'algoritmo è il seguente.

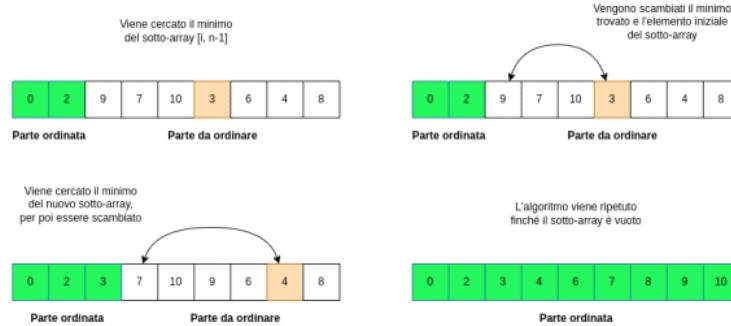
```
def Insertion_Sort(A)
for j in range(1, len(A)):
    (n-1) Θ(1) + Θ(1)
        x = A[j] Θ(1)
        i = j - 1 Θ(1)
        while ((i>=0) and (A[i])>x) tjΘ(1)+ Θ(1)
            A[i+1] = A[i] Θ(1)
            i = i - 1 Θ(1)
        A[i+1] = x Θ(1)
```

# Selection Sort

lunedì 28 ottobre 2024 13:57

Step dell'algoritmo:

- 1) Viene **ricercato il minimo** all'interno dell'array
- 2) Il minimo trovato viene **scambiato di posizione** con il primo elemento dell'array
- 3) Cerca il nuovo minimo nell'array restante, cioè dalla seconda all'ultima e lo cambia con quello in seconda posizione
- 4) Così via, viene cercato l'elemento dalla posizione i fino all'ultima n-1 e lo scambia con l'elemento in posizione i



```
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]
```

$$\begin{aligned} & (n - 1) \Theta(1) + \Theta(1) \\ & \Theta(1) \\ & (n - i) \Theta(1) + \Theta(1) \\ & \Theta(1) \\ & \Theta(1) \end{aligned}$$

Costo computazionale:

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

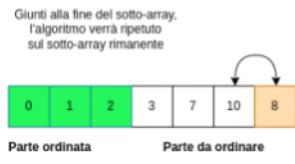
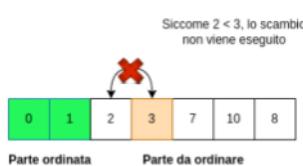
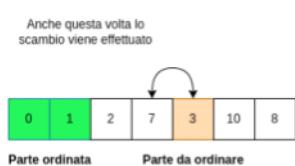
In questo algoritmo non c'è caso peggiore e migliore, e il suo costo è sempre  $\Theta(n^2)$

# Bubble Sort

lunedì 28 ottobre 2024 14:01

Step dell'algoritmo in fasi:

- 1) Partendo **da destra verso sinistra**, vengono analizzate tutte le **coppie di elementi adiacenti**
- 2) Se l'**elemento più a destra è minore del suo elemento precedente**, allora le posizioni vengono **scambiate**, altrimenti no
- 3) Una volta comparata la **coppia più a sinistra**, l'elemento minore dell'intero array risulterà trasportato nella sua posizione finale
- 4) L'algoritmo viene ripetuto sul sotto-array composto dagli altri elementi dell'array



```
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]
```

Costo computazionale:

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

Anche qui non c'è caso peggiore e migliore, e il suo costo è sempre  $\Theta(n^2)$

# Complessità Ordinamento

lunedì 28 ottobre 2024 20:11

Un'algoritmo di **ordinamento** è un algoritmo capace di ordinare gli elementi di un insieme sulla base di una certa relazione d'ordine. Gli algoritmi che abbiamo visto sono:

- **Insertion Sort** (più semplice)
- **Selection Sort** (più semplice)
- **Bubble Sort** (più semplice)

Pero questi tre hanno un costo computazionale di  $\Theta(n^2)$ , vorremmo quindi stabilire un **un limite di costo computazionale** al di sotto del quale **nessun algoritmo di ordinamento basato su confronti fra coppie** di elementi possa andare.

Per stabilire ciò usiamo l'**albero di decisione**, che permette di rappresentare tutte le strade che la computazione di uno specifico **algoritmo può intraprendere**, sulla base dei possibili esiti dei test previsti dall'algoritmo.

Nel caso degli ordinamenti basati sui confronti, **ogni decisione dell'albero sarà solo due possibili esiti:  $a \leq b$  oppure  $a > b$**

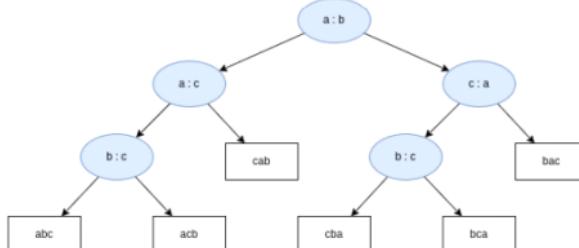
L'albero di decisione relativo a un **ordinamento basato su confronti** ha queste proprietà:

- è un **albero binario** che rappresenta tutti i possibili confronti che vengono effettuati dall'algoritmo; ogni **nodo** possiede due figli, e ogni foglia non ne possiede alcuno
- **Ogni nodo rappresenta un singolo confronto** e i figli del nodo sono relativi ai due possibili esiti di tale confronto



- **Ogni foglia rappresenta una possibile soluzione** del problema, la quale è una specifica **permutazione della sequenza in ingresso**

Esempio: albero di decisione dell'insertion sort su 3 elementi  $a, b, c$ :



Cerchiamo di determinare la **limitazione minima del costo computazionale** dell'algoritmo:

- 1) Il **percorso più lungo** corrisponde al numero di confronti effettuati dall'algoritmo nel caso peggiore
- 2) Dato che la soluzione del problema può corrispondere ad una qualunque delle **permutazioni della sequenza in ingresso**, le **soluzioni possibili** del problema sono  $n!$  preso in input un array di  $n$  elementi
- 3) Un **albero binario di altezza  $h$  non può contenere più di  $2^h$  foglie ( $h = 3, 2^h = 8$ , max foglie dell'albero)**

Quindi ne segue che l'altezza  $h$  deve essere un valore tale per cui:

$$2^h \geq n! \Rightarrow h \geq \log(n!)$$

Valutiamo  $\log(n!)$

$$\sum_{k=1}^n \log(k) \text{ è in } \Theta(n \cdot \log(n))$$

- Verifichiamo l'ipotesi
$$\log(n!) \leq n \cdot \log(n)$$
$$\log(n!) \leq \log(n^n)$$
$$n! \leq n^n$$
- Estendo il fattoriale, e mettiamo in evidenza due categorie di numeri

$$\underbrace{1 \cdot 2 \cdot 3 \cdots}_{\text{Numeri } \leq \frac{n}{2}} \cdot \underbrace{\cdots \cdot (n-2) \cdot (n-1) \cdot n}_{\text{Numeri } \geq \frac{n}{2}} \leq n^n$$

Quindi sappiamo che **tutti i num. minori di  $n/2$  sono anche maggiori di 1**, mentre tutti i num. maggiori di  $n/2$  sono maggiori di  $n/2$

Quindi possiamo riscrivere la seguente disequazione

$$\underbrace{1 \cdot 1 \cdot \dots}_{\frac{n}{2} \text{ volte}} \cdot \underbrace{\frac{n}{2} \cdot \frac{n}{2} \cdot \frac{n}{2}}_{\frac{n}{2} \text{ volte}} \leq 1 \cdot 2 \cdot 3 \cdots \cdot (n-2) \cdot (n-1) \cdot n \leq n^n$$

$$1^{\frac{n}{2}} \cdot \left(\frac{n}{2}\right)^{\frac{n}{2}} \leq n! \leq n^n$$

- Ora riapplichiamo il logaritmo a entrambi le parti della disequazione
$$\log((n/2)^{n/2}) \leq \log(n!) \leq \log(n^n)$$
$$(n/2) * \log(n/2) \leq \log(n!) \leq n * \log(n)$$
$$(n/2) * (\log(n) - 1) \leq \log(n!) \leq n * \log(n)$$
$$\Theta(n * \log(n)) \leq \log(n!) \leq \Theta(n * \log(n))$$
- Poiché  $S_n$  si trova tra due funzioni  $\Theta(n * \log(n))$ , allora **anche esso è in  $\Theta(n * \log(n))$**

Siccome abbiamo visto che  $h \geq \log(n!)$  e che  $\log(n!) = \Theta(n * \log(n))$  allora  $h = \Omega(n * \log(n))$

Dunque possiamo dire che il **costo computazionale di qualsiasi algoritmo di ordinamento basato sui confronti** è  $\Omega(n * \log(n))$

Ora vedremo tre ordinamenti che hanno un costo **più efficiente** raggiungendo il limite inferiore teorico

- **Merge Sort** (più evoluto)
- **Quicksort** (più evoluto)
- **Heap Sort** (più evoluto)

# Merge Sort

lunedì 28 ottobre 2024 19:26

L'algoritmo **Merge Sort** (ordinamento per fusione) è basato sull'utilizzo di una tecnica algoritmica detta **divide et impera**, che può essere descritta come:

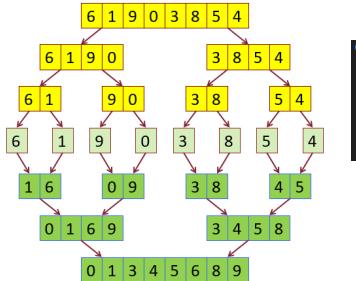
- Il problema complessivo viene diviso in **sottoproblemi** di dimensione inferiore (**divide**)
- I sottoproblemi vengono **risolti** chiamandosi **ricorsivamente** (**impera**)
- Le **soluzioni** dei sottoproblemi vengono **ricomposti** per ottenere la **soluzione** al problema **originale** (**combina**)

Questa tecnica viene usata in diversi algoritmi complessi. Nell'ambito del **Merge Sort**, l'approccio utilizzato è il seguente

- **Divide**: la sequenza di  $n$  elem. viene divisa in due sottosequenze di  $n/2$  elem.
- **Impera**: le due sottosequenze vengono ricorsivamente divise di nuovo in  $n/2$  elem.
- **Caso base**: la ricorsione termina quando la sottosequenza è costituita da un solo elemento, quindi già ordinata
- **Combina**: le sottosequenze già ordinate vengono fuse in un'unica sequenza, fino a tornare alla sequenza di dimensione  $n$

Vediamo come il vero ordinamento avviene nello step **combina**, dove un **sottoalgoritmo** interno al Merge Sort stesso, che chiameremo "**Fondi**", unisce le due sottosequenze creandone una ordinata

L'implementazione del Merge Sort, escludendo "Fondi", risulta essere estremamente facile poiché l'intero "lavoro sporco" viene effettuato dalla ricorsione:



```
def merge_sort(A, primo_i, ultimo_i):
    # Divide ricorsivamente l'array in due sottoarray più piccoli fino al singolo elemento e infine fonde insieme i sottoarray ordinandoli
    if primo_i < ultimo_i:
        medio_i = (primo_i + ultimo_i) // 2 # calcola l'indice medio dell'array
        merge_sort(A, primo_i, medio_i) # ordina il primo sottoarray
        merge_sort(A, medio_i + 1, ultimo_i) # ordina il secondo sottoarray
        return fondi(A, primo_i, medio_i, ultimo_i) # fonde i due sottoarray ordinati
```

Costo computazionale:

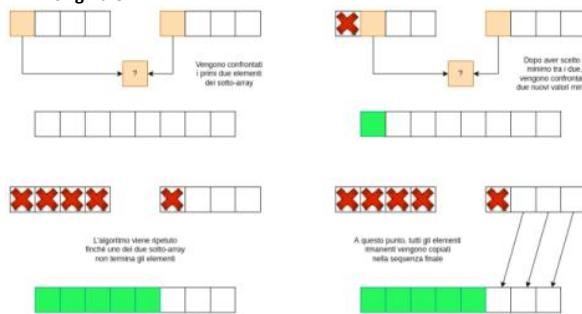
$$T(n) = \begin{cases} T(n) = \Theta(1) + 2T\left(\frac{n}{2}\right) + S(n) \\ T(1) = \Theta(1) \end{cases}$$

Dove  $S(n)$  è il costo di Fondi

## Algoritmo Fondi

L'algoritmo Fondi sfrutta una condizione data per assunta, ossia il fatto che **entrambe le sottosequenze siano ordinate**. Ciò risulta sempre verificato perché il Merge Sort viene richiamato ricorsivamente sulle sottosequenze ordinandole

- Poiché le due sottosequenze sono già ordinate, il **minimo** tra il **primo elemento della prima sottosequenza** e il **primo elemento della seconda sottosequenza** sia il minimo complessivo della sequenza originale, venendo quindi spostato in quest'ultima
- Dopo aver rimosso l'elemento minimo, l'algoritmo viene ripetuto, confronta il **primo elem. della sottosequenza rimanente** e il **primo elem. dell'altra sottosequenza** da cui non è stato rimosso l'elemento
- Non appena una delle sottosequenze ha **terminato i suoi elementi**, tutti gli elementi rimasti nell'altra sottosequenza vengono messi in **coda alla sequenza originale**



```
def fondi(A, primo_i, medio_i, ultimo_i):
    i, j = primo_i, medio_i + 1 # i e j sono gli indici dei due sottoarray da fondere
    B = []
    while i <= medio_i and j <= ultimo_i: # finché nessuno dei due sottoarray è terminato
        if A[i] <= A[j]: # confronta gli elementi dei due sottoarray, se il minore è nel primo sottoarray
            B.append(A[i]) # aggiunge l'elemento minore al vettore B
            i += 1 # incrementa l'indice del primo sottoarray
        else: # se l'elemento minore è nel secondo sottoarray
            B.append(A[j]) # aggiunge l'elemento minore al vettore B
            j += 1 # incrementa l'indice del secondo sottoarray

    while i <= medio_i: # il primo sottoarray non è terminato quindi aggiunge gli elementi rimasti nel sottoarray a B
        B.append(A[i])
        i += 1

    while j <= ultimo_i: # il secondo sottoarray non è terminato quindi aggiunge gli elementi rimasti nel sottoarray a B
        B.append(A[j])
        j += 1

    for i in range(len(B)): # copia il vettore ordinato in A
        A[primo_i + i] = B[i]
    return A
```

Valutazione costo computazionale di Fondi():

- Inizializzazione variabili:  $\Theta(1)$
- 1° ciclo while:
  - Ogni iter. ha costo  $\Theta(1)$  e incrementa di 1 l'indice  $i$  o  $j$ . Quindi il costo del ciclo varia:
    - da un **minimo di  $n/2$**  (nel caso in cui tutti i minimi si trovino in una sola delle due sottosequenze)
    - ad un **massimo di  $n$**  (nel caso in cui le sottosequenze terminano gli elementi in contemporanea)
- Il suo costo è  $\Theta(n)$
- 2° e 3° ciclo while (mai eseguiti entrambi):
  - Poiché non vengono mai eseguiti entrambi, sono considerati come un ciclo unico (poiché svolgono le stesse operazioni).
  - Si ricopia nell'array  $B$  gli elementi rimanenti nella sottosequenza non terminata, il numero di iter. varia:
    - da un **minimo di 1** (quando è rimasto un solo elem. nella sottosequenza)
    - ad un **massimo di  $n/2$**  (coincidente al caso migliore del primo ciclo)
- Il suo costo è  $O(n)$
- Ultimo ciclo:
  - Copia dell'array  $B$  nell'opportuna porzione dell'array  $A$
- Il suo costo è  $\Theta(n)$

Dunque  $S(n) = \Theta(1) + \Theta(n) + O(n) + \Theta(n) = \Theta(n)$

Quindi l'equazione finale del Merge Sort sarà

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

Usando il metodo principale (o qualunque altro metodo) vediamo che il **costo computazionale del Merge Sort** corrisponde a  $\Theta(n \log n)$ , risultando nel costo più efficiente possibile per un algoritmo di ordinamento basato sui confronti

## Osservazioni e Miglioramenti

La **fusion** non si può effettuare in "loco" (ossia aggiornando direttamente l'array A, senza incorrere in un aggravio del costo). Poiché bisognerebbe far spazio via via al **minimo successivo**, dovendo spostare di una posizione tutta la sottosequenza rimanente, il che costerebbe  $\Theta(n)$  per ciascun elemento da inserire, facendo aumentare il costo della fusione da  $\Theta(n)$  a  $\Theta(n^2)$ .

Nell'equazione di ricorrenza

$$T(n) = 2T(n/2) + \Theta(n^2) \text{ il cui costo finale sarebbe } \Theta(n^2). \text{ Peggioro rispetto all'originale } \Theta(n \log(n))$$

Nonostante la velocità del Merge Sort, i fattori costanti sono tali che l'**Insertion Sort** (costo di  $O(n^2)$ ) è più veloce del Merge Sort **per valori piccoli di n**

Dunque possiamo ipotizzare che abbia senso usare l'**Insertion Sort** all'interno del **Merge Sort** quando i sottoproblemi diventano sufficientemente piccoli. Ipotizziamo l'algoritmo con **limite k** che stabilisce se **continuare** la catena di ricorsione o **interromperla** e usare l'Insertion Sort.

```
def Merge_Insertion (A, k, primo, ultimo, dim):
    if dim>k:
        medio =(primo+ultimo)//2
        Merge_Insertion (A,k, primo, medio, medio-primo+1)
        Merge_Insertion (A,k,medio+1, ultimo, ultimo-medio)
        Fondi(primo, medio, ultimo)
    else:
        InsertionSort(primo, ultimo)
Con chiamata Merge_Insertion(A, k, 0, n-1, n)
```

$$T(n) = \begin{cases} T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) \\ T(k) = O(k^2) \end{cases}$$

Calcoliamo il suo costo col metodo iterativo

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = 2 \left[ 2T\left(\frac{n}{4}\right) + \Theta\left(\frac{n}{2}\right) \right] + \Theta(n) = \dots$$

$$T(n) = 2^h \cdot T\left(\frac{n}{2^h}\right) + \sum_{i=0}^{h-1} 2^i \cdot \Theta\left(\frac{n}{2^i}\right) = 2^h \cdot T\left(\frac{n}{2^h}\right) + \sum_{i=0}^{h-1} \Theta(n)$$

Il **caso base** viene raggiunto quando

$$n/2^h = k$$

$$2^h = n/k$$

$$h = \log(n/k)$$

$$T(n) = 2^{\log(\frac{n}{k})} \cdot \Theta(k^2) + \sum_{i=0}^{\log(\frac{n}{k})-1} \Theta(n) = \frac{n}{k} \cdot \Theta(k^2) + \Theta\left(n \log\left(\frac{n}{k}\right)\right) =$$

$$= \Theta(nk) + \Theta(n \log(n)) - \Theta(n \log(k))$$

Se  $k = O(\log(n))$  (dunque  $k \leq c \log(n)$ ), otteniamo che

$n^* \log(n)$  cresce più velocemente di  $n^* \log(\log(n))$  quindi il termine  $\Theta(n^* \log(\log(n)))$  è trascurabile rispetto a  $\Theta(n^* \log(n))$

$$T(n) = \Theta(nk) + \Theta(n \log(n)) - \Theta(n \log(\log(n)))$$

Quindi se per valori  $n \leq c^* \log(n)$  viene usato l'Insertion Sort internamente al Merge Sort, otteniamo un **costo computazionale invariato ma una riduzione notevole del costo in termini di memoria**, poiché l'Insertion Sort è in grado di lavorare in "loco"

## Calcolo Costo di Fondi() con metodo principale

### Metodo Principale

$$\begin{aligned} T(n) &= \Theta(1) + 2T(n/2) + S(n) = \Theta(1) + 2T(n/2) + \Theta(n) = 2T(n/2) + \Theta(n) \\ T(1) &= \Theta(1) \end{aligned}$$

$$\alpha = 2$$

$$\beta = 2$$

$$f(n) = \Theta(n)$$

$$n^{\log_2(2)} = n^1 = n$$

- Ci troviamo nel **caso 2**

$$f(n) = \Theta(n^{\log_2(2)})$$

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

- Dunque il costo sarà

$$T(n) = \Theta(f(n) * \log(n)) = \Theta(n * \log(n))$$

## Quick Sort

martedì 29 ottobre 2024 14:54

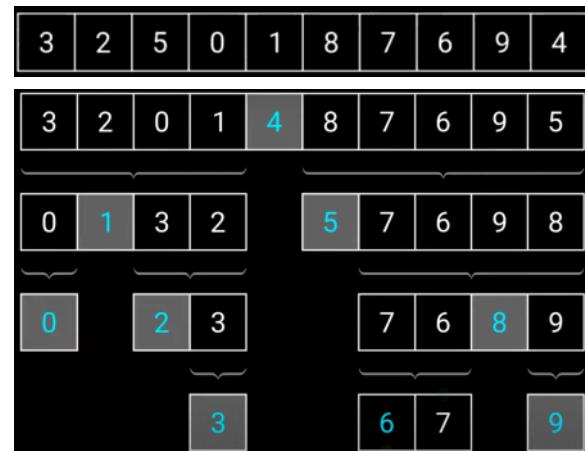
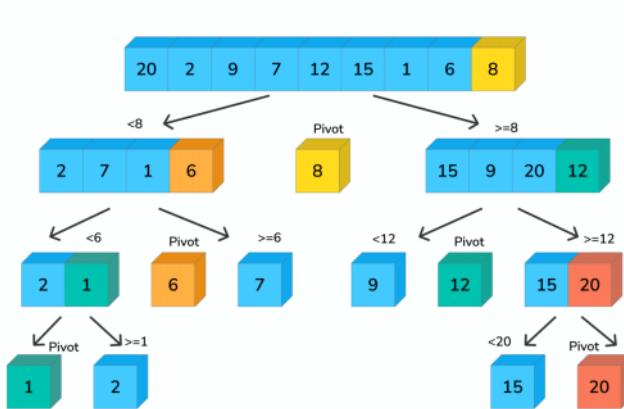
L'algoritmo **Quicksort** riunisce i vantaggi del Selection Sort (**ordinamento in loco**) e del Merge Sort (**ridotto tempo di esecuzione**). Nonostante abbia un costo nel caso peggiore di  $O(n^2)$ , nella pratica è spesso la soluzione migliore per grandi valori di  $n$  perché:

- ha un **costo medio** di  $\Theta(n \log n)$
- i fattori costanti nascosti sono molto piccoli
- Permette l'ordinamento "in loco"

Anche l'algoritmo Quicksort è un algoritmo ricorsivo che adotta la tecnica di "**divide et impera**":

- **Divide**: nella sequenza di  $n$  elem. si seleziona un valore **pivot**. Normalmente il pivot scelto corrisponde al primo, all'ultimo o al valore a metà dell'array su cui si applica l'algoritmo
- La sequenza viene divisa in due sottosequenze: quella con elem.  $<$  pivot e quella degli elem.  $>$  pivot
- **Impera**: le due sottosequenze sono ordinate ricorsivamente
- **Caso base**: la ricorsione termina quando la sottosequenza contiene un solo elemento quindi è già ordinata
- **Combina**: non occorre

Rispetto al Merge Sort, le sottosequenze non vengono fuse tra di loro, poiché il vero e proprio **ordinamento** viene svolto nella **fase di continua divisione ordinata ricorsiva** effettuata dall'algoritmo, ossia il **partizionamento**



```
def quick_sort(A, primo_i, ultimo_i):
    if primo_i < ultimo_i: # se l'array ha più di un elemento
        medio_i = Partiziona(A, primo_i, ultimo_i) # partiziona l'array e restituisce l'indice del pivot
        quick_sort(A, primo_i, medio_i - 1) # ordina il sottoarray sinistro
        quick_sort(A, medio_i + 1, ultimo_i) # ordina il sottoarray destro
    return A

def Partiziona(A, primo_i, ultimo_i):
    pivot = A[primo_i] # il pivot è il primo elemento
    i = primo_i + 1 # i è l'indice del primo elemento successivo al pivot
    for j in range(primo_i + 1, ultimo_i + 1): # scorre tutti gli elementi
        if A[j] < pivot: # se l'elemento è minore di pivot
            A[i], A[j] = A[j], A[i] # scambia l'elemento minore di pivot con il primo elemento successivo al pivot
            i += 1 # incrementa l'indice del primo elemento successivo al pivot
    A[i-1], A[primo_i] = A[primo_i], A[i-1] # scambia il pivot con l'ultimo elemento minore di esso
    return i - 1 # restituisce l'indice del pivot
```

Il costo di **Partiziona()**:

- linee 1-2: num. costante di op. preliminari  $\Theta(1)$
- linee 3-6: una scansione della sequenza, compiendo un num. di operazioni pari alla **dimensione dell'array**:  $\Theta(n)$
- linee 7-8: op. elementare finale, che posiziona il pivot nella sua posizione definitiva e ne ritorna l'indice,  $\Theta(1)$

Dunque il **costo di Partiziona** è  $\Theta(n)$

**Partiziona()** poi **divide la sequenza originale in due sottosequenze**:

- la prima con  $k$  elem.
- la seconda con  $n-k$  elem.

L'equazione di ricorrenza sarà

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

Dove  $P(n)$  è il costo di **Partiziona**, che sappiamo essere  $\Theta(n)$ , quindi:

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

Dove  $0 \leq k \leq n-1$

Poiché non possiamo calcolare un costo strettamente asintotico tramite alcun metodo, possiamo ipotizzare il suo **caso peggiore, migliore e medio**:

- **Caso migliore**: caso in cui ad ogni ricorsione ogni **sottoproblema** viene **diviso a metà**, ossia quando  $k = \frac{n}{2}$ :

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

Vediamo che è, in modo simile al Merge Sort, ha soluzione

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

- **Caso peggiore:** caso in cui ad ogni ricorsione, la dimensione di uno dei due sottoproblemi è 1, e l'altro di n-1, ossia quando k = 1:

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

Applicando il **metodo iterativo** abbiamo:

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

Quindi il costo è

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

- **Caso medio:** poiché il caso peggiore e migliore non coincidono, valutiamo il caso medio

Ipotizziamo che il pivot suddivida con **uguale probabilità**  $\frac{1}{n-1}$  la sequenza in due sottosequenze di dimensioni k e n-k **per tutti i valori di k tra 0 e n-1**

Dunque l'equazione di ricorrenza sarà:

$$T(n) = \frac{1}{n-1} \left[ \sum_{k=0}^{n-1} T(k) - T(n-k) \right] + P(n)$$

Il cui **costo computazionale** è, usando il **metodo della sostituzione**,  $\Theta(n^2 \log(n))$

Dunque, poiché sappiamo che tale equazione è  $\Theta(n^2 \log(n))$  e che il **teorema della complessità di un algoritmo di ordinamento basato sui confronti** [2] impone che esso sia anche  $\Omega(n^2 \log(n))$ , possiamo dire che il **caso medio sia  $\Theta(n^2 \log(n))$**

Quindi il costo dell'algoritmo di **Quicksort** è quasi sempre  $\Theta(n^2 \log(n))$ , risultando l'**algoritmo ideale per input di grandi dimensioni**, poiché a differenza del Merge Sort esso svolge le operazioni in loco.

A volte però l'ipotesi di equiprobabilità non è soddisfatta (es. quando i valori in input sono "poco disordinati"). Difatti il **caso peggiore** è quando la **lista è già ordinata e il pivot è il primo elemento dell'array**, quindi il minore di tutti i valori a suo seguito, andando a ogni volta a dividere l'array in due sotto-array di lunghezza 1 e n-1, risultando in un costo pari a  $\Theta(n^2)$

Per risolvere tale problema possiamo rendere l'**algoritmo indipendente dall'input**:

- Prima di avviare l'algoritmo, la **sequenza viene randomizzata**, evitando che l'array sia parzialmente ordinato
- Durante il partizionamento **viene scelto un pivot casuale** nella sequenza

## Calcolo costo Caso medio

per ogni val. di k = 1, 2, ..., n-1 il termine T(k) compare due volte nella sommatoria, la prima quando k = i e la seconda quando k = n-i.

Valutiamo quindi il valore di:

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

Usiamo il metodo della sostituzione:

- Eliminiamo la notazione asintotica

$$\begin{aligned} T(n) &= \frac{2}{n-1} \sum_{q=1}^{n-1} T(q) + \Theta(n) \longrightarrow T(n) = \frac{2}{n-1} \sum_{q=1}^{n-1} T(q) + hn \\ T(1) &= \Theta(1) \quad T(1) = k \end{aligned}$$

Per ragioni che saranno chiare tra breve, calcoliamo:

$$T(2) = \frac{2}{2-1} \sum_{q=1}^1 T(1) + 2h = \frac{2}{1} k + 2h = 2k + 2h.$$

Ipotizziamo ora la soluzione:

$$T(n) \leq an \log n$$

Sostituendo la soluzione innanzi tutto nel caso base.

Visto che  $\log 1 = 0$ , non possiamo utilizzare  $T(1)$  e sostituendo quindi in  $T(2)$ , ottenendo:

$$T(2) = 2k + 2h \leq 2a \log 2 = 2a$$

che è vera per a opportunamente grande ( $a \geq k + h$ )

Per il passo induttivo scriviamo:

$$\begin{aligned} T(n) &= \frac{2}{n-1} \sum_{q=1}^{n-1} T(q) + hn \leq \frac{2}{n-1} \sum_{q=1}^{n-1} (aq \log q) + hn = \\ &= \frac{2a}{n-1} \sum_{q=1}^{n-1} (q \log q) + hn \end{aligned}$$

Valutiamo ora la sommatoria  $\sum_{q=1}^{n-1} (q \log q)$ , che spezziamo in due:

$$\sum_{q=1}^{n-1} (q \log q) = \sum_{q=1}^{\lfloor \frac{n}{2} \rfloor - 1} (q \log q) + \sum_{q=\lfloor \frac{n}{2} \rfloor}^{n-1} (q \log q)$$

$$\begin{aligned} &\boxed{\leq \log \frac{n}{2} = \log n - 1} \quad \boxed{\leq \log n} \end{aligned}$$

Dunque possiamo scrivere:

$$\begin{aligned}
 \sum_{q=1}^{n-1} (q \log q) &\leq \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q (\log n - 1) + \sum_{q=\lceil \frac{n}{2} \rceil}^{n-1} q \log n = \\
 &= (\log n - 1) \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q + \log n \sum_{q=\lceil \frac{n}{2} \rceil}^{n-1} q = \\
 &= \log n \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q - \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q + \log n \sum_{q=\lceil \frac{n}{2} \rceil}^{n-1} q = \\
 &= \log n \sum_{q=1}^{n-1} q - \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q
 \end{aligned}$$

Ora:

$$\begin{aligned}
 \log n \sum_{q=1}^{n-1} q - \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q &= \log n \frac{(n-1)n}{2} - \frac{1}{2} \left( \left[ \frac{n}{2} \right] - 1 \right) \left[ \frac{n}{2} \right] = \\
 &= \log n \frac{(n-1)n}{2} - \frac{1}{2} \left( \frac{n}{2} - 1 \right) \frac{n}{2} = (\text{perché } \left[ \frac{n}{2} \right] \geq \frac{n}{2}) \\
 &= \frac{1}{2} n(n-1) \log n - \frac{1}{4} \left( \frac{n}{2} - 1 \right) n \leq \\
 &\leq \frac{1}{2} n(n-1) \log n - \frac{1}{4} \left( \frac{n}{2} - 1 \right) (n-1) = \\
 &= (n-1) \left( \frac{1}{2} n \log n - \frac{n}{8} + \frac{1}{4} \right)
 \end{aligned}$$

Ricapitolando:

$$\begin{aligned}
 \sum_{q=1}^{n-1} (q \log q) &\leq \log n \sum_{q=1}^{n-1} q - \sum_{q=1}^{\lceil \frac{n}{2} \rceil - 1} q \leq \\
 &\leq (n-1) \left( \frac{1}{2} n \log n - \frac{n}{8} + \frac{1}{4} \right)
 \end{aligned}$$

Sapendo quindi che:

$$T(n) \leq \frac{2a}{n-1} \sum_{q=1}^{n-1} (q \log q) + hn$$

e che:

$$\sum_{q=1}^{n-1} (q \log q) \leq (n-1) \left( \frac{1}{2} n \log n - \frac{n}{8} + \frac{1}{4} \right)$$

possiamo scrivere:

$$\begin{aligned}
 T(n) &\leq \frac{2a}{n-1} (n-1) \left( \frac{1}{2} n \log n - \frac{n}{8} + \frac{1}{4} \right) + hn = \\
 &= an \log n - \frac{an}{4} + \frac{a}{2} + hn
 \end{aligned}$$

Scegliendo  $a$  sufficientemente grande si ha che:

$$hn - \frac{an}{4} + \frac{a}{2} \leq 0 \text{ e per tale } a \text{ si avrà:}$$

$$T(n) \leq an \log n - \frac{an}{4} + \frac{a}{2} + hn \leq an \log n$$

il che ci permette di dimostrare che

$$T(n) = O(n \log n).$$

# Heap Sort

mercoledì 30 ottobre 2024 17:34

Come il **Quick Sort**, l'**Heap Sort** unisce i vantaggi del **Selection Sort** (Ordinamento in loco) e del **Merge Sort** (costo computazionale  $O(n \log(n))$ ), però, rispetto al Quick Sort, nel caso peggiore il costo è sempre  $O(n \log(n))$ .

Sfrutta l'uso di una **struttura dati** (organizzazione dei dati) detta **Heap**, che garantisce una o più proprietà specifiche, **essenziali** per il corretto funzionamento dell'algoritmo.

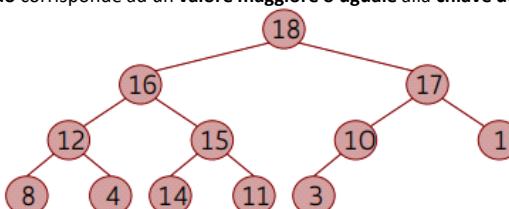
L'**Heap** è stata introdotta per eseguire in modo efficiente le operazioni di:

- **estrazione** (ricerca+rimozione) del massimo
- **inserimento e cancellazione**

Il tempo di esecuzione è

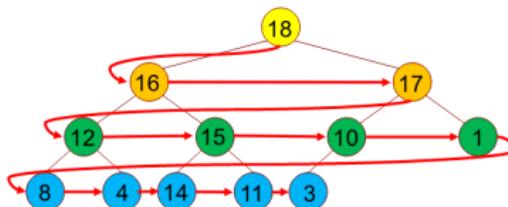
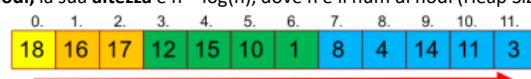
	Estraz. max	Inserim. di una chiave	Cancellaz. di una chiave
Array qualunque	$O(n)$	$O(1)$	$O(1)$
Array ordinato	$O(1)$	$O(n)$	$O(n)$
heap	$O(\log n)$	$O(\log n)$	$O(\log n)$

Un Heap è un **albero binario** completo o quasi completo, in cui tutti i livelli eccetto l'ultimo sono pieni e in cui i nodi sono **addensati a sinistra**, con la proprietà principale dell'**ordinamento verticale**: la **chiave di ogni nodo** corrisponde ad un **valore maggiore o uguale** alla **chiave dei suoi due figli** in ogni nodo.



L'Heap viene comunque **implementato con un array**, i cui indici vanno da 1 al numero di nodi dell'Heap, **Heap Size**, tuttavia deve rispettare alcune caratteristiche:

- L'array è **riempito a partire da sinistra** e se contiene più elementi dell'Heap Size, allora gli elementi di indice > Heap Size non fanno parte dell'Heap
- Ogni nodo dell'albero corrisponde a **un solo elemento** dell'array
- La **radice dell'albero** è  $A[0]$  ed è il **valore massimo dell'Heap**, quindi può essere trovato in tempo  $O(1)$
- Considerato il **nodo  $A[i]$** , il suo **figlio sinistro**, se esiste, corrisponde all'elemento  $A[2i+1]$ , mentre il **figlio destro** corrisponde a  $A[2i+2]$
- Quindi considerato il nodo  $A[i]$ , il suo **padre** sarà l'elemento  $A[\lfloor \frac{i-1}{2} \rfloor]$ :  $\text{parent}(i) = \lfloor \frac{i-1}{2} \rfloor$
- Poiché **ogni livello dell'Heap contiene  $2^h$  nodi**, la sua **altezza** è  $h = \log(n)$ , dove  $n$  è il num di nodi (Heap Size), quindi **( $\log(n)$ )**



La proprietà di ordinamento verticale implica che per tutti gli elem. (tranne  $A[0]$ ) vale:

$$A[i] \leq A[\text{parent}(i)]$$

e poiché l'elemento **massimo** è la radice, esso può essere trovato in  $O(1)$ .

Per poter usare l'**algoritmo Heap Sort** è necessario **modificare tale vettore** in modo che rispetti le **proprietà della struttura Heap**. Ciò viene realizzato usando due funzioni:

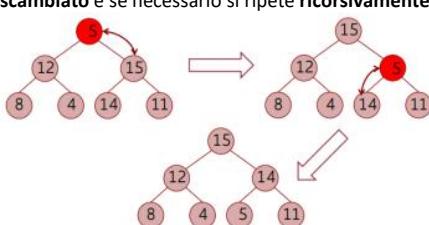
- **Heapify**: ripristina la proprietà dell'**ordinamento verticale**; richiede tempo  $O(\log(n))$
- **Buildheap**: trasforma un **array disordinato** in uno **heap**, sfruttando Heapify; richiede tempo  $O(n)$
- **Heapsort**: l'algoritmo di ordinamento stesso; ordina in loco un array disordinato, sfruttando Build\_heap; richiede tempo  $O(n \log(n))$

## Heapify

Questa funzione ha lo scopo di **mantenere la proprietà di Heap**, dando per assunta l'**ipotesi** che nell'albero su cui viene fatta lavorare **sia garantita la proprietà di heap per entrambi i sotto-alberi** della radice

Di conseguenza l'unico modo che può violare le proprietà di Heap è la radice dell'albero che può essere minore di uno o entrambi i figli

La funzione **opera sulla radice confrontandola coi suoi figli** e, se necessario, la **scambia col maggiore** dei figli. Dopo lo scambio si verifica il "**trasferimento**" al figlio **scambiato** e se necessario si ripete ricorsivamente l'operazione su tale nodo.



```
def Heapify(A, i, heap_size):  
    L = 2*i+1 # indice del figlio sinistro  
    R = 2*i+2 # indice del figlio destro  
    max_i = i # indice del massimo tra i, L e R  
    if L < heap_size and A[L] > A[i]: # se il figlio sinistro è maggiore del padre  
        max_i = L # salva l'indice del figlio sinistro  
    if R < heap_size and A[R] > A[max_i]: # se il figlio destro è maggiore del padre  
        max_i = R # salva l'indice del figlio destro  
    if max_i != i: # se il padre non è il massimo  
        A[i], A[max_i] = A[max_i], A[i] # scambia il padre con il massimo  
        Heapify(A, max_i, heap_size) # richiama ricorsivamente Heapify sul figlio massimo  
    return A
```

Tutte le operazioni svolte nella funzione corrispondono a  $O(1)$ , tranne per la chiamata ricorsiva. Dunque l'equazione di ricorrenza si può trovare in due modi

- In base al **cammino di Heapify**:

Si può considerare che **Heapify** effettua un lavoro  $O(1)$  su **ogni nodo** lungo un cammino la cui lunghezza è limitata da  $O(\log(n))$ , per cui l'eq. di ricorrenza è:

$$T(h) = \begin{cases} T(h-1) + O(1) \\ T(1) = O(1) \end{cases}$$

poiché  $h = O(\log(n))$  sappiamo che la soluzione nel caso peggiore è  $T(n) = O(\log(n))$

- In base al **massimo dei nodi** di un **sotto-albero**:

$$T(n) = \begin{cases} T(n) = T(n') + \Theta(1) \\ T(1) = \Theta(1) \end{cases}$$

Dove  $n'$  è il **num. di nodi del sotto-albero più grande** tra figlio destro e sinistro. Poiché ogni livello contiene  $2^h$  nodi,  $n'$  non può essere più grande di  $\frac{2n}{3}$ , situazione che accade quando l'ultimo livello è pieno esattamente a metà. Dunque l'equazione di ricorrenza diventa:

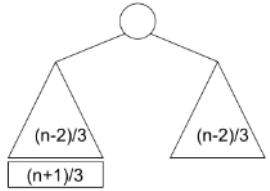
$$T(n) = T\left(\frac{2}{3}n\right) + \Theta(1)$$

$$\bullet \alpha = 1, \beta =$$

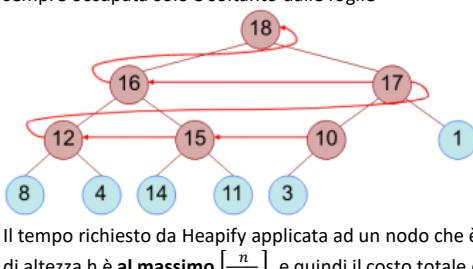
$$\bullet f(n) = \Theta(1)$$

$$\bullet n^{\log_{\beta}(\alpha)} = n^{\log_2(1)} = n^0 = 1$$

• Siamo nel **caso 2**, quindi  $T(n) = O(\log(n))$  (non usiamo  $\Theta(\log(n))$  perché stiamo considerando il **caso peggiore** in cui sotto-albero è riempito al massimo)



Questa funzione si occupa di **trasformare qualunque vettore in un Heap**, chiamando ripetutamente la **funzione Heapify** sugli opportuni nodi dell'Heap. Però siccome Heapify dà per assunto che entrambi i sotto-alberi della radice siano a loro volta in Heap, essa viene **richiamata** scorrendo l'albero dal **basso verso l'alto**. Inoltre, poiché **ogni foglia** (el. in indice  $\lfloor \frac{n}{2} \rfloor + 1$ ) è già un **Heap** (dato che non ha figli), possiamo **ridurre il num. di chiamate a  $n/2$** , poiché la seconda metà dell'array sarà sempre occupata solo e soltanto dalle foglie



```
def Build_heap(A):
    for i in reversed(range(len(A)//2)):
        # scorre tutti i nodi interni al contrario
        Heapify(A, i, len(A)) # chiama Heapify su ogni nodo interno
    return A
```

Il tempo richiesto da Heapify applicata ad un nodo che è radice di un albero di altezza  $h$  (**nodo di altezza  $h$** ) è  $O(h)$ ; inoltre il num. di nodi che sono radice di un sottoalbero di altezza  $h$  è **al massimo**  $\lceil \frac{n}{2^{h+1}} \rceil$ , e quindi il costo totale di Build\_heap è:

$$\sum_{h=0}^{\lfloor \log(n) \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil * O(h) = O\left(n * \sum_{h=0}^{\lfloor \log(n) \rfloor} \frac{h}{2^h}\right)$$

Ora ricordando che se  $|x| < 1$  allora  $\sum_{h=0}^{\infty} h * x^h = \frac{x}{(1-x)^2}$  e ponendo in tale formula  $x = \frac{1}{2}$ , otteniamo:

$$\sum_{h=0}^{\infty} h * \frac{1}{2^h} = \sum_{h=0}^{\infty} h * \frac{1}{2^{h+1}} = \frac{1}{2} * \sum_{h=0}^{\infty} h * \frac{1}{2^h} = \frac{1}{2} * 2 = 1$$

e dunque possiamo scrivere

$$T(n) = O\left(n * \sum_{h=0}^{\lfloor \log(n) \rfloor} \frac{h}{2^h}\right) = O\left(n * \sum_{h=0}^{\lfloor \log(n) \rfloor} \frac{\lfloor \log(n) \rfloor}{2^h}\right) = O(n * \log(n))$$

Poiché la Build\_heap costruisce un heap a partire da **n elementi**, che vanno **almeno letti**, il suo costo è necessariamente  $\Omega(n)$ . Quindi il **costo medio** è  $\Theta(n)$ .

Un'altra analisi **meno accurata**, ma comunque **corretta**, del costo si può spiegare così:

Il **ciclo** viene eseguito  **$n/2$  volte**, e al suo interno vi è solo la chiamata di **Heapify()**, che nel suo **caso peggiore** richiede tempo  $O(h) = O(\log(n))$ .

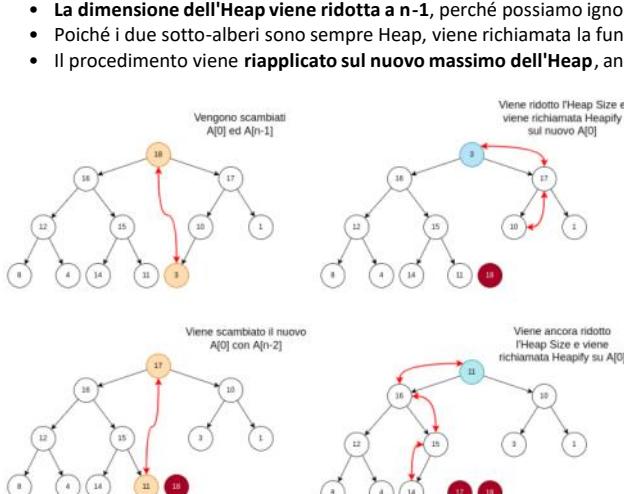
Quindi anche Build\_heap avrà costo  **$O(n*\log(n))$  nel caso peggiore**.

La differenza tra il costo immediato  $\Theta(n*\log(n))$  e  $\Theta(n)$  non influisce sul costo finale dell'algoritmo

## Heap Sort

Funzionamento algoritmo Heap Sort:

- Viene richiamata la **funzione Buildheap**, per trasformare l'array di dim.  $n$  in un **heap (di  $n$  nodi)**
- Poiché il **massimo dell'Heap è in pos.  $A[0]$** , si **scambia con  $A[n-1]$** , ossia l'ultimo elemento per metterlo nella corretta posizione
- La dimensione dell'Heap viene ridotta a  $n-1$** , perché possiamo ignorare l'ultimo elemento siccome è nella posizione finale
- Poiché i due sotto-alberi sono sempre Heap, viene richiamata la funzione **Heapify sulla radice**, in modo da ripristinare le **proprietà dell'Heap**
- Il procedimento viene **riapplicato sul nuovo massimo dell'Heap**, andando man mano a ridurre l'Heap Size fino al minimo ( $(n-2), (n-3), \dots, 2$ )



```
def Heapsort (A) :
    Build_heap(A)
    for x in reversed(range(1, len(A) )) : # (n-1) iteraz.
        A[0], A[x] = A[x], A[0] # (1)
        Heapify(A, 0, x) # O(log n)

T(n) = O(n) + (n-1)(O(1) + O(log(n))) = O(n) + O(n) + O(n*log(n)) = O(n*log(n))

Poiché O(n*log(n)) cresce più velocemente di O(n) e O(n).
```

```
def heap_sort(A):
    Build_heap(A)  # costruisce un heap massimo
    for i in reversed(range(1, len(A))):  # scorre tutti gli elementi al contrario tranne l'ultimo
        A[0], A[i] = A[i], A[0]  # scambia il primo elemento con l'elemento corrente
        Heapify(A, 0, i)  # richiama Heapify sul primo elemento
    return A
```

# Counting Sort

lunedì 4 novembre 2024 12:01

Abbiamo visto algoritmi che **operano per confronti** e quindi hanno **costo computazionale di  $\Omega(n \log n)$** .

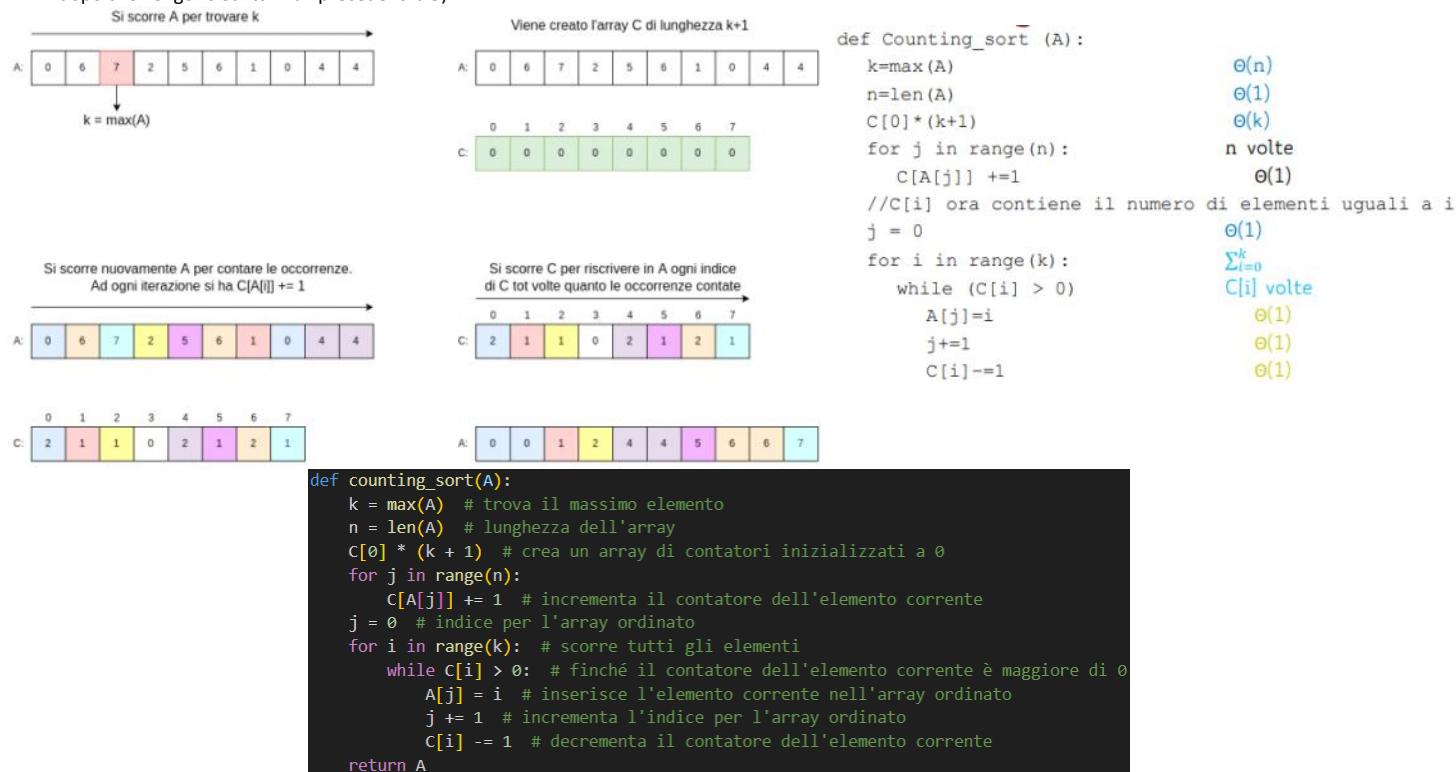
Vediamo ora un algoritmo che riesce ad arrivare ad un **costo computazionale di  $\Theta(n)$**  poiché **non è basato sui confronti**, facendo ipotesi aggiuntive

Ipotizziamo di avere un array di **n interi compresi in un intervallo  $[0..k]$** , ed ha costo  **$\Theta(n+k)$** . Se  $k = O(n)$ , l'algoritmo ordina  $n$  elementi in tempo lineare, cioè  $\Theta(n)$ .

Ora possiamo usare un **secondo array di supporto**, i cui valori corrispondono al **num. di occorrenze dell'indice stesso**.

Infine viene usato il num. di concorrenze per riscrivere nell'array iniziale i valori in **ordine di indice**.

- Viene trovato il **val. massimo di A ( $k$ )**
- Viene creato un **secondo array C** di lunghezza  $k+1$
- Viene **conteggiato** ogni elem. di A, **incrementando di uno il corrispettivo indice nell'array C**. Es. se  $A[i] = 6$ , allora  $C[6]$  viene incrementato di 1
- Viene **sovrascritto** l'array A scorrendo C, ricopriando in A ciascun indice di C tante volte quanto è il val. in C di quell'indice (es. se  $C[5] = 2$ , ci saranno 2 val. 5 dopo che vengono scritti i val. precedenti a 5)



- Il costo di  $\max()$  è  $\Theta(n)$  perché scorriamo tutto A per trovare il val. massimo
- La **somma di tutti i val. in C** corrisponde a  $n$

$$\sum_{i=0}^k C[i] = n$$

Il costo è

$$T(n) = \Theta(n) + \Theta(k) + \Theta(1) + n * \Theta(1) + \sum_{i=0}^k C[i] * \Theta(1) = \Theta(n) + \Theta(k) + \Theta(n) + n * \Theta(1) = \Theta(n) + \Theta(k) = \Theta(k+n) = \Theta(\max(k, n))$$

Il costo computazionale si divide in due casi

- Se  $k \leq n$ , allora  $T(n) = \Theta(n)$
- Se  $k > n$ , allora  $T(n) = \Theta(k)$

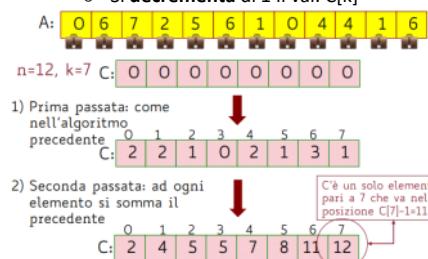
Pero questa versione del Counting Sort è adeguata se non vi sono dati satellite (ossia altri dati legati all'insieme stesso).

Infatti il ciclo che ricopia i val. da C ad A ne sovrascrive i dati

## Counting Sort con Dati Satellite

Se ci sono dati satellite, oltre ad A e C si deve introdurre un **nuovo vettore B** di  $n$  elem. che conterrà la sequenza ordinata:

- Dopo aver conteggiato gli elem. in C, si fa **una seconda passata su C** partendo da  $C[2]$ , nella quale a **ogni elem.  $C[i]$  si somma il precedente**
- Dopo si ha che:
  - $C[i]$  indica la **posizione corretta**, nel vett. ordinato, per l'elem. pari a i più a destra tra quelli in A
  - $C[i] - C[i-1]$  indica **quanti elem. ci sono con valore pari a  $C[i]$**
- Si scorre quindi A da destra a sinistra e per ogni  $A[j]$ :
  - Si **copia in B** ogni elem.  $A[j] = k$  nella pos. giusta che è  $C[k]$
  - Si **decrementa** di 1 il val.  $C[k]$



```

def counting_sort_2 (A):
    k = max(A)
    n = len(A)
    C = [0]*(k+1)
    B = [0]*n

    for j in range(n)
        C[A[j]]+=1
    #in C[i] ora c'è il num. di elem. = i

```

elemento si somma il precedente

A:

```
for j in range(n)
    C[A[j]]+=1
    #in C[i] ora c'è il num. di elem. = i
```

```
for i in range(1,k)
    C[i]+=C[i-1]
    #in C[i] ora c'è il num. di elem. <= i
```

```
for j in range(n,-1)
    B[C[A[j]]]=A[j]
    C[A[j]]-=1
return B
```

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

C:

B:

## Bucket Sort

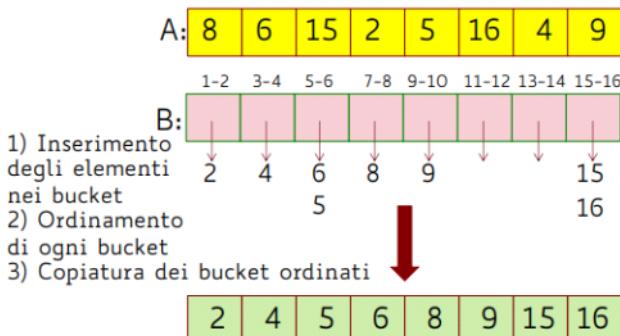
lunedì 4 novembre 2024 13:11

Ipotizziamo di avere  $n$  elem. **distribuiti in modo uniforme** nell'intervallo  $[1 \dots k]$ , senza alcuna ipotesi su  $k$

- Il costo computazionale medio è  $\Theta(n)$

Ideia: dividere l'intervallo  $[...k]$  in **n** **sottointervalli di uguali dimensioni  $k/n$** , detti **bucket**, e distribuire i val. nei bucket **senza confrontare gli elem. tra loro.**  
 Poiché gli elem. in input sono distribuiti in modo uniforme, non ci si aspetta che molti elem. cadano nello stesso bucket

$n=8, k=16$



```

Funzione Bucket_Sort (A; k, n: interi)
for i=1 to n
    inserisci A[i] nella lista corrispondente
                                in B          n Θ(1)
for i=1 to n
    ordina la lista B[i] con insertion_sort
                                 $\sum_{i=1}^n$ (costi di ordinamento)
concatena le liste B[1] B[2] ... B[n]
                                in quest'ordine      Θ(n)
Copia la lista unificata in A
                                Θ(n)

```

I costi di ordinamento dipendono dalla lunghezza delle liste

Ci sono **n bucket** e **n valori**; per l'ipotesi di equidistribuzione, in media ci sarà un num. costante di val. in ogni bucket

Quindi la lista B[i] ha in media lunghezza costante.

Quindi in media  $T(n) = \Theta(n)$

# Strutture Dati

martedì 29 ottobre 2024 12:38

Un **dato** è un valore che una variabile può assumere.

Un **tipo di dato** rappresenta una collezione di valori e un insieme di operazioni ammesse su questi valori (intero, char, stringa)

Le **strutture dati** sono:

- Un **modo sistematico di organizzare dati**
- Un **insieme di operatori** che permettono di manipolare la struttura
- In grado di **memorizzare e manipolare insieme dinamici** (composti solitamente da **chiave**, il cui val. fa parte di un insieme ordinato (es. num. interi) e **dati satelliti**, relativi all'elem. ma non sono direttamente usati nelle operaz. di manipolazione dell'insieme), i cui elem. possono variare nel tempo
- **Lineari o non lineari** (se esiste o no l'organizzazione sequenziale dei val.)
- **Statiche o dinamiche** (se possono variare o no la dimensione nel tempo)
- **Omogenee o disomogenee** (rispetto ai dati contenuti se sono dello stesso tipo o no)

Le strutture dati che vedremo hanno la capacità di memorizzare **insiemi dinamici** (gli elem. possono variare nel tempo) ma differiscono per le **proprietà che le caratterizzano**.

Gli elementi dell'insieme dinamico, denominati **record**, contengono comunemente:

- Una **chiave**, univoca per ogni elemento e usata per distinguerli all'interno dell'insieme
- Ulteriori **dati satellite**, che sono relativi all'elemento stesso ma non sono direttamente usati nelle op. di manipolazione dell'insieme

Sono le proprietà della struttura che **determinano l'importanza della scelta della struttura** da usare quando si deve progettare un algoritmo.

Le **operazioni** che si possono eseguire su una **struttura dati S** sono divise in **due categorie**:

- **Operazioni di interrogazione:**
  - **Search(S, k)**: recupera l'elem. con chiave k, se è presente in S, restituire un valore speciale nullo altrimenti
  - **Min(S)/Max(S)**: recupera il **min/max di val** in un array S
  - **Predecessor(S, k)/Successor(S, k)**: recupera l'elem. in S che **precede/segue k** se S fosse ordinato
- **Operazioni di manipolazione**
  - **Insert(S, k)**: inserisce un elem. k in S
  - **Delete(S, k)**: elimina da S l'elem. k

## Costo operazioni

L'array è una struttura dati semplice con le seguenti caratteristiche:

- Memorizza **elem. omogenei**
- È **statico** (lunghezza definita a priori)
- È ad **accesso casuale** (possiamo accedere ad ogni suo elemento con tempo costante indipendentemente dalla posizione)

Vediamo il costo delle principali operazioni su insiemi dinamici memorizzati su **array come lista statica di elementi, disordinati e ordinati**

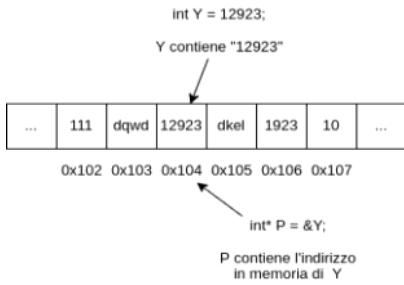
- **Array Disordinato:**
  - **Search**: bisogna scorrere l'array per trovare l'elem.:  $\Theta(n)$
  - **Min/Max**: bisogna scorrere l'array per trovare l'elem.:  $\Theta(n)$
  - **Predecessor/Successor**: bisogna scorrere l'array per trovare l'elem.:  $\Theta(n)$
  - **Insert**: inserimento nella prima pos. libera:  $\Theta(1)$
  - **Delete**: eliminazione e scambio con l'ultimo elem.:  $\Theta(1)$
- **Array Ordinato:**
  - **Search**: ricerca binaria.:  $O(\log(n))$
  - **Min/Max**: primo/ultimo elem.:  $\Theta(1)$
  - **Predecessor/Successor**: Si prende l'elem. precedente o seguente:  $\Theta(1)$
  - **Insert**: è necessario ricercare la posizione in cui effettuare l'inserimento e poi scorrere a destra gli elementi maggiori:  $\Theta(n)$
  - **Delete**: è necessario ricercare la posizione da eliminare e poi scorrere a sinistra per coprire lo spazio lasciato:  $\Theta(n)$

Struttura dati	Search(S,k)	Minimum(S) Maximum(S)	Predecessor(S,k) Successor(S,k)	Insert(S,k)	Delete(S,k)
Array qualunque	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$O(1)$
Array ordinato	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$

# Liste puntate

lunedì 4 novembre 2024 17:01

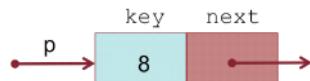
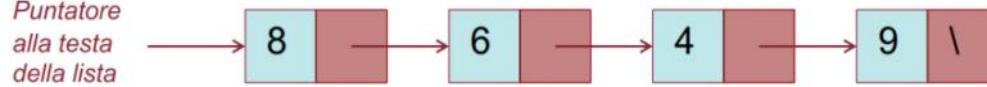
Un **puntatore** è una var. che assume come val. un **indirizzo di memoria**, e non un valore in se per se. Una var. normale fa riferimento al val. **direttamente**, mentre un puntatore lo fa **indirettamente**, puntando all'indirizzo di memoria che contiene tale valore.



Una **lista puntata** è una struttura dati in cui gli elem. sono organizzati in **successione** e dove ogni elem. è un **record a due campi**:

- **Un campo Key:** contiene l'**informazione vera e propria** dell'elem. Viene rappresentato con la notazione  $p \rightarrow \text{key}$
- **Un campo Next:** contiene un **puntatore all'elemento successivo della lista**. Nel caso dell'ultimo elem. della lista, il puntatore sarà un **valore None** (simbolo \)

Viene rappresentato con la notazione  $p \rightarrow \text{next}$



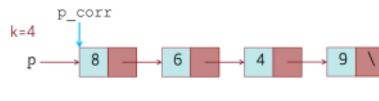
Rispetto agli **array**, che godono di un **accesso diretto**, dove per accedere ad un dato basta conoscerne la posizione nell'array (costo  $\Theta(1)$ ), nelle **liste puntate**, la successione degli elementi viene implementata con un **collegamento esplicito da un elem. all'altro** (ossia il puntatore), rendendo possibile solo l'**accesso sequenziale**.

Dunque l'**accesso** a qualsiasi dato della lista ha un **costo proporzionale alla sua posizione**, nel caso peggiore  $\Theta(n)$

Poiché l'unico accesso è quello sequenziale, si può pensare che la maggior parte delle operazioni svolte sulla lista abbia un **costo  $O(n)$** :

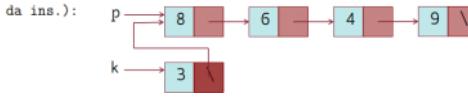
- **Search:** Nel caso peggiore, l'elem. sta in fondo alla lista:  $O(n)$

```
def Search (p: punt. alla testa; k: valore):
    p_corr = p
    while ((p_corr != NULL) and (p_corr -> key != k))
        p_corr = p_corr -> next
    return p_corr
```



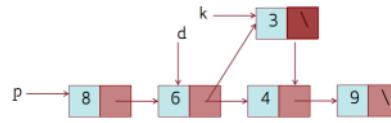
- **Insert in testa:** L'elem. viene aggiunto in testa alla lista:  $O(1)$

```
def Insert_in_testa (p: punt. alla testa; k: punt. a elem. da ins.):
    if (k != None):
        k->next = p
        p = k
    return p
```



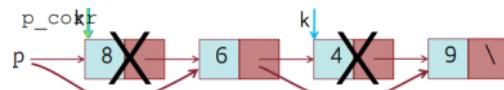
- **Insert dopo un elemento:** Nel caso peggiore, il puntatore dopo cui inserire l'elem. sta in fondo alla lista:  $O(n)$

```
def Insert_Dopo_d(p: punt. alla testa; k: punt. a elem. da ins.; d: punt. dopo cui ins.):
    if d != None:
        k->next = d->next;
        d->next = k;
    return p
else:
    return None
```



- **Delete:** Bisogna trovare l'elemento da eliminare, quindi il costo sarà come il Search:  $O(n)$

```
def Delete (p: punt. alla testa; k: punt. a elem. da canc.):
    if (k != None):
        #se k=null non c'è niente da cancellare
        if k == p:
            #cancella 1° elem
            p = p->next
            return p
        p_corr = p
        while (p_corr-> next != k)
            p_corr = p_corr-> next
            #p_corr punta a elem. che prec. k
        p_corr-> next = k-> next
    return p
```



Siccome le **liste puntate sono strutture dati inherentemente ricorsive**, gli algoritmi delle liste puntate possono essere anche in **versione ricorsiva**

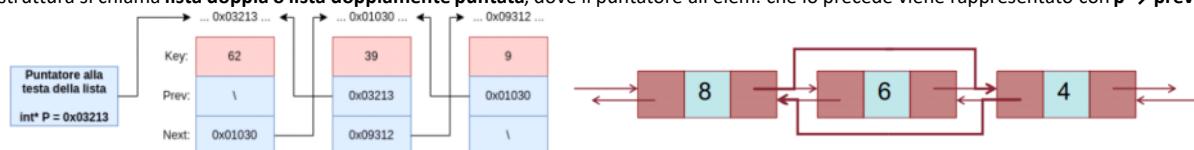
- **Delete Ricorsivo:**

```
def Delete_Ric(p: punt. alla testa; k: punt. a elem. da canc.)
    if p==k:
        p = p->next
    else:
        p->next = Delete_Ric(p->next, k)
    return p
```

## Liste doppiamente puntate

Alcuni problemi delle **liste puntate semplici** possono essere risolti facendo sì che si possa accedere sia all'elemento che lo segue che a quello che lo precede nella lista.

Tale struttura si chiama **lista doppia** o **lista doppiamente puntata**, dove il puntatore all'elem. che lo precede viene rappresentato con  $p \rightarrow \text{prev}$



Le op. di ricerca e inserimento sono analoghe a quelle di una lista semplice, l'op di **eliminazione** invece cambia.

```

def Delete_Doppia(p, k):
    # elimina il nodo col punt. k in una lista doppia
    if k.prev != None: # se il punt. precedente a k non è nullo
        k.prev.next = k.next # il punt. successivo al punt. precedente a k diventa il punt. successivo a k
    else: # se il punt. precedente a k è nullo (quindi k è la testa)
        p = k.next # la testa diventa il punt. successivo a k
    if k.next != None: # se il punt. successivo a k non è nullo
        k.next.prev = k.prev # il punt. precedente al punt. successivo a k diventa il punt. precedente a k
    return p

```

Riassumendo, il costo delle operazioni per le liste puntate semplici e doppie è:

Struttura dati	Search(S,k)	Minimum(S) Maximum(S)	Predecessor(S,k)	Insert(S,k)	Delete(S,k)
Lista semplice	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$O(n)$
Lista doppia	$O(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

Esiste anche la **lista circolare**, nella quale il primo e l'ultimo puntatore sono collegati tra loro.

Essa è utile quando si vuole effettuare una computazione organizzata per fasi, nella quale ciascuna fase richiede la scansione dell'intera struttura dati.

# Pile e Code

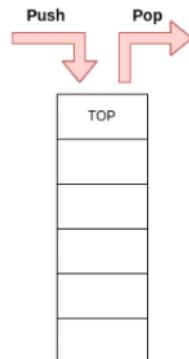
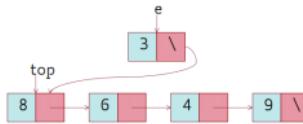
martedì 5 novembre 2024 11:09

## Pile

La **pila** è una struttura dati con comportamento **LIFO (Last In First Out)**, l'ultimo entrato è il primo ad uscire.  
Su una pila possono essere eseguite **solo due operazioni**, entrambe svolte sul **top** della pila:

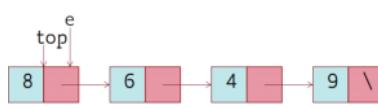
- **Push:** Inserimento del nuovo elemento in cima:  $\Theta(1)$

```
def Push(top: punt.; e: punt. a e. da ins.):
    e->next = top
    top = e
    return top
```



- **Pop:** Estrazione dell'elem. in cima (l'ultimo inserito):  $\Theta(1)$

```
def Pop (top: punt.):
    if (top==None):
        return None
    e = top
    top = e->next
    e->next=None
    return e, top
```



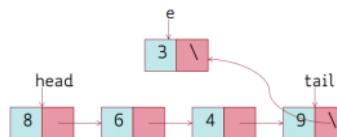
## Code

Diversamente dalla pila, una **coda** è basata sul comportamento **FIFO (First In First Out)**, il primo entrato è il primo ad uscire.  
Gli elem. vengono **estratti** dalla coda esattamente nello **stesso ordine** col quale vi sono stati **inseriti**

Anche sulla coda si possono eseguire **solo due operazioni**:

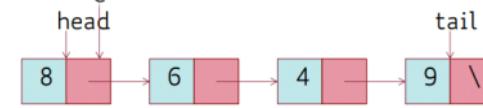
- **Enqueue:** Inserimento del nuovo elem. in coda (**tail**):  $\Theta(1)$

```
def Enqueue (head: punt.; tail: punt.; e: punt. a elem. da ins.):
    if (tail == NULL): # la coda è vuota
        tail = e
        head = e
    else:
        tail->next = e
        tail = e
    return head, tail
```



- **Dequeue:** Estrazione dell'elemento in testa (**head**):  $\Theta(1)$

```
def Dequeue (head: punt.; tail: punt.):
    if (head == None): # la coda è vuota
        return none
    e = head
    head = e->next
    if (head == None):
        tail = None
    return head, tail, e
```



## Coda con priorità

La **coda con priorità** è una variante della coda. L'inserimento e l'estrazione avvengono sempre alle estremità opposte.

A differenza della coda, la **posizione di ciascun elemento** non dipende dal momento in cui è stato inserito, ma **dal valore di una determinata grandezza, detta priorità**, la quale in generale è associata ad uno dei campi presenti nell'elem. stesso.

Di conseguenza, gli elem. di una coda con priorità sono collocati in ordine crescente rispetto alla priorità.

Gli elem. di una coda con priorità sono collocati in **ordine crescente** (o decrescente) **rispetto alla grandezza della priorità**.

Es.

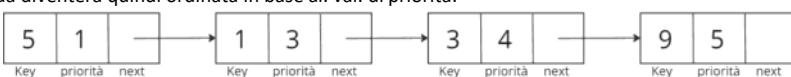
- La priorità è associata al val. numerico del campo **key**  
Un **array ordinato** è una coda con priorità che coincide con la chiave
- Anche l'**heap** è una coda con priorità rispetto alla stessa priorità
- Una **coda** può essere intesa come coda con priorità, in cui la priorità è il maggior tempo di permanenza nella coda stessa
- La **pila** è una coda con priorità, in cui la priorità è il minor tempo di permanenza nella pila stessa

### Esempio Inserimento:

- Vogliamo aggiungere il seguente elem.



- La coda diventerà quindi ordinata in base al. val. di priorità:



La coda con priorità presenta un pericolo di **starvation (attesa illimitata)**: un elemento potrebbe **non essere mai estratto**, se viene **continuamente scavalcato** da altri elem. con priorità più alta che vengono via via immessi nella struttura dati.

# Alberi

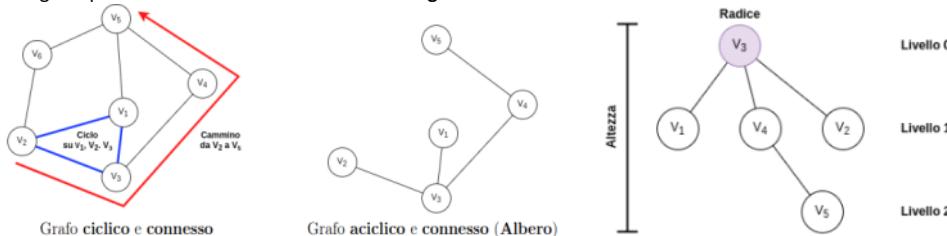
martedì 5 novembre 2024 11:52

Un **albero** è una struttura dati estremamente **versatile**, utile per modellare una grande quantità di situazioni reali e progettare le relative soluzioni algoritmiche. Gli alberi sono dei particolari **tipi di Grafi**.

Un **grafo**  $G = (V, E)$  è costituito da una coppia di elem. (insiemi):

- un insieme finito **V** dei **nodi o vertici**
- un insieme finito **E ⊆ V x V** di coppie non ordinate di nodi, dette **archi o spigoli**
- un **cammino** è una sequenza  $(v_1, v_2, \dots, v_k)$  di nodi distinti di **V** **connessi a due a due** (t.c.  $(v_i, v_{i+1})$  sia un arco di **E**  $\forall 1 \leq i \leq k-1$ )
- se nel cammino  $(v_1, v_2, \dots, v_k)$  i **nodi**  $v_1$  e  $v_k$  (**primo e ultimo**) **coincidono** vi è un **ciclo**
- si dice che un grafo è **connesso** se,  $\forall$  coppia di nodi  $(u, v)$ , esiste almeno un cammino tra  $u$  e  $v$
- un grafo è **aciclico** se non contiene cicli

Definito il grafo possiamo definire la **struttura albero = grafo aciclico e connesso**



Il num. di archi di un albero è uguale a  $|E| = |N| - 1$  ( $E$ : insieme degli archi,  $N$ : num. nodi)

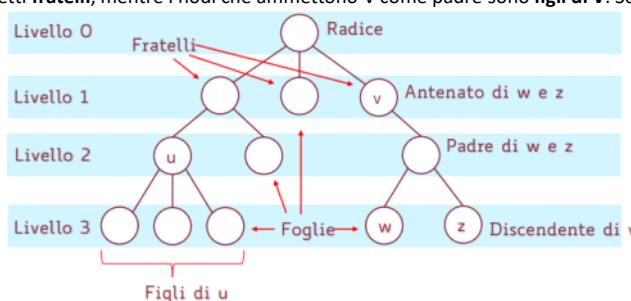
## Alberi radicati

Un **albero radicato** è un albero in cui si distingue un nodo particolare, detto **radice**. L'albero radicato si può rappresentare con i cammini da ogni nodo alla radice seguano un percorso **dal basso verso l'alto**. I **nodi** organizzati in **livelli** (di norma la radice è il liv. 0) e l'**altezza** è la lunghezza del cammino più lungo della radice alla foglia.

Un albero di altezza  $h$  contiene  $(h+1)$  liv., numerati da 0 ad  $h$ .

In un albero radicato distinguiamo i vari **nodi**:

- Dato un nodo  $V$  (che non sia la radice), il **primo nodo che si incontra sul cammino da  $V$  alla radice** viene detto **padre di  $V$**
- I nodi che hanno lo **stesso padre** sono detti **fratelli**, mentre i nodi che ammettono  $V$  come padre sono **figli di  $V$** . Se un nodo non ha figli è una **foglia**



Un albero radicato viene detto **ordinato**, se viene attribuito **un qualche ordine ai figli di ciascun nodo**:

## Alberi binari

Una **sottoclassificazione** di alberi radicati e ordinati è quella degli **alberi binari**, dove **ogni nodo ha massimo due figli** (sinistro e destro).

Un albero binario nel quale **tutti** i livelli contengono **due figli** è chiamato **albero binario completo** (altrimenti si dice **quasi completo**), dove:

- L'**altezza** dell'albero è  $h$
- Il **numero di nodi ad ogni liv.** è  $2^h$ , dove  $i$  è il liv.
- Il **numero di foglie** è  $2^h$
- Il **numero di nodi interno** (ossia i nodi non foglie) è  $\sum_{k=0}^{h-1} 2^k = \frac{2^h - 1}{2 - 1} = 2^h - 1$
- Il **numero totale di nodi** è  $2^h + 2^h - 1 = 2^{h+1} - 1$

Considerando  $n$  come il **num. totale di nodi**, l'altezza  $h$  di un albero completo è:

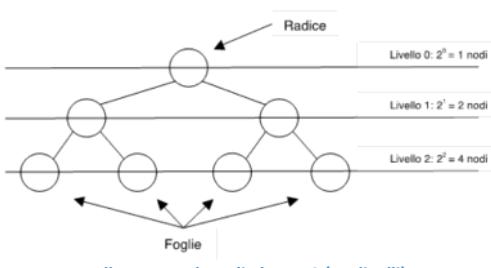
$$\begin{aligned} n &= 2^{h+1} - 1 \\ n + 1 &= 2^{h+1} \\ \log(n + 1) &= \log(2^{h+1}) \\ \log(n + 1) &= h + 1 \\ \log(n + 1) - 1 &= h \\ \log(n + 1) - \log(2) &= h \\ h &= \log\left(\frac{n + 1}{2}\right) \end{aligned}$$

Quindi l'**altezza di un albero completo** è  $h = \Theta(\log(n))$ .

Nel caso di un **albero non completo**, l'altezza è  $O(n)$

## Rappresentazione e memorizzazione degli alberi

Vediamo tre modi per **rappresentare e memorizzare** questo albero binario



Albero completo di altezza 2 (tre livelli)

## Rappresentazione e memorizzazione degli alberi

Vediamo tre modi per rappresentare e memorizzare questo albero binario

- **Tramite puntatori:**

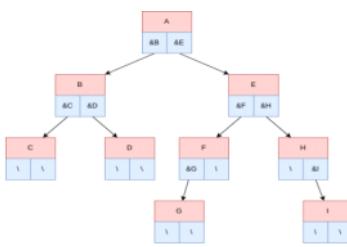
Ogni nodo è costituito da un **record** contenente:

- **Key:** informazioni pertinenti al nodo stesso
- **Left:** il puntatore al figlio sinistro (o **null** se non ha figlio sinistro)
- **Right:** il puntatore al figlio destro (o **null**)

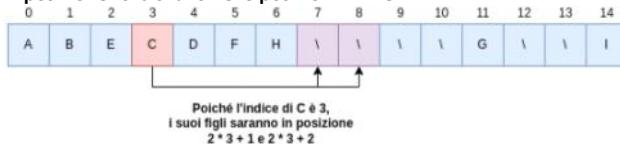
Similmente alla testa delle liste, si accede all'albero tramite il **puntatore alla radice**

**Vantaggi:** tutti i vantaggi delle strutture dinamiche basate su puntatori (inserimento nuovi nodi, spostamento, ecc)

**Svantaggi:** l'unico modo per accedere ad un nodo è scendere verso di esso partendo dalla radice, spostandosi di padre in figlio (non sapendo se sinistro o destro)



- **Tramite indici posizionali:** Come nella struttura **Heap**, i nodi sono memorizzati in un **array**, nel quale la **radice** è nella posizione 0 e i **figli sinistro e destro del nodo in posizione  $i$**  si trovano nelle **posizioni  $2i+1$  e  $2i+2$**



**Vantaggi:** Più veloce l'accesso ai nodi

**Svantaggi:**

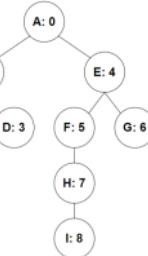
- Richiede di conoscere in anticipo l'**altezza massima** (è un array **statico**)
- Bisogna allocare un array in grado di contenere l'intero albero
- Se l'albero non è abbastanza "denso" vi è uno spreco di memoria

- **Tramite vettore dei padri:**

Vengono usati **due array**: uno con i nodi dell'albero A e uno con il padre di ciascun nodo P

L'elem. A[i] conterrà il **val. di un nodo**, mentre l'elem. P[i] conterrà l'**indice del padre del nodo  $i$  nell'albero**.

Questo metodo funziona senza alcuna modifica anche per gli **alberi n-ari**, in cui ogni nodo può avere un **numero qualunque di figli**



## Confronto tra rappresentazioni

- **Trovare il padre di un nodo**

- **Puntatori:** non siamo in grado di farlo senza implementare un algoritmo complesso (**visita dell'albero**)
- **Posizionale:** il padre del nodo  $i$  è in pos  $\left\lfloor \frac{i-1}{2} \right\rfloor$ :  $\Theta(1)$
- **Vettore:** l'indice del padre del nodo  $i$  è memorizzato in  $P[i]$ :  $\Theta(1)$

- **Determinare se un nodo abbia 0, 1 o 2 figli**

- **Puntatori:** controllo se i campi **left** e **right** sono settati a **None**:  $\Theta(1)$
- **Posizionale:** controllo se gli elementi  $2i+1$  e  $2i+2$  sono settati a **'-'**:  $\Theta(1)$
- **Vettore:** scorro l'array P e si conta il num. di occorenze di  $i$ :  $\Theta(n)$

- **Determinare la distanza di un nodo dalla radice**

- **Puntatori:** non si può fare senza implementare un algoritmo complesso (**visita dell'albero**)
- **Posizionale:** il liv. del nodo  $i$  (e quindi la sua distanza dalla radice) è  $\lceil \log(i+1) \rceil$ :  $\Theta(1)$
- **Vettore:** a partire da i risaliamo di padre in padre passando per  $P[i], P[P[i]], P[P[P[i]]], \dots$  fino alla radice:  $\Theta(h)$

# Visite di un albero

martedì 5 novembre 2024 14:23

Un'operazione comune degli alberi è l'**accesso progressivo a tutti i nodi**, uno dopo l'altro, al fine di poter effettuare una specifica operazione su ciascuno di essi. Rispetto alle liste su cui basta una semplice iterazione, sugli alberi l'operazione è **più complessa**.

L'**accesso progressivo** a tutti i nodi dell'albero si chiama **visita dell'albero**.

In base all'**ordine** col quale si accede ai nodi, esiste **più di una possibilità** a seconda del momento in cui si svolge l'**operazione richiesta** sul nodo stesso.

Negli **alberi binari**, abbiamo **tre tipi di visite**:

- **Visita in preordine:** si accede al nodo **prima** di proseguire la vista nei suoi sottoalberi

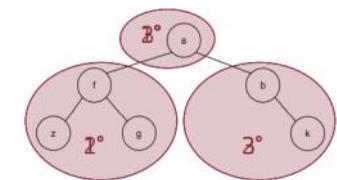
```
def visita_pre_order(p):
    if (p != None)
        fun(p) #accesso al nodo e operazioni consequenti
        visita_pre_order(p->left)
        visita_pre_order(p->right)
    return
```

- **Visita inordine:** il nodo è visitato **prima** della visita al **sottoalbero destro** e **dopo** la visita del **sottoalbero sinistro**

```
def visita_in_order(p):
    if (p != None)
        visita_in_order(p->left)
        fun(p) #accesso al nodo e operazioni consequenti
        visita_in_order(p->right)
    return
```

- **Visita postordine:** il nodo è visitato **dopo** entrambe le visite dei sottoalberi

```
def visita_post_order(p):
    if (p != None)
        visita_post_order(p->left)
        visita_post_order(p->right)
        fun(p) #accesso al nodo e operazioni consequenti
    return
```



• visita in preordine: a f z g b k  
• visita in inordine: z f g a b k  
• visita in postordine: z g f k b a

## Costo computazionale

Il **costo delle tre visite** è lo stesso e varia in base alla struttura dati usata per memorizzare l'albero.

Nel caso di record e puntatori, detto **k** il **num. di nodi del sottoalbero sinistro della radice**, e **n-k-1** il **num. di nodi di quello destro**; l'eq. è:

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

Analizziamo il caso migliore e peggiore:

- **Caso migliore:** è un **albero completo** (ogni nodo possiede esattamente due figli). Se **h** è il num. dei livelli, si ha  $n = 2^{h+1} - 1$  nodi:

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

Poiché ricadiamo nel **caso 1** del **metodo principale**, il costo è  $T(n) = \Theta(n^{\log_2(2)}) = \Theta(n)$

- **Caso peggiore:** è un **albero estremamente sbilanciato**, tutti i nodi discendenti della radice sono agglomerati nel sottoalbero sinistro, dunque quando  $n-k-1 = 0$ , o nel sottoalbero destro, dunque  $k = 0$ :

- Se  $k = 0$  si ha  $n-k-1 = n-1$ :

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

- Se  $n-k-1 = 0$ , si ha  $k = n-1$ :

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

In entrambi i casi, usando il **metodo iterativo**, si ha che il costo è  $T(n) = n * \Theta(1) = \Theta(n)$

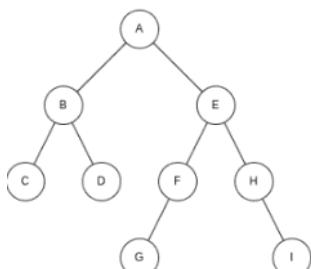
- **Caso generale:** poiché si ha che in entrambi i casi è  $\Theta(n)$ , possiamo ipotizzare che il suo costo sia  $\Theta(n)$ . Per confermare che è così dobbiamo usare il metodo della sostituzione.

Difatti, poiché una visita consiste nell'**analizzare tutti i nodi di un albero** e poiché  $n$  corrisponde al **num. di nodi totali**, viene logico che il **costo di una visita sia sempre  $\Theta(n)$**

In fondo alla pagina ho inserito anche la risoluzione con il metodo della **sostituzione**.

## Applicazioni delle Visite di Alberi

- 1) Consideriamo l'albero:



Immaginiamo di voler stampare tutti i nodi di questo albero. A seconda della **tipologia di visita** otteniamo un **output diverso**:

- **Preordine:** A-B-C-D-E-F-G-H-I
- **Inordine:** C-B-D-A-G-F-E-H-I
- **Postordine:** C-D-B-G-F-I-H-E-A

- 2) Vogliamo contare il **num. di nodi** di un albero binario.

Poiché il num di nodi di un sottoalbero corrisponde al **num. di nodi dei due sottoalberi sinistro e destro sommati alla radice stessa del sottoalbero**, è prima

necessario chiamare ricorsivamente la funzione su entrambi i due sottoalberi, per poi effettuare la somma.

Il risultato è l'applicazione di una **visita postordine**:

```
def Calcola_n(p):
    if p != None:
        num_l = Calcola_n(p.left) # nodo sinistro
        num_r = Calcola_n(p.right) # nodo destro
        num = num_l + num_r + 1 # accesso al nodo
    return num
return 0
```

compresso

```
def Calcola_n(p):
    if p != None:
        return Calcola_n(p.left) + Calcola_n(p.right) + 1
    return 0
```

- 3) Vogliamo **ricercare** un **nodo** in un albero binario

Restituisce **True** se k è tra le chiavi nell'albero, **False** altrimenti.

Immaginando di star effettuando la ricerca, viene logico pensare che **prima** di controllare i nodi successivi, venga controllato il **nodo attuale**.

Il risultato è l'applicazione di una **visita preordine**:

```
def Cerca(p, k):
    if p != None:
        if p.info == k: return True
        else:
            if Cerca(p.left, k) == True:
                return True
            else:
                return Cerca(p.right, k)
    return False
```

- 4) Vogliamo calcolare l'altezza di un albero binario

Poiché l'altezza di un sottoalbero corrisponde all'**altezza dei suoi sotto-alberi incrementata di uno**, ne deduciamo che sia necessario chiamare ricorsivamente la funzione prima di effettuare l'incremento.

Il risultato è l'applicazione di una **visita postordine**:

```
def Calcola_h(p):
    if p == None:
        return -1
    if p.left == None and p.right == None:
        return 0
    h = max(Calcola_h(p.left), Calcola_h(p.right)) + 1
    return h
```

compresso

```
def Calcola_h(p):
    if p != None:
        return max(Calcola_h(p.left), Calcola_h(p.right)) + 1
    return -1
```

## Costo Visite tramite Sostituzione

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

Tramite i due casi visti prima di O(n) e  $\Omega(n)$  abbiamo visto che il costo medio è  $\Theta(n)$

Proviamo a risolvere l'equazione con il **metodo della sostituzione**:

- Eliminiamo la notazione asintotica:
 
$$T(n) = \begin{cases} T(n) = T(k) + T(n - k - 1) + c \\ T(1) = d \end{cases}$$
- Tentiamo la sol.  $T(n) \leq a^*n$  per qualche costante a da determinare
  - Sostituendo **caso base T(1)**:  
 $d \leq a^*1$
  - Sostituendo **caso generico T(n)**:  
 $T(k) + T(n - k - 1) + c \leq a^*k + a^*(n - k - 1) + c = a^*k + a^*n - a^*k - a^*1 + c = a^*n - a + c \leq an$  se e solo se  $c \leq a$   
 Poiché entrambe le diseguaglianze ( $d \leq a$  e  $c \leq a$ ) sono **ammisibili**, deduciamo che  $T(n) = O(n)$
- Tentiamo la sol  $T(n) \geq b^*n$  per qualche costante b da determinare
  - Sostituendo **caso base T(1)**:  
 $d \geq b^*1$
  - Sostituendo **caso generico T(n)**:  
 $T(k) + T(n - k - 1) + c \geq b^*k + b^*(n - k - 1) + c = b^*n - b + c \geq b^*n$  se e solo se  $c \geq b$   
 Poiché entrambe le diseguaglianze ( $d \geq b$  e  $c \geq b$ ) sono **ammisibili**, deduciamo che  $T(n) = \Omega(n)$
- Quindi concludiamo che  $T(n) = \Theta(n)$

# Alberi binari di ricerca

martedì 5 novembre 2024 15:00

Un **albero binario di ricerca (ABR)** è una versione dell'**albero binario** nel quale vengono **mantenute** le seguenti **proprietà aggiuntive**:

- Ogni nodo **contiene una chiave x**
- il val. della chiave contenuta in ogni nodo è **maggior**e della chiave contenuta in **ciascun nodo del suo sottoalbero sinistro (se esiste)**.  
Dunque, il **val. minimo assoluto** dell'albero è il nodo **più a sinistra**
- il val. della chiave contenuta in ogni nodo è **minore** della chiave contenuta in **ciascun nodo del suo sottoalbero destro (se esiste)**.  
Dunque, il **val. massimo assoluto** dell'albero è il nodo **più a destra**.

Non è detto che il minimo e il massimo siano foglie dell'albero (ad es. può essere anche un nodo più a sinistra che non ha figlio sinistro)

## Ordinamento dei nodi

Se volessimo stampare i nodi di un ABR in **ordine crescente**, ricordando che le **proprietà degli ABR**, i val. minori di un nodo sono nel sottoalbero sinistro e i val. maggiori sono nel sottoalbero destro, possiamo applicare una **visita inordine** per stampare l'intero albero in ordine:

```
def Stampa_crescente(p):
    if (p != None)
        Stampa_crescente(p->left)
        print(p->val)
        Stampa_crescente(p->right)
```

Potremmo considerare l'ABR come un **algoritmo di ordinamento**, costituito dalla costruzione dell'ABR e dalla sua visita inordine ( $\Theta(n)$ )

## Ricerca di un nodo

La **ricerca su un ABR** è simile alla **ricerca binaria**: si esegue una discesa dalla radice che viene guidata da un **confronto (maggior o minore) tra i val. dei nodi** che si incontrano nel cammino:

```
def ABR_search_ricorsiva(p, k):
    if p == None or p.key == k:
        return p
    if k < p.key:
        return ABR_search_ricorsiva(p.left, k)
    else:
        return ABR_search_ricorsiva(p.right, k)
```

```
def ABR_search_iterativa(p, k):
    while p != None and p.key != k:
        if k < p.key:
            p = p.left
        else:
            p = p.right
    return p
```

- Nella ricerca **ricorsiva** compie un num. costante di op. **sulla radice** e poi esegue la **ricerca su uno dei due sottoalberi**, che al **massimo ha altezza h-1**, mentre nel **caso base**, che si ha quando **h = 0** (la radice è l'elem da trovare), il tempo di esecuzione è  $T(0) = \Theta(1)$ , quindi possiamo scrivere:

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

Col **metodo iterativo** vediamo che  $T(h) = \Theta(h)$

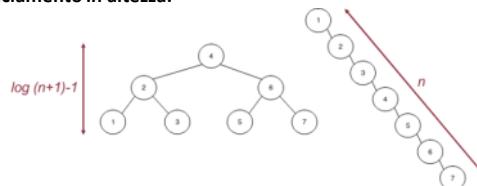
- Nella ricerca **iterativa**, il costo dipende dal num. di volte che viene **eseguito il ciclo while**.

Nel **caso peggiore** (ricerca senza successo) il ciclo viene eseguito  $h$  volte, quindi il costo è, come per la versione ricorsiva,  $O(h)$ .

Tuttavia, a differenza della ricerca binaria, **non riusciamo a garantire un costo computazionale di  $O(\log(n))$** , poiché il costo di ricerca è  $O(h)$ , dove  $h$  è l'altezza dell'ABR:

- **Caso peggiore**: albero pienamente sbilanciato (tutti i nodi agglomerati a sinistra/destra), l'altezza corrisponde a  $n-1$ :  $O(n)$
- **Caso migliore**: albero completo, l'altezza corrisponde a  $\log(n+1) - 1$ :  $\Omega(\log(n))$

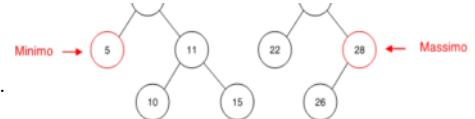
Per garantire che il costo della ricerca sull'ABR sia  $O(\log(n))$ , è necessario mettere usare qualche tecnica per tenere sotto controllo la crescita dell'altezza, ossia il **bilanciamento in altezza**.



## Ricerca di minimo, massimo, predecessore, successore

Il **massimo** e **minimo**, si trovano agli **estremi destri e sinistri** dell'albero, basta quindi **scendere** sempre a destra/sinistra finché il nodo non ha più un figlio destro/sinistro

Entrambe le op. richiedono di scendere dalla radice su un unico cammino, quindi il costo è sempre  $O(h)$ .



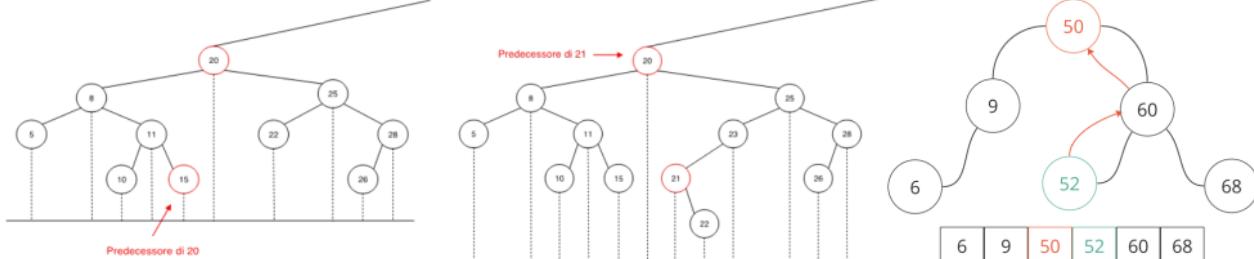
Per **predecessore (successore)** di una chiave  $k$  si intende il nodo contenente la chiave che **precede (segue)  $k$**  se le chiavi fossero ordinate.

Si possono verificare **2 casi**:

- 1: se il nodo con chiave  $k$  ha **sottoalbero sinistro (destro)**, allora il **predecessore (successore)** è il **massimo (minimo)** del sottoalbero **sinistro (destro)**.
- 2: se il nodo con chiave  $k$  **non ha sottoalbero sinistro (destro)**, allora lui stesso è il **nodo più a sinistra (destra)** del suo sottoalbero.

Per trovare il suo **predecessore (successore)** bisogna **risalire verso destra (sinistra)** fino alla **radice del sottoalbero**, il cui padre è il **predecessore (successore)** di  $k$ .

Entrambe le op. il costo è  $O(h)$ .



Per controllare se stiamo **risalendo verso destra o sinistra** basta vedere **se il nodo** che si sta controllando **è uguale al figlio destro o sinistro del proprio padre**.

- if  $x == x.parent.left$  se tale condizione è vera, è il figlio sinistro
- if  $x == x.parent.right$  se tale condizione è vera, è il figlio destro

## Eliminazione di un nodo

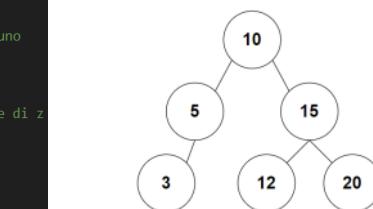
L'**eliminazione di un nodo** è il problema più complicato. Si possono verificare **3 casi**:

- 1) Se il nodo è una **foglia**, lo si elimina ponendo **None** nel suo **nodo padre**
- 2) Se il nodo ha **un solo figlio**, lo si **"cortocircuita"** collegando **suopadre** e **il suo unico figlio**
- 3) Se il nodo ha **entrambi i figli**, è necessario **riaggiustare l'albero**, trovando un **nodo** da collocare al **posto del nodo** che va **eliminato**, così da **mantenere l'albero connesso** e da **garantire il mantenimento delle proprietà dell'ABR**.

Per fare ciò si sostituisce col **predecessore (o successore)** (caso 1 di ricerca del predecessore), che va eliminato dalla sua pos. originale (oper. che ricade in uno dei due casi precedenti)

```
def ABR_delete(p, z):
    # CASO 1 o 2
    if z.left == None or z.right == None: # z non ha figli o ne ha uno
        y = z # z è il nodo da eliminare
    # CASO 3
    else: # z ha due figli
        y = ABR_successore(z) # il nodo da eliminare è il successore di z
    # Ora y punta al nodo da eliminare
    if y.left != None: # y ha un figlio SX
        x = y.left # x è il figlio SX di y
    else: # y non ha figli SX
        x = y.right # x è il figlio DX di y
    # y non può avere due figli, x punterà al figlio o sarà None
    # CASO 2: Se y ha un figlio, lo collegiamo al padre di y
    if x != None: # se x esiste, quindi y ha un figlio
        x.parent = y.parent # il padre di x diventa il padre di y

    if y.parent == None: # y è la radice?
        p = x # si: x diventa la radice
    else: # y non è la radice
        if y == y.parent.left: # se y era il figlio SX
            y.parent.left = x # x prende il posto di y
        else: # se y era il figlio DX
            y.parent.right = x # x prende il posto di y
    # CASO 3: Se y è il successore di z, copiamo la chiave di y in z
    if y != z: # y è il successore di z
        z.key = y.key # copia la chiave di y in z
    return p
```

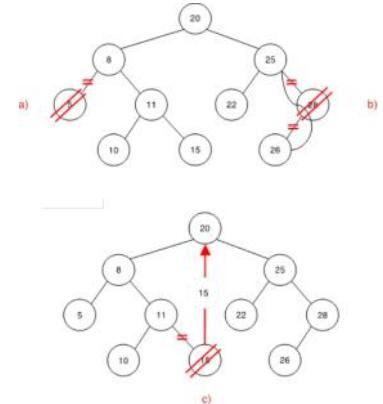


Esempio di **Cancellazione**:

- Caso 1: il nodo **12 non ha figli**, basta **rimuoverlo** e aggiornare il puntatore del padre (**15**) a **None**
- Caso 2: il nodo **5 ha un solo figlio**, **5 viene eliminato** e suo figlio (**3**) viene **collegato al padre di 5 (10)**
- Caso 3: il nodo **15 ha due figli**, si trova il **successore (20)**, si **copia 20 in 15** e si **elimina il nodo 20**.

Dove si trovano i casi nel codice?

- Caso 1: y viene semplicemente **scollegato** dall'ARB
- Caso 2: il nodo viene **"cortocircuitato"** sostituendolo col figlio
- Caso 3: si **elimina il successore** e si **copia la sua chiave** nel nodo da eliminare



La funzione effettua un num. costante di op. e una chiamata alla **ricerca del successore**, il costo è:  $\Theta(1) + \Theta(h) = \Theta(h)$

# Alberi rosso-neri

martedì 5 novembre 2024 15:40

(non mi andava di aggiungerli perché nell'anno 24/25 non li ha fatti e negli scorsi esami sono capitati 3 volte (pg. 204 dispense del Monti))

# Dizionari

martedì 5 novembre 2024 15:41

Un **dizionario** è una struttura dati che permette di gestire un insieme dinamico di dati, che di norma è un **insieme totalmente ordinato**, tramite queste tre sole operazioni:

- **Insert**: inserire un elemento
- **Search**: ricerca un elemento
- **Delete**: elimina un elemento

Per realizzare un dizionario si ricorre a soluzioni specifiche:

- **Tabelle ad indirizzamento diretto**
- **Tabelle hash**
- **Alberi binari di ricerca**

Prima di è necessario introdurre delle **assunzioni** e delle **nomenclature**:

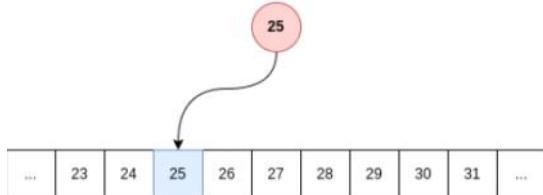
- **Insieme U**: è l'insieme universo dei **val.** che le **chiavi possono assumere** ed è costituito da **val. interi**
- **Valore m**: è il **num. di posizioni a disposizione** nella struttura
- **Valore n**: è il **num. di elem. da memorizzare nel dizionario** e i val. delle chiavi degli elem. da memorizzare sono tutti diversi tra loro

## Tabelle ad indirizzamento diretto

martedì 5 novembre 2024 15:52

Un dizionario costituito da una **tabella ad indirizzamento diretto**, è un vettore nel quale **ogni indice corrisponde al val. della chiave** dell'elem. da memorizzare in tale pos.

L'elemento con chiave  
25 verrà inserito nella  
posizione di indice 25



Affinché tale dizionario funzioni, è **necessario che  $n \leq m = |U|$** , dove  $|U|$  indica il num. di elem. dell'insieme  $U$ , assolvendo al compito di dizionario con **grande efficienza**. Tutte e tre le oper. hanno costo pari a  $\Theta(1)$ :

```
def Insert_Indirizz_Diretto (A; e: elem da ins.) def Search_Indirizz_Diretto (A; k: chiave da cerc.) def Delete_Indirizz_Diretto (A; k: chiave da canc.)  
    A[e->key] = e                                return A[k]  
    return                                         A[k] = None  
                                                 return
```

Sebbene sia molto efficiente, l'indirizzamento diretto risulta essere il meno efficiente per quanto riguarda il **costo in memoria**:

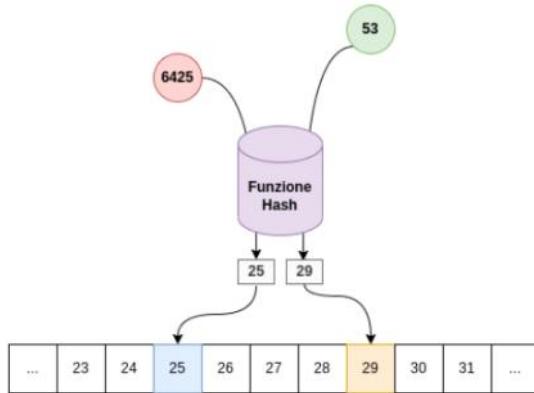
- Poiché si ha  $m = |U|$ , un insieme  $U$  enorme rende impraticabile l'**allocazione in memoria** di un array lungo  $m$
- Il num. di chiavi effettivamente usate può essere molto più piccolo di  $|U|$ : in tal caso vi è un **rilevante spreco di memoria**

Perciò si ricorre spesso a differenti implementazioni dei dizionari

# Tabelle Hash

martedì 5 novembre 2024 16:03

Per risolvere il problema degli indirizzamenti diretti, si ricorre all'uso di una **funzione di hash**, in grado di calcolare la pos. finale di un elem. sulla base del val. della sua chiave, il quale può essere un **intero molto più grande del val. generato dalla funzione di hash**.



Tuttavia, anche se le chiavi da memorizzare sono meno di  $m$ , non si può escludere che due chiavi  $k_1 \neq k_2$ , siano tali per cui  $\text{hash}(k_1) = \text{hash}(k_2)$ , ossia che la **funzione hash restituisca lo stesso val. per entrambe le chiavi**, che quindi andrebbero memorizzate nella stessa pos. della tabella, dando vita a quello che viene definito come **collisione di hash**.

Tale fenomeno è **matematicamente inevitabile**. Dunque possiamo solo tentare di evitarlo il più possibile con alcune strategie:

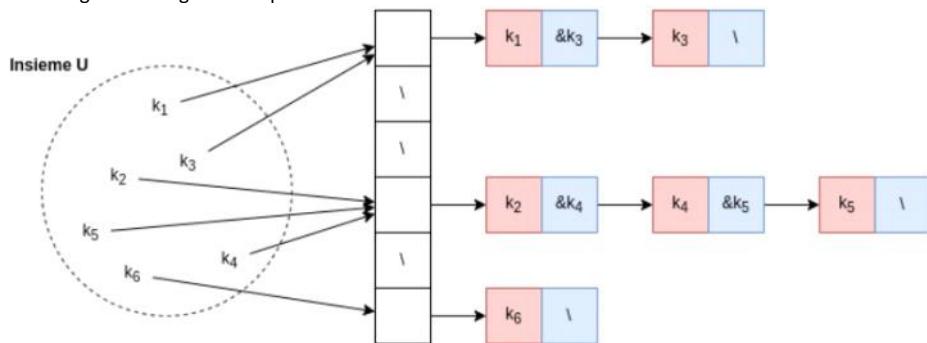
- Una buona funzione hash deve essere tale da rendere il **più equiprobabile possibile il val. risultante** dall'applicazione della funzione: tutti i val. fra 0 e  $m - 1$  dovrebbero essere restituiti con uguale probabilità.  
In altre parole, la funz. deve far apparire come "casuale" il val. risultante, disregando qualunque regolarità della chiave.
- La funzione deve essere **deterministica**: se applicata più volte alla stessa chiave deve fornire lo stesso risultato
- La situazione ideale è quella in cui ciascuna delle  $m$  pos. della tabella è scelta deterministicamente con la stessa probabilità: **ipotesi di uniformità semplice della funzione hash**

L'**ipotesi di uniformità semplice** minimizza il num. di collisioni, ma queste possono comunque avvenire; è **impossibile evitare del tutto le collisioni**, poiché siccome si ha che  $|U| > m$ , è inevitabile che esistano chiavi diverse che producono una collisione

# Tabelle a lista di trabocco

martedì 5 novembre 2024 16:22

Prevede di inserire **tutti gli elem.** le cui chiavi mappano la stessa posizione in una lista puntata, detta **lista di trabocco**. Invece degli elementi in se nell'array del dizionario vengono immagazzinati i puntatori alle teste delle liste di trabocco.



Poiché viene implementato tramite l'uso di liste puntate, il costo delle oper. è **strettamente legato ad esse**:

- **Inserimento:** il costo rimane  $\Theta(1)$  (effettuiamo l'inserimento in testa alle liste)

```
def Insert_Liste_Trabocco (A; e: elem. da ins.):
    lista_di_trabocco = A[hash(e->key)]
    insert_in_testa(lista_di_trabocco, e)
    return
```

- **Ricerca:** la ricerca in una lista puntata ha costo pari a  $O(i)$  dove  $i$  è la **lunghezza della lista**; quindi il costo della ricerca in un dizionario così nel **caso peggiore** è  $O(n)$ , dovuto all'inserimento di tutti gli elem. nella stessa lista di trabocco.

Nel **caso medio**, il costo risulta essere  $O(\alpha)$ , dove  $\alpha$  è il **fattore di carico della tabella**. Considerando l'**ipotesi di uniformità semplice**, otteniamo che  $\alpha = n/m$ , dunque il costo della ricerca è  $O(n/m)$

```
def Search_ListeTrabocco (A; k: chiave da cercare):
    lista_di_trabocco = A[hash(k)]
    elem = ricerca_elem(lista_di_trabocco, k)
    return elem
```

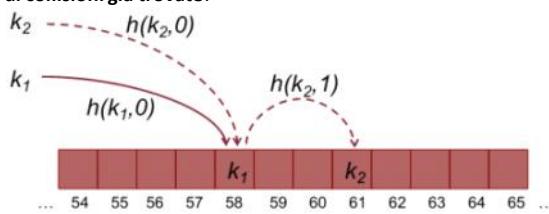
- **Eliminazione:** dipende dall'implementazione delle liste di trabocco e valgono, **tutte le osservazioni fatte** per il costo dell'oper. di cancellazione nelle liste concatenate. Con liste semplici il costo è lo stesso di quello della ricerca

```
def Delete_ListeTrabocco (A; k: chiave da canc.):
    lista_di_trabocco = A[hash(k)]
    elimina_elem(lista_di_trabocco, k)
    return
```

## Tabella ad indirizzamento aperto

martedì 5 novembre 2024 16:29

Prevede l'inserimento solo all'interno dell'array stesso, calcolando la **sequenza delle posizioni da esaminare**. La funzione hash dipende da 2 parametri: la chiave  $k$  e il **num. di collisioni già trovate**.



Questa tecnica è applicabile solo quando:

- $m \geq n$  (il fattore di carico  $\alpha = n/m$  è sempre  $\leq 1$ )
- $|U| \gg m$  (il num. di elem. dell'insieme  $U$  è molto più grande di  $m$ )

### Operazioni sul dizionario

- **Inserimento:** se la pos. calcolata con  $h(k, 0)$  (dove 0 è il num. di collisioni generate durante l'inserimento attuale) è **già occupata**, allora viene **ricalcolato l'hash** incrementando di 1 il num. di collisioni (dunque  $h(k, 1)$ ), generando un val. completamente diverso dal precedente. Se anche l'a nuova pos. è già occupata, verrà ripetuto il processo, **fino ad un massimo di  $h(k, m-1)$**   
Nel **caso peggiore**, il costo di tale oper. è  $O(n)$  e si verifica quando tutti gli  $n$  elem. inseriti restituiscono lo stesso  $h(k, 0)$ . Considerando l'ipotesi di uniformità semplice, il **caso medio** risulta essere  $O(1/(1-\alpha)) = O(m/(m-n))$
- **Ricerca:** si analizza la tabella mediante la stessa **sequenza di funzioni hash** usate per l'inserimento fino a quando si incontra l'elem. oppure una casella vuota, deducendo che l'elem. non è presente. Ne segue che i suoi costi computazionali sono analoghi a quelli dell'inserimento
- **Eliminazione:** tale oper. risulta essere molto problematica, poiché nel caso in cui si vada ad eliminare l'elem. lasciando la casella vuota, ciò **spezzerebbe la catena di sequenza della funzione hash** di un qualunque elem. già inserito nella tabella  
Una soluzione sarebbe quella di eliminare un elem. marcandolo con un val. **deleted**, tuttavia il costo computazionale della ricerca e dell'inserimento non più dal fattore di carico ma **anche dal num. di pos. marcate**  
Per queste ragioni, la cancellazione **non è supportata con l'indirizzamento aperto**.

Garantire l'**ipotesi di uniformità semplice** risulta essere quasi impossibile, tuttavia alcune tecniche si avvicinano molto ad essa, in particolare l'**hashing doppio**.

L'idea è di usare **due diverse funzioni di hash**, una per l'**accesso iniziale** alla tabella e l'altra per il **passo di scansione** (ossia la ripetizione con l'incremento in base alla collisione):

$$h(k, i) = [h_1(k) + i * h_2(k)] \bmod m \text{ dove } i \text{ appartiene a } [0, m-1]$$

Se le due funz. sono ben progettate, è **estremamente improbabile che due chiavi  $k_1 \neq k_2$  producano una collisione su entrambe le funzioni hash**.

# Argomenti per gli esami

martedì 28 gennaio 2025 14:56

Questi sono gli argomenti necessari da conoscere per superare gli esami, basati sugli ultimi 4 esami prima di febbraio 2025 (Esami del 21/1/25, 24/10/24, 18/9/24, 10/7/24)

## Cose importanti:

- Tecniche di risoluzione delle ricorrenze, in particolare Teorema di Master
- Strutture dati fondamentali come Alberi binari, liste concatenate e array
- Pratica scrittura algoritmi in pseudocodice
- Analizzare costi computazionali degli algoritmi

## 1. Ricorrenze e complessità algoritmica

- Teorema Principale:
  - Applicazione ai diversi casi (dominanza di funzioni nella ricorrenza: caso 1, 2, 3).
  - Analisi dettagliata delle condizioni del teorema e dimostrazioni.
- Metodi per risolvere ricorrenze:
  - Metodo iterativo (unwrapping).
  - Uso delle sommatorie e manipolazioni algebriche.
  - Equazioni di ricorrenza particolari (esempi da esercizi:  $T(n) = 2T(n/2) + \sqrt{n}$ ,  $T(n) = 4T(n/2) + \Theta(n^2)$ ).
- Notazione asintotica:
  - $\Theta, O, \Omega$ : definizioni, significato e confronto.
  - Manipolazione di funzioni polinomiali, logaritmiche e radici.

## 2. Algoritmi di ordinamento

- MergeSort:
  - Descrizione e struttura dell'algoritmo.
  - Equazione di ricorrenza classica  $T(n) = 2T(n/2) + \Theta(n)$ .
  - Varianti come NEW-MERGESORT (con fusione modificata a costo  $\Theta(\sqrt{n})$ ).
- Counting Sort:
  - Uso in contesti con dominio limitato.
  - Vantaggi in termini di complessità lineare.
- Heap Sort:
  - Ordinamento con struttura heap.
  - Analisi del costo computazionale.

## 3. Matrici e array

- Algoritmi su matrici:
  - Analisi di matrici simmetriche e operazioni specifiche (verifica proprietà come "conoscenza" tra elementi).
- Ricerca di coppie in array:
  - Strategie efficienti per trovare duplicati e soddisfare vincoli.
  - Ordinamento seguito da controlli (caso di età e matrice di conoscenze).

## 4. Alberi binari

- Cammini in alberi binari:
  - Analisi dei costi di cammini radice-foglia.
  - Visite in post-ordine per raccogliere informazioni sui sottoalberi.
  - Implementazione ricorsiva senza uso di variabili globali.
- Calcolo della complessità:
  - Relazioni di ricorrenza per alberi binari, come  $T(n) = T(h) + T(n-1-h) + \Theta(1)$ .

## 5. Liste concatenate

- Manipolazione di liste concatenate:
  - Filtraggio basato su condizioni (nodi pari o dispari).
  - Uso della ricorsione per modifiche senza array di appoggio.

- **Analisi del costo:**
  - Ricorrenza  $T(n) = T(n-1) + \Theta(1)$  e risoluzione.

## 6. Algoritmi iterativi

- **Ricerca di sequenze crescenti in array:**
  - Uso di due indici (inizio e massimo) per trovare la lunghezza massima.
  - Complessità  $O(n)$  con spazio aggiuntivo  $\Theta(1)$ .
- **Struttura di cicli nidificati:**
  - Analisi del costo per cicli for o while.

## 7. Analisi del costo computazionale

- **Distinzione tra migliori, peggiori e casi medi.**
- **Dimostrazione rigorosa della complessità:**
  - Contributo di ogni istruzione.
  - Analisi del costo totale.

## 8. Pseudocodice e implementazione

- **Pseudocodice commentato:**
  - Descrizione delle operazioni fondamentali.
  - Traduzione in linguaggi come Python.
- **Efficienza degli algoritmi:**
  - Valutazione delle soluzioni migliori in base alla complessità asintotica.

## 9. Altri concetti chiave

- **Gestione degli input limitati** (età minima e massima).
- **Ottimizzazione della memoria:**
  - Minimizzazione dell'uso di array aggiuntivi.
- **Simmetria nelle strutture dati:**
  - Proprietà delle matrici simmetriche e loro utilizzo.

# Costo Computazionale - Esercizi per Casa

martedì 8 aprile 2025 18:32

Calcolare il costo del seguente algoritmo scritto in pseudocodice, distinguendo tra caso migliore e caso peggiore se necessario:

Es 1	Es 2	Es 3
<pre>def Insertion_Sort(A):     for j in range (1,len(A)):         x = A[j]         i = j - 1         while (i&gt;=0) and (A[i]&gt;x):             A[i+1] = A[i]             i=i-1         A[i+1]=x     • Il for itera da 1 a n, costo Θ(n)     • Il while itera a ritroso partendo dalla pos.       precedente a j fino a che non trova un elem.       minore di quello corrente tra quelli prima di       quello precedente o finché non finisce l'array.       ○ Nel caso peggiore, potrebbe scorrere           tutto l'array precedente a j poiché non           c'è un elemento minore O(n)       ○ Nel caso migliore l'elemento precedente           è già minore e esce subito dal while Ω(1)     • Le altre op. hanno costo costante Θ(1) Quindi il costo si divide in:     • Peggiore: Θ(n)*O(n) = O(n²)     • Migliore: Θ(n)*Ω(1) = Ω(n)</pre>	<pre>def Selection_Sort(A):     for i in range(len(A)-1):         m=i         for j in range(i+1,len(A)):             if A[j] &lt; A[m]:                 m = j         A[m],A[i]=A[i],A[m]     • Il primo for itera da 0 fino a n-1, costo Θ(n)     • il secondo for itera dalla posizione successiva a       quella corrente (i+1) fino a n     • Le altre operazioni hanno costo costante Θ(1) T(n) = (Σ<sub>i=0</sub><sup>n-1</sup> Θ(1) + (n - i)Θ(1) + Θ(1)) + Θ(1) = Σ<sub>i=0</sub><sup>n-1</sup> i * Θ(1) + Θ(1) = Θ(n<sup>2</sup>) + Θ(n) + Θ(1) = Θ(n<sup>2</sup>)</pre>	<pre>def Bubble_Sort(A)     for i in range(len(A)-1):         for j in range(len(A)-i-1):             if (A[j] &gt; A[j+1]):                 A[j],A[j+1]=A[j+1],A[j]     • Il primo for itera da 0 a n-1     • Il secondo for itera da 0 a n-i-1     • Le altre op. hanno costo costante Θ(1) T(n) = (Σ<sub>i=0</sub><sup>n-1</sup>(n - i) * Θ(1) + Θ(1)) + Θ(1) = (Σ<sub>i=0</sub><sup>n-1</sup> i * Θ(1) + Θ(1)) + Θ(1) = Θ(n<sup>2</sup>) + Θ(n) + Θ(1) = Θ(n<sup>2</sup>)</pre>

## Altri Esercizi

### Es 0

```
def es0(n):
    t=0
    n=abs(n)
    if n>100: return 1
    for i in range(n):
        t+=3
    return t
```

- Se  $n > 100$  allora esegue 4 op. di costo costante e esce dalla funzione. Costo  $\Theta(1)$
- Altrimenti se  $n \leq 100$ , itera il for da 0 a  $n-1$ , però siccome  $n \leq 100$ , itera al massimo 100 volte che è un num. costante, quindi ha costo  $\Theta(1)$
- Tutte le altre op. hanno costo costante  $\Theta(1)$

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

### Es 1

```
def es1(n):
    if n<0: n=-n
    while n:
        if n%2: return 1
        n-=2
    return 0
```

- Se  $n$  è negativa allora la rende positiva in costo  $\Theta(1)$
- Esegue un while finché  $n$  è minore di 0.
  - Ad ogni iterazione controlla se  $n$  è dispari e in quel caso termina la funzione
  - E poi diminuisce di 2

Se partiamo con  $n$ :

- Pari, allora appena entra nel while termina la funzione senza dover iterare il while più volte, quindi si ha costo costante.  $T(n) = \Omega(1)$
- Dispari, allora ad ogni iterazione del while rimuove 2 lasciando sempre  $n$  dispari, quindi itera il while finché  $n$  non diventa negativo, quindi  $T(n) = O(n)$

### Es 2

```

def es2(n):
    n=abs(n)
    x=r=0
    while x*x<n:
        x+=1
        r*=3*x
    return r

```

- Itera il while da 0 a  $\sqrt{n}$  (poiché ad ogni iterazione aumenta x di 1, e finisce quando  $x^2 \geq n \rightarrow x \geq \sqrt{n}$ )
- Il resto delle op. ha costo costante  $\Theta(1)$

$$T(n) = \Theta(\sqrt{n})$$

**Es 3**

```

def es3(n):
    n=abs(n)
    x=r=0
    while n>1:
        r+=2
        n=n/3
    return r

```

- Itera il while da n a  $\log_3(n)$  (poiché ad ogni iterazione dimezza di 3 n, e finisce quando  $\frac{n}{3^k} \leq 1 \rightarrow n \leq 3^k \rightarrow \log_3(n) \leq k$ )
- Il resto delle op. ha costo costante  $\Theta(1)$

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

**Es 4**

```

def es4(n):
    n=abs(n)
    x=t=1
    for i in range(n):
        t=3*t
    while t>=x:
        x+=2
        t-=2
    return x

```

- Il for itera da 0 a  $n-1 \rightarrow \Theta(n)$  e alla fine del for t ha val  $3^n$
- Il while itera da  $3^n$  a  $3^n/4$  (poiché ad ogni iterazione diminuisce di 2 il  $3^n$  e aumenta di 2 x, quindi entrambi dimezzano il num. di iterazioni)  $\rightarrow \Theta(3^n)$

$$T(n) = \Theta(n) + \Theta(3^n) + \Theta(1) = \Theta(3^n)$$

**Es 5**

```

def es5(n):
    n=abs(n)
    p=2
    while n>=p:
        p=p*p
    return p

```

p parte da 2 e ad ogni iterazione del while viene moltiplicato per se stesso, quindi viene elevato alla seconda. Quindi all'iterazione i avremmo  $p = (2^i)^2 = 2^{2i}$ . Il ciclo while itera da n fino a che n non viene raggiunto o superato da p. Quindi si ferma quando  $2^{2k} = n \rightarrow 2k = \log(n) \rightarrow k = \log(n)/2$ . Quindi viene iterato  $\Theta(\log(n))$

**Es 6**

```

def es6(n):
    n=abs(n)
    i,j,t,s=1
    while i*i<=n:
        for j in range(t)
            s+=1
        i=i+1
        t+=1
    return s

```

i e t partono da 1 e ad ogni iter. del while vengono incrementati entrambi di 1. Quindi all'iter. x avremmo i = x e t = x  
Il ciclo while termina quando  $i^2 > n \rightarrow i > \sqrt{n}$ , quindi viene iterato  $\sqrt{n}$  volte.

Il ciclo for invece itera da 0 a t e quindi da 0 a i.

$$T(n) = \Theta(1) + \sum_{i=0}^{\sqrt{n}} i\Theta(1) + \Theta(1) = \Theta(1) + \Theta\left(\frac{\sqrt{n}(\sqrt{n}+1)}{2}\right) + \Theta(\sqrt{n}) = \Theta(1) + \Theta\left(\frac{n+\sqrt{n}}{2}\right) + \Theta(\sqrt{n}) = \Theta(1) + \Theta(n) + \Theta(\sqrt{n}) = \Theta(n)$$

### Esempio 7

```

def es7(n):
    n=abs(n)
    t,s=n
    p=0
    while s>=1:
        s=s//4
        p+=1
    while n-s>0:
        n-=s
        t+=5
    return t

```

In realtà se s arriva a 1 e poi viene eseguito l'ultimo s//4  $\rightarrow 1//4 = 0$ , quindi il ciclo while dopo viene eseguito all'infinito, perché viene rimosso 0 da n.  
Quindi facciamo finta che s si ferma a 1.

il primo ciclo while itera da n a  $\log_4(n)$ , poiché ad ogni iterazione s viene divisa per 4, quindi s raggiunge 1 quando  $\frac{n}{4^k} = 1 \rightarrow n = 4^k \rightarrow k = \log_4(n)$ .  $\rightarrow \Theta(\log(n))$

A questo punto s sarà 1

Il secondo while itera n volte poiché ad ogni iter. viene rimosso s (1) e il controllo viene effettuato con n-s però s non incrementa di valore  $\rightarrow \Theta(n)$   
 $T(n) = \Theta(1) + \Theta(\log(n)) + \Theta(n) = \Theta(n)$

### Esempio 8

```

def es8(n):
    n=abs(n)
    t,s=n
    p=0
    while s>=1:
        s=s//4
        p+=1
    while n-p>0:
        n-=p
        t+=5
    return t

```

il primo ciclo itera  $\log_4(n)$ , poiché si ferma quando s (che inizia da n) raggiunge 0 e ad ogni iter. viene divisa per 4, quindi si ferma quando  $\frac{n}{4^k} = 1 \rightarrow n = 4^k \rightarrow k = \log_4(n)$ .  $\rightarrow \Theta(\log(n))$

alla fine avremmo che p ha come valore  $\log_4(n)$

Nel secondo ciclo, viene rimosso p ( $\log_4(n)$ ) da n ad ogni iterazione, quindi si ferma quando  $\frac{n}{\log_4(n)} = 0 \rightarrow \Theta(n/\log(n))$

$T(n) = \Theta(1) + \Theta(\log(n)) + \Theta(n/\log(n)) = \Theta(n/\log(n))$  (per n molto grande n/log(n) cresce più velocemente di log(n))

### Esempio 9

```

def es9(n):
n=abs(n)
s=n
p=2
i,r=1
while s>=1:
    s=s//5
    p+=2
p=p*p
while i*i*i<n:
    for j in range(p):
        r+=1
    i+=1
return r

```

il primo ciclo while viene iterato da n fino a 0, però ad ogni iter. s viene diviso per 5, quindi viene iterato  $\frac{n}{5^k} = 0 \rightarrow k = \log_5(n)$  volte  $\rightarrow \Theta(\log(n))$  alla fine del ciclo, p ha come valore circa  $2^{[\log_5(n)]+1}$  (il 2 iniziale), poiché ad ogni iterazione del ciclo incrementa di 2, quindi ha val,  $\Theta(\log(n))$  viene poi elevato alla seconda, quindi ha val  $\Theta(\log^2(n))$

il secondo ciclo while incrementa i di 1 ad ogni iter, e finisce quando  $i^3 = n \rightarrow i = \sqrt[3]{n} \rightarrow$  viene iterato  $\sqrt[3]{n}$  volte

il ciclo for annidato viene iterato da 0 a p, quindi viene iterato  $\Theta(\log^2(n))$  volte

$$T(n) = \Theta(1) + \Theta(\log(n)) + \sqrt[3]{n} * (\Theta(\log^2(n)) + \Theta(1)) = \Theta(\log(n)) + \Theta((\sqrt[3]{n}) * \log^2(n)) + \Theta(\sqrt[3]{n}) = \Theta((\sqrt[3]{n}) * \log^2(n))$$

### Esercizio 10

```

def es10(n):
tot=n
if n<=100: return 5
j=512
while j>=2:
    k=1
    while k*k<=n:
        k=2*k
    tot=tot+5
    j=j//2
return tot

```

il primo ciclo while parte da j=512 fino a 1, e ad ogni iter. j viene dimezzato, quindi si ferma quando  $\frac{512}{2^k} = 1 \rightarrow \log_2(512) = 9 \rightarrow$  viene iterato un num. costante di volte quindi  $\Theta(1)$

il secondo while invece viene iterato finché  $k^2$  supera n, e ad ogni iter. k viene raddoppiato quindi si ferma quando  $(2^k)^2 > n \rightarrow 2^{2k} > n \rightarrow 2k > \log_2(n) \rightarrow k > \frac{\log_2(n)}{2} \rightarrow$  viene iterato  $\Theta(\log(n))$  volte

$$T(n) = \Theta(1) + \Theta(1) * \Theta(\log(n)) + \Theta(1) = \Theta(\log(n))$$

### Soluzioni

Esercizio 0:  
Caso migliore=peggiore:  $\Theta(1)$

Esercizio 6:  
Caso migliore=peggiore:  $\Theta(n)$

Esercizio 1:  
Caso migliore:  $\Theta(1)$  (quando n è dispari)  
Caso peggiore:  $\Theta(n)$  (quando n è dispari)

Esercizio 7:  
Caso migliore=peggiore:  $\Theta(n)$

Esercizio 8:  
Caso migliore=peggiore:  $\Theta(n / \log n)$

Esercizio 2:  
Caso migliore=peggiore:  $\Theta(n^{1/2})$

Esercizio 9:  
Caso migliore=peggiore:  $\Theta(n^{1/3} \log^2 n)$

Esercizio 3:  
Caso migliore=peggiore:  $\Theta(\log n)$

Esercizio 10:  
Caso migliore=peggiore:  $\Theta(\log n)$

Esercizio 4:  
Caso migliore=peggiore:  $\Theta(3^n)$

Esercizio 5:  
Caso migliore=peggiore:  $\Theta(\log \log n)$

# Ricorsione - Esercizi per Casa

mercoledì 9 aprile 2025 11:40

## Es 1

Progettare una funzione ricorsiva che, dati due interi  $k$  ed  $n$ ,  $0 \leq k \leq n$ , calcoli il valore del coefficiente binomiale  $n$  su  $k$  utilizzando la seguente relazione:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} \text{ con } \binom{n}{0} = \binom{n}{n} = 1$$

```
def CoeffBin(n, k):
    if k == n or k == 0:
        return 1
    return CoeffBin(n-1, k) + CoeffBin(n-1, k-1)
```

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

$$T(n, 0) = T(n, n) = \Theta(1)$$

## Es 2

Progettare un algoritmo ricorsivo che, dati due interi  $x$  e  $y$ ,  $x>y>0$ , ne calcoli il massimo comun divisore utilizzando il seguente procedimento (di Euclide):

- se  $y=0$  allora  $\text{MCD}(x,y)=x$
- altrimenti  $\text{MCD}(x,y)=\text{MCD}(y,x\%y)$  dove  $x\%y$  rappresenta il resto della divisione tra  $x$  ed  $y$

```
def MCD(x, y):
    if y == 0:
        return x
    return MCD(y, x%y)
```

$$T(x, y) = T(y, x\%y) + \Theta(1)$$

$$T(x, 0) = \Theta(1)$$

## Eq. Ricorrenza - Esercizi per casa

mercoledì 9 aprile 2025 11:48

Calcolare la soluzione delle seguenti equazioni di ricorrenza con tutti e quattro i metodi ove possibile:

$$T(n) = 2T(n/2) + \Theta(n) \quad T(1) = \Theta(1)$$

Iterativo	Albero	Sostituzione	Principale
<p>Sviluppando:</p> <ol style="list-style-type: none"> <li>1) <math>2T(n/2) + \Theta(n)</math></li> <li>2) <math>4T(n/4) + 2\Theta(n/2) + \Theta(n)</math></li> <li>3) <math>8T(n/8) + 4\Theta(n/4) + 2\Theta(n/2) + \Theta(n)</math></li> </ol> <p>La ricorsione continua finché <math>\frac{n}{2^k} = 1 \rightarrow k = \log(n)</math>:</p> $T(n) = 2^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 2^i \Theta\left(\frac{n}{2^i}\right) =$ $2^{\log(n)} T(1) + \sum_{i=0}^{\log(n)-1} \Theta\left(2^i * \frac{n}{2^i}\right) =$ $n * \Theta(1) + \Theta(n) * \sum_{i=0}^{\log(n)-1} 1 =$ $\Theta(n) + \Theta(n) * \Theta(\log(n)) = \Theta(n * \log(n))$	<ul style="list-style-type: none"> <li>Ad ogni liv. abbiamo <math>2^i</math> nodi</li> <li>Costo nodo al liv. i è: <math>\Theta\left(\frac{n}{2^i}\right)</math></li> <li>Costo liv. i è: <math>2^i * \Theta\left(\frac{n}{2^i}\right) = \Theta(n)</math></li> <li>L'altezza dell'albero è: <math>\frac{n}{2^i} = 1 \rightarrow i = \log(n)</math></li> <li>Costo di tutti i liv.: <math>\Theta(n) * \Theta(\log(n)) = \Theta(n * \log(n))</math></li> </ul>	<p>Rimuoviamo la notazione asintotica:</p> $T(n) = 2T(n/2) + cn$ per $c > 0$ $T(1) = d$ per $d > 0$ <p><b>Ipotesi:</b> <math>T(n) \geq a * n * \log(n)</math></p> <p>Caso base: <math>T(1) \geq a * 1 * \log(1) = 0</math>, vera</p> <p>Passo induttivo:</p> $T(n) \geq 2 \left( a \frac{n}{2} \log\left(\frac{n}{2}\right) \right) + cn =$ $an * \log\left(\frac{n}{2}\right) + cn = an(\log(n) - 1) + cn =$ $an\log(n) - an + cn \geq an\log(n) \rightarrow$ $-an + cn \geq 0 \rightarrow cn \geq an \rightarrow c \geq a$ <p>Quindi <math>T(n) = \Omega(n * \log(n))</math></p> <p><b>Ipotesi:</b> Siccome usando <math>T(n) \leq b * n * \log(n)</math> nel caso base avremmo <math>T(n) \leq 0</math>, tentiamo con</p> $T(n) \leq b * n * \log(n) + h$ <p>Caso base: <math>T(1) \leq 0 + h</math>, vera per alcuni <math>h</math></p> <p>Passo induttivo:</p> $T(n) \leq 2 \left( b \frac{n}{2} \log\left(\frac{n}{2}\right) + h \right) + cn =$ $bn\log\left(\frac{n}{2}\right) + 2h + cn =$ $bn\log(n) - bn + 2h + cn \leq bn\log(n) + h \rightarrow$ $-bn + h + cn \leq 0 \rightarrow cn + h \leq bn$ <p>Quindi <math>T(n) = O(n * \log(n))</math></p> $T(n) = \Omega(n * \log(n)) \text{ e } O(n * \log(n)) \rightarrow \Theta(n * \log(n))$	<ul style="list-style-type: none"> <li><math>a=2, b=2</math></li> <li><math>f(n) = \Theta(n)</math></li> <li><math>n^{\log_2 2} = n</math></li> </ul> <p>Ci troviamo nel caso 2, siccome <math>f(n) = \Theta(n^{\log_2 2})</math></p> $T(n) = \Theta(n^{\log_2 2} * \log(n)) = \Theta(n * \log(n))$

$$T(n) = 2T(n/2) + \Theta(n^2) \quad T(1) = \Theta(1)$$

Iterativo	Albero	Sostituzione	Principale
<p>Sviluppando:</p> $2T(n/2) + \Theta(n^2)$ $4T(n/4) + 2\Theta(n^2/4) + \Theta(n)$ $8T(n/8) + 4\Theta(n^2/16) + 2\Theta(n^2/4) + \Theta(n)$ <p>La ricorsione continua finché <math>\frac{n}{2^k} = 1 \rightarrow k = \log(n)</math>:</p> $T(n) = 2^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 2^i \Theta\left(\frac{n^2}{2^{2i}}\right) =$ $2^{\log(n)} * \Theta(1) + \Theta(n^2) * \sum_{i=0}^{\log(n)-1} \left(\frac{1}{2}\right)^i =$ $n^{\log(2)} * \Theta(1) + \Theta(n^2) * 1 =$ $\Theta(n) + \Theta(n^2) = \Theta(n^2)$	<ul style="list-style-type: none"> <li>Ad ogni liv. abbiamo <math>2^i</math> nodi</li> <li>Costo nodo al liv. i è: <math>\Theta\left(\left(\frac{n}{2^i}\right)^2\right) = \Theta\left(\frac{n^2}{2^{2i}}\right)</math></li> <li>Costo liv. i è: <math>2^i * \Theta\left(\frac{n^2}{2^{2i}}\right) = \Theta\left(\frac{n^2}{2^i}\right)</math></li> <li>L'altezza dell'albero è: <math>\log(n)</math></li> <li>Costo di tutti i liv.: <math>\Theta(n^2) * \sum_{i=0}^{\log(n)-1} \left(\frac{1}{2}\right)^i = \Theta(n^2) * \sum_{i=0}^{\log(n)-1} \left(\frac{1}{2}\right)^i = \Theta(n^2) * 1 = \Theta(n^2)</math></li> </ul>	<p>Rimuoviamo la notazione asintotica</p> $T(n) = 2T(n/2) + cn^2$ e $T(1) = d$ <p><b>Ipotesi:</b> <math>T(n) \leq an^2</math></p> <p>Caso base: <math>d \leq a</math></p> <p>Passo induttivo: <math>T(n) \leq 2a * \left(\frac{n}{2}\right)^2 + cn^2 = 2a * \frac{n^2}{4} + cn^2 = \frac{an^2}{2} + cn^2 \leq an^2</math></p> <p>Solo se <math>c \leq a/2</math>. Quindi <math>T(n) = O(n^2)</math></p> <p><b>Ipotesi:</b> <math>T(n) \geq bn^2</math></p> <p>Caso base: <math>d \geq b</math></p> <p>Passo induttivo: <math>T(n) \geq 2b * \left(\frac{n}{2}\right)^2 + cn^2 = \frac{bn^2}{2} + cn^2 \geq bn^2 \rightarrow \frac{b}{2} + c \geq b</math></p> <p>Solo se <math>c \geq b/2</math></p> <p>Quindi <math>T(n) = \Omega(n^2)</math> e quindi <math>T(n) = \Theta(n^2)</math></p>	<ul style="list-style-type: none"> <li><math>a=2, b=2</math></li> <li><math>f(n) = \Theta(n^2)</math></li> <li><math>n^{\log_2 2} = n</math></li> </ul> <p>Ci troviamo nel caso 1: abbiamo che <math>\Theta(n^2) = \Omega(n^{\log_2 2+\varepsilon})</math> con <math>\varepsilon=1</math> e <math>2 * \frac{n^2}{4} \leq c * n^2</math>, se prendiamo <math>c=1/2</math> abbiamo: <math>\frac{1}{2}n \leq \frac{1}{2}n</math></p> $T(n) = \Theta(n^2)$

$$T(n) = 2T(n/3) + \Theta(n) \quad T(1) = \Theta(1)$$

Iterativo	Albero	Sostituzione	Principale
<p>Sviluppando:</p> <ol style="list-style-type: none"> <li>1) <math>2T(n/3) + \Theta(n)</math></li> <li>2) <math>4T(n/9) + 2\Theta(n/3) + \Theta(n)</math></li> <li>3) <math>8T(n/27) + 4\Theta(n/9) + 2\Theta(n/3) + \Theta(n)</math></li> </ol> <p>La ricorsione continua finché <math>\frac{n}{3^k} = 1 \rightarrow k = \log_3 n</math>:</p> $T(n) = 2^k T\left(\frac{n}{3^k}\right) + \sum_{i=0}^{k-1} 2^i \Theta\left(\frac{n}{3^i}\right) =$ $2^{\log_3(n)} * \Theta(1) + \Theta(n) * \sum_{i=0}^{\log_3(n)-1} \left(\frac{2}{3}\right)^i =$ $n^{\log_3 2} * \Theta(1) + \Theta(n) * 1 =$ $\Theta(n^{\log_3 2}) + \Theta(n) = \Theta(n)$ <p>Perchè <math>\log_3(2) &lt; 1</math></p>	<ul style="list-style-type: none"> <li>Ad ogni liv. abbiamo nodi</li> <li>Costo nodo al liv. i è:</li> <li>Costo liv. i è:</li> <li>L'altezza dell'albero è:</li> <li>Costo di tutti i liv.</li> </ul>	$f$	<ul style="list-style-type: none"> <li><math>a=2, b=3</math></li> <li><math>f(n) = \Theta(n)</math></li> <li><math>n^{\log_3 2}</math></li> </ul> <p>Ci troviamo nel caso 3, siccome <math>\log_3(2) &lt; 1</math>, abbiamo che <math>f(n) = \Omega(n^{\log_3 2+\varepsilon})</math> e <math>2 * \frac{n}{3} \leq c * n</math>, se prendiamo <math>c=2/3</math> abbiamo: <math>\frac{2}{3}n \leq \frac{2}{3}n</math></p> $T(n) = \Theta(n)$

$$T(n) = 3T(n/2) + \Theta(n) \quad T(1) = \Theta(1)$$

Iterativo	Albero	Sostituzione	Principale
<p>Sviluppando:</p> <ol style="list-style-type: none"> <li>1) <math>3T(n/2) + \Theta(n)</math></li> <li>2) <math>9T(n/4) + 3\Theta(n/2) + \Theta(n)</math></li> <li>3) <math>27T(n/8) + 9\Theta(n/4) + 3\Theta(n/2) + \Theta(n)</math></li> </ol> <p>La ricorsione continua finché <math>\frac{n}{2^k} = 1 \rightarrow k = \log(n)</math>:</p>	<ul style="list-style-type: none"> <li>Ad ogni liv. abbiamo nodi</li> <li>Costo nodo al liv. i è:</li> <li>Costo liv. i è:</li> <li>L'altezza dell'albero è:</li> <li>Costo di tutti i liv.</li> </ul>	$f$	<ul style="list-style-type: none"> <li><math>a=3, b=2</math></li> <li><math>f(n) = \Theta(n)</math></li> <li><math>n^{\log_2 3}</math></li> </ul> <p>Ci troviamo nel caso 1, siccome <math>\log_2(3) &gt; 1</math>, abbiamo che <math>f(n) = \Omega(n^{\log_2 3})</math></p>

$$\begin{aligned}
T(n) &= 3^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 3^i \Theta\left(\frac{n}{2^i}\right) = \\
3^{\log_2(n)} * \Theta(1) + \Theta(n) * \sum_{i=0}^{\log_2(n)-1} \left(\frac{3}{2}\right)^i &= \\
n^{\log_2 3} * \Theta(1) + \Theta(n) * \left(\frac{\left(\frac{3}{2}\right)^{\log_2 n} - 1}{\frac{3}{2} - 1}\right) &= \\
\Theta(n^{\log_2 3}) + \Theta(n) * \left(\frac{n^{\log_2(\frac{3}{2})} - 1}{\frac{1}{2}}\right) &= \\
\Theta(n^{\log_2 3}) + \Theta(n) * \Theta\left(n^{\log_2 \frac{3}{2}}\right) &= \\
\Theta(n^{\log_2 3}) + \Theta(n^{\log_2 3}) + \Theta(n^{1+\log_2 3-\log_2 2}) &= \\
\Theta(n^{\log_2 3}) + \Theta(n^{\log_2 3}) = \Theta(n^{\log_2 3})
\end{aligned}$$

$$T(n) = \Theta(n^{\log_2 3})$$

$$T(n) = 3T(n/4) + \Theta(n) \quad T(1) = \Theta(1)$$

### Iterativo

Sviluppando:

$$3T(n/4) + \Theta(n)$$

$$9T(n/16) + 3\Theta(n/4) + \Theta(n)$$

$$27T(n/64) + 9\Theta(n/16) + 3\Theta(n/4) + \Theta(n)$$

La ricorsione continua finché  $\frac{n}{4^k} = 1 \rightarrow k = \log_4 n$ :

$$\begin{aligned}
T(n) &= 3^k T\left(\frac{n}{4^k}\right) + \sum_{i=0}^{k-1} 3^i \Theta\left(\frac{n}{2^i}\right) = \\
3^{\log_4 n} T(1) + \Theta(n) \sum_{i=0}^{\log_4 n-1} \left(\frac{3}{4}\right)^i &= \\
n^{\log_4 3} * \Theta(1) + \Theta(n) * 1 &= \\
\Theta(n^{\log_4 3}) + \Theta(n) = \Theta(n) \text{ poiché } \log_4(3) < 1
\end{aligned}$$

$$T(n) = 4T(n/2) + \Theta(n) \quad T(1) = \Theta(1)$$

### Albero

- Ad ogni liv. abbiamo  $3^k$  nodi
- Costo nodo al liv. i è:  $\Theta\left(\frac{n}{4^k}\right)$
- Costo liv. i è:  $3^k * \Theta\left(\frac{n}{4^k}\right) = \Theta(n) * \left(\frac{3}{4}\right)^k$
- L'altezza dell'albero è:  $\frac{n}{4^k} = 1 \rightarrow k = \log_4 n$
- Costo di tutti i liv.

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_4 n} \Theta(n) * \left(\frac{3}{4}\right)^k = \\
\Theta(n) * \sum_{i=0}^{\log_4 n} \left(\frac{3}{4}\right)^k &= \Theta(n) * 1 = \Theta(n)
\end{aligned}$$

### Sostituzione

$$f$$

### Principale

$$\bullet a=3, b=4$$

$$\bullet f(n) = \Theta(n)$$

$$\bullet n^{\log_4 3}$$

Ci troviamo nel caso 3, siccome  $\log_4(3) < 1$ , abbiamo che  $f(n) = \Omega(n^{\log_4 3})$  e  $3 * \frac{n}{4} \leq c * n$ , se prendiamo  $c = 3/4$  allora abbiamo  $\frac{3n}{4} \leq \frac{3}{4} * n$

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

$$T(n) = 2T(n/2) + \Theta(n^3) \quad T(1) = \Theta(1)$$

### Iterativo

Sviluppando:

$$2T\left(\frac{n}{2}\right) + \Theta(n^3)$$

$$4T\left(\frac{n}{4}\right) + 2\Theta\left(\left(\frac{n}{2}\right)^3\right) + \Theta(n^3)$$

La ricorsione continua finché  $\frac{n}{2^k} = 1 \rightarrow k = \log(n)$ :

$$\begin{aligned}
T(n) &= 2^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 2^i \Theta\left(\left(\frac{n}{2^i}\right)^3\right) = \\
2^{\log(n)} T(1) + \Theta(n^3) * \sum_{i=0}^{\log(n)-1} \frac{1}{2^{2i}} &= \\
n^{\log(2)} \Theta(1) + \Theta(n^3) * \sum_{i=0}^{\log(n)-1} \left(\frac{1}{4}\right)^i &= \\
n * \Theta(1) + \Theta(n^3) * \Theta(1) &= \Theta(n) + \Theta(n^3) \\
&= \Theta(n^3)
\end{aligned}$$

$$T(n) = 16T(n/4) + \Theta(n^2) \quad T(1) = \Theta(1)$$

### Albero

- Ad ogni liv. abbiamo  $2^i$  nodi
- Costo nodo al liv. i è:  $\Theta\left(\left(\frac{n}{2^i}\right)^3\right)$
- Costo liv. i è:  $2^i * \Theta\left(\left(\frac{n}{2^i}\right)^3\right) = \Theta\left(n^3 * \frac{2^i}{2^{3i}}\right) = \Theta\left(n^3 * \frac{1}{2^{2i}}\right) = \Theta\left(n^3 * \left(\frac{1}{4}\right)^i\right)$
- L'altezza dell'albero è:  $\log(n)$
- Costo di tutti i liv.

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log(n)} \Theta\left(n^3 * \left(\frac{1}{4}\right)^i\right) = \Theta(n^3) * \sum_{i=0}^{\log(n)} \left(\frac{1}{4}\right)^i = \\
\Theta(n^3) * \Theta(1) &= \Theta(n^3)
\end{aligned}$$

### Sostituzione

$$f$$

### Principale

$$\bullet a=2, b=2$$

$$\bullet f(n) = \Theta(n^3)$$

$$\bullet n^{\log_2 2} = n$$

Ci troviamo nel caso 3, siccome  $f(n) = \Omega(n^{\log_2 2})$

$$e 2 * \left(\frac{n}{2}\right)^3 \leq c * n^3 \rightarrow$$

$$2 * \frac{n^3}{8} \leq c * n^3 \rightarrow$$

$$\frac{n^3}{4} \leq c * n^3 \text{ se prendiamo } c = 1/4$$

$$\text{abbiamo } \frac{n^3}{4} \leq 1/4$$

$$T(n) = \Theta(n^3)$$

### Iterativo

Sviluppando:

$$16T\left(\frac{n}{4}\right) + \Theta(n^2)$$

### Albero

- Ad ogni liv. abbiamo  $16^i$  nodi
- Costo nodo al liv. i è:  $\Theta\left(\left(\frac{n}{4^i}\right)^2\right) =$

### Sostituzione

$$f$$

### Principale

$$\bullet a=16, b=4$$

$$\bullet f(n) = \Theta(n^2)$$

$$\bullet n^{\log_4 16} = n^2$$

$16^2 T\left(\frac{n}{4^2}\right) + 16\theta\left(\frac{n^2}{4^2}\right) + \Theta(n^2)$ $16^3 T\left(\frac{n}{4^3}\right) + 16^2\theta\left(\frac{n^2}{4^4}\right) + 16\theta\left(\frac{n^2}{4^2}\right) + \Theta(n^2)$ <p>La ricorsione continua finche <math>\frac{n}{4^k} = 1 \rightarrow k = \log_4 n</math>:</p> $T(n) = 16^k T\left(\frac{n}{4^k}\right) + \sum_{i=0}^{k-1} 16^i \theta\left(\frac{n^2}{4^{2i}}\right) =$ $16^{\log_4 n} T(1) + \Theta(n^2) * \sum_{i=0}^{\log_4 n - 1} 16^i * \frac{1}{4^{2i}} =$ $n^{\log_4 16} \theta(1) + \Theta(n^2) * \sum_{i=0}^{\log_4 n - 1} \frac{4^{2i}}{4^{2i}} =$ $n^2 * \theta(1) + \Theta(n^2) * \sum_{i=0}^{\log_4 n - 1} 1 =$ $\Theta(n^2) + \Theta(n^2) * \Theta(\log(n)) =$ $= \Theta(n^2 * \log(n))$	$\Theta\left(\frac{n^2}{4^{2i}}\right)$ <ul style="list-style-type: none"> <li>Costo liv. i è: <math>16^i * \Theta\left(\frac{n^2}{4^{2i}}\right) = \Theta(n^2) * \frac{4^{2i}}{4^{2i}} = \Theta(n^2)</math></li> <li>L'altezza dell'albero è: <math>\frac{n}{4^i} = 1 \rightarrow i = \log_4 n</math></li> <li>Costo di tutti i liv.</li> </ul> $T(n) = \sum_{i=0}^{\log_4 n} \Theta(n^2) = \Theta(n^2) * \sum_{i=0}^{\log_4 n} 1 =$ $\Theta(n^2) * \Theta(\log(n)) = \Theta(n^2 * \log(n))$	Ci troviamo nel caso, siccome $f(n) = \Theta(n^{\log_4 16})$ $T(n) = \Theta(n^2 * \log(n))$
---	---	--

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

Iterativo	Albero	Sostituzione	Principale
Sviluppando: $T(n-1)+\Theta(n)$ $T(n-2)+\Theta(n-1)+\Theta(n)$ $T(n-3)+\Theta(n-2)+\Theta(n-1)+\Theta(n)$ La ricorsione continua finche $n-k=1 \rightarrow k=n-1$ : $T(n) = T(n-k) + \sum_{i=0}^{k-1} \Theta(n-i)$ $T(1) + \sum_{j=0}^{n-2} \Theta(j) = \Theta(1) + \Theta(n^2) = \Theta(n^2)$	<b>Albero</b> <ul style="list-style-type: none"> <li>Ad ogni liv. abbiamo 1 nodi</li> <li>Costo liv. i è: <math>\Theta(n-i)</math></li> <li>L'altezza dell'albero è: <math>n-k=1 \rightarrow k=n-1</math></li> <li>Costo di tutti i liv.</li> </ul> $T(n) = \sum_{i=0}^{n-1} \Theta(n-i) = \sum_{j=0}^{n-1} \Theta(j) = \Theta(n^2)$	<b>f</b>	• $a=1, b=0$ Non siamo nella forma corretta per poter applicare il metodo principale

$T(n)=3T(n/2)+\Theta(n \log n)$   $T(1)=\Theta(1)$

Questo viene uno sviluppo iterativo un po più lungo quindi lo faccio esterno alla tabella sviluppando:

$$3T\left(\frac{n}{2}\right) + \Theta(n \log(n))$$

$$9T\left(\frac{n}{4}\right) + 3\Theta\left(\frac{n}{2} \log\left(\frac{n}{2}\right)\right) + \Theta(n \log(n)) = 9T\left(\frac{n}{4}\right) + 3\Theta\left(\frac{n}{2}(\log(n)-1)\right) + \Theta(n \log(n))$$

$$27T\left(\frac{n}{8}\right) + 9\Theta\left(\frac{n}{4} \log\left(\frac{n}{4}\right)\right) + 3\Theta\left(\frac{n}{2}(\log(n)-1)\right) + \Theta(n \log(n)) = 27T\left(\frac{n}{8}\right) + 9\Theta\left(\frac{n}{4}(\log(n)-2)\right) + 3\Theta\left(\frac{n}{2}(\log(n)-1)\right) + \Theta(n \log(n))$$

La ricorsione continua finche  $\frac{n}{2^k} = 1 \rightarrow k = \log_2 n$ :

$$T(n) = 3^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 3^i \Theta\left(\frac{n}{2^i} (\log(n)-i)\right) = 3^{\log_2 n} T(1) + \Theta(n) \sum_{i=0}^{\log_2 n - 1} \left(\frac{3}{2}\right)^i (\log(n)-i) = n^{\log_2 3} \Theta(1) + \Theta(n)$$

Albero	Sostituzione	Principale
<ul style="list-style-type: none"> <li>Ad ogni liv. abbiamo <math>3^i</math> nodi</li> <li>Costo nodo al liv. i è: <math>\Theta\left(\frac{n}{2^i} \log\left(\frac{n}{2^i}\right)\right) = \Theta\left(\frac{n}{2^i} (\log(n) - \log(2^i))\right) = \Theta\left(\frac{n}{2^i} (\log(n) - i)\right)</math></li> <li>Costo liv. i è: <math>3^i \Theta\left(\frac{n}{2^i} (\log(n)-i)\right)</math></li> <li>L'altezza dell'albero è: <math>\frac{n}{2^k} = 1 \rightarrow k = \log_2 n</math></li> <li>Costo di tutti i liv.</li> </ul> $\sum_{i=0}^{\log_2 n} 3^i \Theta\left(\frac{n}{2^i} (\log(n)-i)\right) = \Theta(n) \sum_{i=0}^{\log_2 n} \left(\frac{3}{2}\right)^i (\log(n)-i) =$	<b>f</b>	• $a=3, b=2$ • $f(n) = \Theta(n \log(n))$ • $n^{\log_2 3}$ Siccome $\log_2(3) > 1$ , $\Theta(n \log(n))$ è asintoticamente più grande di $n^{\log_2 3}$ ma non polinomialmente più grande. Infatti $\log(n)$ è asintoticamente minore di $n^\epsilon$ per qualunque $\epsilon > 0$ . Quindi non possiamo applicare il metodo principale

Calcolare l'equazione di ricorrenza associata al seguente algoritmo e risolverla con tutti i metodi possibili:

```
def Test (n)
    k = 0
    for i in range(1, n):
        k = k + 1
    if n ≤ 1: return k
    else: return (Test(n DIV 2)+Test(n DIV 4))
```

Il ciclo for viene iterato n volte sia nel caso che nel passo ricorsivo  $\rightarrow \Theta(n)$

il caso base si incontra quando  $n = 1$  quindi il ciclo verrà eseguito 1 volta con costo costante  $\rightarrow T(1) = \Theta(1)$

Si effettuano due chiamate ricorsive differenti, una con  $n/2$  e una con  $n/4$

L'eq. di ricorrenza sarà:

$$T = \begin{cases} T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + \Theta(n) \\ T(1) = \Theta(1) \end{cases}$$

Poiché abbiamo due chiamate ricorsive diverse conviene calcolare i limiti asintotici superiori e inferiori separatamente

Per calcolare  $O(T)$ , calcoliamo su  $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ , poiché  $T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + \Theta(n) \leq T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \Theta(n)$

Per calcolare  $\Omega(T)$ , calcoliamo su  $T(n) = 2T\left(\frac{n}{4}\right) + \Theta(n)$ , poiché  $T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + \Theta(n) \geq T\left(\frac{n}{4}\right) + T\left(\frac{n}{4}\right) + \Theta(n)$

### Iterativo

Calcolo  $O(T)$ :  $2T(n/2) + \Theta(n)$

Espansione:

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

$$4T\left(\frac{n}{4}\right) + 2\Theta\left(\frac{n}{2}\right) + \Theta(n) = 4T\left(\frac{n}{4}\right) + \Theta(n) + \Theta(n) = 4T\left(\frac{n}{4}\right) + 2\Theta(n)$$

$$8T\left(\frac{n}{8}\right) + 4\Theta\left(\frac{n}{4}\right) + 2\Theta\left(\frac{n}{2}\right) + \Theta(n) = 8T\left(\frac{n}{8}\right) + \Theta(n) + \Theta(n) + \Theta(n) = 8T\left(\frac{n}{8}\right) + 3\Theta(n)$$

Si ferma quando  $\frac{n}{2^k} = 1 \rightarrow k = \log(n)$ :

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + k * \Theta(n) = 2^{\log(n)} T(1) + \log(n) * \Theta(n) = n^{\log(2)} * \Theta(1) + \Theta(n * \log(n)) = n * \Theta(1) + \Theta(n * \log(n)) = \Theta(n * \log(n))$$

$$T(n) = O(n * \log(n))$$

Calcolo  $\Omega(T)$ :  $4T(n/4) + \Theta(n)$

Espansione:

$$4T\left(\frac{n}{4}\right) + \Theta(n)$$

$$16T\left(\frac{n}{16}\right) + 4\Theta\left(\frac{n}{4}\right) + \Theta(n) = 16T\left(\frac{n}{16}\right) + 2\Theta(n)$$

$$64T\left(\frac{n}{64}\right) + 16\Theta\left(\frac{n}{16}\right) + 4T\left(\frac{n}{4}\right) + \Theta(n) = 64T\left(\frac{n}{64}\right) + 3\Theta(n)$$

Si ferma quando  $\frac{n}{4^k} = 1 \rightarrow k = \log_4 n$ :

$$T(n) = 4^k T\left(\frac{n}{4^k}\right) + k * \Theta(n) = 4^{\log_4 n} * T(1) + \log_4 n * \Theta(n) = n^{\log_4 4} * \Theta(1) + \Theta(n * \log(n)) = n * \Theta(1) + \Theta(n * \log(n)) = \Theta(n * \log(n))$$

$$T(n) = \Omega(n * \log(n))$$

Poiché  $T(n) = O(n * \log(n))$  e  $\Omega(n * \log(n))$  allora  $T(n) = \Theta(n * \log(n))$

### Albero

Calcolo  $O(T)$ :  $2T(n/2) + \Theta(n)$

- nodi al liv. i:  $2^i$

### Principale

Calcolo  $O(T)$ :  $2T(n/2) + \Theta(n)$

- $a = 2, b = 2$
- $f(n) = \Theta(n)$
- $n^{\log_2 2} = n$

Calcolo  $\Omega(T)$ :  $4T(n/4) + \Theta(n)$

- $a = 4, b = 4$
- $f(n) = \Theta(n)$
- $n^{\log_4 4} = n$

In entrambi i casi siamo nel caso 2:  $f(n) = \Theta(n^{\log_2 2}) = \Theta(n^{\log_4 4})$

Quindi abbiamo in entrambi i casi il costo  $T(n) = \Theta(n * \log(n))$ , quindi siccome  $T(n) = O(n * \log(n))$  e  $T(n) = \Omega(n * \log(n))$  allora  $T(n) = \Theta(n * \log(n))$

# Esonero 4/24

giovedì 10 aprile 2025 11:32

## Es 1

Si calcoli la complessità asintotica della seguente funzione iterativa

```
def es1(n):
    z, x, y = 0, 0, n
    while y > 1:
        x += 1
        y = y//3
    x = x * x
    while y < n:
        z += 5
        y += x
    return z
```

### Soluzione

Il primo ciclo si ferma quando  $y \approx 1$ ,  $y$  parte da  $n$  e ad ogni iter. viene diviso per 3, quindi si fermerà dopo  $\frac{n}{3^k} \rightarrow k = \log_3 n$  iterazioni  $\rightarrow \Theta(\log(n))$

Alla fine del primo ciclo,  $x$  ha val  $\log_3(n)^2$

il secondo ciclo si ferma quando  $y \geq n$ ,  $y$  parte da  $\approx 1$  e ad ogni iter. gli viene aggiunto  $\log_3(n)^2$ , quindi si fermerà dopo  $k * \log_3(n)^2 = n \rightarrow i = \frac{n}{\log_3(n)^2}$  iter  $\rightarrow \Theta(\frac{n}{\log(n)^2})$

Poiché le altre op. hanno tempo costante, il costo è dato dalla somma dei due while:  $\Theta(\log(n)) + \Theta(\frac{n}{\log(n)^2}) = \Theta(\frac{n}{\log_3(n)^2})$

## Es 2

Si consideri la seguente funzione ricorsiva:

- Si imposti la relazione di ricorrenza che ne definisce il tempo di esecuzione giustificando dettagliatamente l'equazione ottenuta.
- Si risolva la ricorrenza usando prima il metodo principale e poi a vostra scelta uno tra il metodo iterativo e il metodo di sostituzione.

```
def es2(n):
    if n <= 1:
        return 5
    j = x = 0
    while j*j < n:
        if j < 2:
            x += 2*es2(n//4) + 3
        j += 2
        x += 3
    return x
```

Il caso base si incontra quando  $n \leq 1$  ed ha costo costante  $\rightarrow T(1) = \Theta(1)$

il ciclo while viene eseguito quando  $j^2 \approx n$ , ad ogni iter. gli viene aggiunto 2, quindi si fermerà quando  $2k^2 = n \rightarrow k^2 = \frac{n}{2} \rightarrow k = \sqrt{\frac{n}{2}} \rightarrow k = \sqrt{n} \rightarrow \Theta(\sqrt{n})$

La chiamata ricorsiva viene effettuata però quando  $j < 2$  e siccome parte da 0 e ad ogni iter, viene aggiunto 2, la funzione verrà chiamata una sola volta quando  $j = 0$ , nella sua chiamata  $n$  viene diviso per 4

Quindi l'eq di ricorrenza è:

$$T = \begin{cases} T(n) = T\left(\frac{n}{4}\right) + \Theta(\sqrt{n}) \\ T(1) = \Theta(1) \end{cases}$$

### Metodo Iterativo

Espansione:

$$T\left(\frac{n}{4}\right) + \Theta(\sqrt{n})$$

$$T\left(\frac{n}{16}\right) + \Theta\left(\sqrt{\frac{n}{4}}\right) + \Theta(\sqrt{n}) = T\left(\frac{n}{16}\right) + \Theta\left(\frac{\sqrt{n}}{2}\right) + \Theta(\sqrt{n})$$

$$T\left(\frac{n}{64}\right) + \Theta\left(\sqrt{\frac{n}{16}}\right) + \Theta\left(\sqrt{\frac{n}{4}}\right) + \Theta(\sqrt{n}) = T\left(\frac{n}{64}\right) + \Theta\left(\frac{\sqrt{n}}{4}\right) + \Theta\left(\frac{\sqrt{n}}{2}\right) + \Theta(\sqrt{n})$$

Si ferma quando  $\frac{n}{4^k} = 1 \rightarrow k = \log_4(n)$ :

$$T(n) = T\left(\frac{n}{4^k}\right) + \sum_{i=0}^{k-1} \Theta\left(\frac{\sqrt{n}}{2^i}\right) = T(1) + \Theta(\sqrt{n}) * \sum_{i=0}^{k-1} \frac{1}{2^i} = \Theta(1) + \Theta(\sqrt{n}) * \sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^i = \Theta(1) + \Theta(\sqrt{n}) * \Theta(1) = \Theta(\sqrt{n})$$

Quindi  $T(n) = \Theta(\sqrt{n})$

### Metodo Sostituzione

Rimuoviamo la notazione asintotica

$$T = \begin{cases} T(n) = T\left(\frac{n}{4}\right) + b\sqrt{n} \\ T(1) = a \end{cases}$$

Ipotesi:  $T(n) \leq c*\sqrt{n}$  per qualche costante  $a$  da determinare

- Caso base:  $a \leq c*\sqrt{1} \rightarrow a \leq c$ , vera per  $c \geq a$
- Passo induttivo:  $T(n) \leq c\sqrt{\frac{n}{4}} + b\sqrt{n} = c\frac{\sqrt{n}}{2} + b\sqrt{n} \leq c\sqrt{n} + 2b\sqrt{n} \leq 2c\sqrt{n} \rightarrow 2b\sqrt{n} \leq c\sqrt{n} \rightarrow b \leq \frac{c}{2}$

Quindi  $T(n) = O(\sqrt{n})$

calcolando allo stesso modo per l'ipotesi  $T(n) \geq d*\sqrt{n}$ , troviamo che  $T(n) = \Omega(\sqrt{n})$

Siccome abbiamo  $T(n) = \Omega(\sqrt{n})$  e  $T(n) = O(\sqrt{n})$  allora  $T(n) = \Theta(\sqrt{n})$

## Esercizio 1

funzione Exam( $n$ ):

```

if n <= 2:  return 2 * n;
b ← n/2;
tot ← n * n;
for i = 1 to n:
    for j = 1 to i:
        tot ← tot + i - j;
for i = 1 to 4:
    for j = 1 to 4:
        if i = j:  tot ← tot + Exam(b)
        else:  tot ← tot + i - j;
return tot.
```

- a) Si imposti la relazione di ricorrenza che ne definisce il tempo di esecuzione giustificando l'equazione ottenuta.
- b) Si risolva l'equazione usando il metodo iterativo, commentando opportunamente i passaggi del calcolo;
- c) Si risolva l'equazione usando il metodo principale, specificando quale caso del teorema si applica e perche' e oppure per quale motivo non si puo' applicare il teorema.

## Soluzione

- a) **Costo Exam():**

- Linee 2-4: Op. costanti →  $\Theta(1)$
- Linee 5-7: Due cicli for
  - Ciclo esterno: itera da 1 a  $n \rightarrow n$  iter
  - Ciclo interno: itera da 1 a  $i \rightarrow i$  iter
  - Le op nel ciclo sono costanti →  $\Theta(1)$

Costo:

$$\Theta(1) * \sum_{i=1}^n i = \Theta(1) * \frac{n(n+1)}{2} = \Theta\left(\frac{n^2+n}{2}\right) = \Theta(n^2)$$

- Linee 8-11: Due cicli for
  - Ciclo esterno e interno iterano da **1 a 4** entrambi, #tot iterazioni =  $4 \times 4 = 16$
  - Per 4 volte viene chiamato ricursivamente **Exam(b)**, dove  $b = n/2$

Costo:

$$4T(n/2) + \Theta(1)$$

Costo totale:

$$T(n) = \begin{cases} T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^2) \\ T(1) = \Theta(1) \end{cases}$$

- b) **Metodo Iterativo**

Sviluppo  $T(n)$ :

- $4T\left(\frac{n}{2}\right) + \Theta(n^2)$
- $4\left(4T\left(\frac{n}{2^2}\right) + \Theta\left(\frac{n^2}{2}\right)\right) + \Theta(n^2) = 4^2T\left(\frac{n}{2^2}\right) + 2\Theta(n^2)$
- $4\left(4\left(4T\left(\frac{n}{2^3}\right) + \Theta\left(\frac{n^2}{2^2}\right)\right) + \Theta(n^2)\right) + \Theta(n^2) = 4^3T\left(\frac{n}{2^3}\right) + 3\Theta(n^2)$

Generalizzo:

$$4^k T\left(\frac{n}{2^k}\right) + k\Theta(n^2) \quad 4^{\log(n)} = 2^{2\log(n)} = 2^{\log(n^2)} = n^2$$

Ci fermiamo quando  $\frac{n}{2^k} = 1$ , quindi quando  $k = \log(n)$ :

$$4^{\log(n)}T(1) + \log(n) * \Theta(n^2) = n^2 * \Theta(1) + \Theta(n^2 * \log(n)) = \Theta(n^2) * \Theta(n^2 * \log(n)) = \Theta(n^2 * \log(n))$$

- c) **Metodo Principale**

- $a = 4$
- $b = 2$
- $f(n) = \Theta(n^2)$
- $n^{\log_2(4)} = n^2$

Poiché  $\Theta(n^{\log_b(a)}) = \Theta(f(n))$ , siamo nel **caso 2**, quindi il costo è  $\Theta(n^2 * \log(n))$

## Esercizio 2

Progettare un algoritmo che, dato un array A di n interi distinti i cui elementi sono all'inizio in ordine crescente e da un certo punto in poi in ordine decrescente, restituisce in tempo  $O(\log n)$  il massimo intero presente nell'array.

Ad esempio: per  $A = [8, 10, 20, 80, 100, 200, 400, 500, 3, 2, 1]$  l'algoritmo deve restituire il valore 500.

Dell'algoritmo proposto

- a) si dia la descrizione a parole;
- b) si scriva lo pseudocodice;
- c) si calcoli il costo computazionale.

### Soluzione

L'array è diviso in due parti e il massimo si trova nel punto in cui la sequenza cambia direzione.

Applico la ricerca binaria:

Sia m la pos. in mezzo dell'array:

- se  $A[m-1] < A[m]$  vuol dire che stiamo nella parte crescente e il val. max sta nel sottoarray a destra, quindi cerco nel sottoarray di destra che parte da m.
- se  $A[m-1] > A[m]$  vuol dire che stiamo nella parte decrescente e abbiamo superato il val max e dobbiamo ricercarlo nel sottarray a sinistra, quindi cerchiamo nel sottoarray che arriva fino a m-1.

```
def Exam2(A):  
    if len(A) == 1: # se A ha un solo elemento  
        return A[0]  
    m = len(A)//2 # calcolo l'indice medio  
    if A[m-1] < A[m]:  
        # se ancora stiamo nella parte crescente, allora è nella metà superiore  
        return Exam2(A[m:])  
    else:  
        # se siamo nella parte decrescente, max è nella metà inferiore (l'abbiamo superato)  
        return Exam2(A[:m-1])
```

Ad ogni passo la dim. dell'array in cui ricercare il val. massimo si dimezza, di conseguenza il num. di iter. per un array di dimensione n è  $\log(n)$ .

Oppure possiamo calcolare l'eq. di ricorrenza e usare il metodo principale per calcolare il costo.

La chiamata ricorsiva viene eseguita 1 volta sulla metà dell'array.

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

- $a = 1, b = 2, f(n) = \Theta(1), \Theta(n^{\log_2(1)}) = \Theta(n^0) = \Theta(1)$
- Stiamo nel caso 2 quindi il costo è  $\Theta(\log(n))$

## Esercizio 3

Dato un albero binario T, radicato e con n nodi, definiamo un nodo u di T equilibrato se il sottoalbero sinistro di u e il sottoalbero destro di u hanno entrambi lo stesso numero di nodi.

Progettare un algoritmo che, dato il puntatore r alla radice di un albero binario memorizzato tramite record e puntatori, restituisca in tempo  $O(n)$  il numero dei suoi nodi equilibrati.

Dell'algoritmo proposto

- a) si dia la descrizione a parole;
- b) si scriva lo pseudocodice;
- c) si motivi il costo computazionale.

Qual è il numero minimo e qual è il numero massimo di nodi equilibrati che l'albero T pu' o avere? Motivare la risposta.

### Soluzione

Effettuiamo una visita postordine dove ogni nodo restituisce al padre il num. di nodi equilibrati e di nodi presenti nel suo sottoalbero.

Le foglie restituiscono 0 e 0.

Ogni nodo interno controllerà quanti nodi tot. e equilibrati ci sono nel suo sottoalbero sinistro e destro e restituirà al padre:

- il num. di nodi totali (sottoalbero destro, sinistro e se stesso (+1))
- la somma dei nodi equilibrati, più 1 se eventualmente il num. di nodi ne sottoalbero sinistro è uguale al num. di nodi del destro

```
def Exam3(p):  
    if p == None: # se siamo nella foglia  
        return 0, 0 # restituiamo 0 nodi equilibrati e 0 nodi totali  
    equi_l, num_l = Exam3(p.left) # chiamata ricorsiva a sinistra  
    equi_r, num_r = Exam3(p.right) # chiamata ricorsiva a destra  
    equi = equi_l + equi_r # sommiamo i nodi equilibrati  
    if num_l == num_r: # se il num. di nodi a sinistra è uguale a quello a destra  
        equi += 1 # incrementiamo il contatore di nodi equilibrati  
    return equi, num_l + num_r + 1 # restituiamo il num. nodi equilibrati di nodi tot
```

Il costo è quello di una semplice visita dell'albero, quindi  $O(n)$

Se l'albero è completo, allora tutti i suoi nodi sono equilibrati e il num. massimo di nodi equilibrati è n.

Se invece l'albero è sbilanciato, allora tutti i nodi sono in un unico ramo e nessun nodo è equilibrato, quindi il num. minimo è 0 (a meno che non si contino le foglie come equilibrate, in quel caso il num. minimo è 1)

# Esame 8/9/21

mercoledì 12 febbraio 2025 16:01

## Esercizio 1

funzione Exam( $n$ ):

```
tot ← n;  
if n <= 4: return tot;  
b ← n//4;  
tot ← tot+Exam(b);  
j ← 1;  
while j * j <= n do:  
    tot ← tot + j;  
    j ← j + 1;  
return tot+Exam(b).
```

- a) Si imposti la relazione di ricorrenza che ne definisce il tempo di esecuzione giustificando l'equazione ottenuta
- b) Si risolva l'equazione usando il metodo dell'albero, dettagliando i passaggi del calcolo e giustificando ogni affermazione.

### Soluzione

a)

- Caso Base:  $T(n \leq 4) = \Theta(1)$
- Costo di Exam():
  - op. costanti →  $\Theta(1)$
  - chiamata ricorsiva Exam( $n/4$ )
  - ciclo while itera fino a quando  $j^2 \leq n \Rightarrow j \leq \sqrt{n}$ , quindi il costo è  $\Theta(\sqrt{n})$
  - seconda chiamata ricorsiva Exam( $n/4$ )
- Eq. di ricorrenza:

$$T(n) = \begin{cases} T(n) = 2T\left(\frac{n}{4}\right) + \Theta(\sqrt{n}) \\ T(n) = \Theta(1) \text{ se } n \leq 4 \end{cases}$$

- Ad ogni liv. i abbiamo  $2^i$  nodi, poiché ad ogni nodo avremmo 2 chiamate ricorsive
- Costo di un nodo al liv. i sarà  $\sqrt{\frac{n}{4^i}} = \frac{\sqrt{n}}{\sqrt{4^i}} = \Theta\left(\frac{\sqrt{n}}{2^i}\right)$
- Costo complessivo al liv. i sarà  $2^i * \frac{\sqrt{n}}{2^i} = \Theta(\sqrt{n})$
- Ci fermiamo quando  $\frac{n}{4^i} = 1 \Rightarrow i = \log_4(n)$ , quindi l'altezza dell'albero è  $\Theta(\log(n))$
- Costo complessivo: dato dalla somma dei costi di tutti i livelli. Poiché ogni liv. ha costo  $\Theta(\sqrt{n})$  e l'altezza dell'albero è  $\Theta(\log(n))$ , il costo è:

$$\sum_{i=0}^{\log_4(n)} \Theta(\sqrt{n}) = \Theta(\sqrt{n}) * \Theta\left(\sum_{i=0}^{\log_4(n)} 1\right) = \Theta(\sqrt{n}) * \Theta(\log(n)) = \Theta(\sqrt{n} * \log(n))$$

Oppure potevamo scrivere: ad ogni liv. i ci sono  $2^i$  chiamate ricorsive, ciascuna con costo  $\Theta\left(\frac{\sqrt{n}}{2^i}\right)$ :

$$\sum_{i=0}^{\log_4(n)} 2^i * \Theta\left(\frac{\sqrt{n}}{2^i}\right) = \Theta(\sqrt{n}) * \sum_{i=0}^{\log_4(n)} 2^i * \frac{1}{2^i} = \Theta(\sqrt{n}) * \sum_{i=0}^{\log_4(n)} 1 = \Theta(\sqrt{n}) * \Theta(\log(n)) = \Theta(\sqrt{n} * \log(n))$$

## Esercizio 2

Progettare un algoritmo che, dati tre array A, B e C ordinati e contenenti ciascuno  $n$  interi distinti, stampi in tempo  $O(n)$  gli interi che compaiono nell'intersezione dei tre array. L'algoritmo proposto deve utilizzare spazio di lavoro  $\Theta(1)$ .

Ad esempio: per  $A = [1, 2, 3, 4, 5, 6]$ ,  $B = [1, 4, 5, 6, 8, 9]$  e  $C = [2, 4, 6, 7, 8, 9]$  l'algoritmo deve stampare gli elementi 4 e 6.

Dell'algoritmo proposto:

- a) si dia la descrizione a parole
- b) si scriva lo pseudocodice
- c) si giustifichi formalmente il costo computazionale.
- d) si dia un'idea di quello che accadrebbe al costo computazionale se si volesse generalizzarlo a  $\Theta(n)$  array.

### Soluzione

Il prof nella soluzione scorre gli elem. da destra verso sinistra usando il `max()` per calcolare gli elementi non in comune.

Nella mia soluzione faccio il contrario: scorro da sinistra verso destra e uso il `min()`

a)

Utilizziamo tre indici **a**, **b** e **c** che seguono i rispettivi array A, B e C e vengono incrementati se:

- Se  $A[a] = B[b] = C[c]$ , allora abbiamo trovato un elemento in comune e incrementiamo di 1 tutti e 3 gli indici
- altrimenti calcoliamo  $m = \min(A[a], B[b], C[c])$  per trovare il val. min tra i 3 array:
  - Se  $m = A[a]$ : il min. corrente è in A e decrementiamo l'indice a, in quanto l'elem. non è in comune
  - Se  $m = B[b]$ : il min. corrente è in B e decrementiamo l'indice b, in quanto l'elem. non è in comune

- o Se  $m = C[c]$ : il min. corrente è in C e decrementiamo l'indice c, in quanto l'elem. non è in comune

b)

```
def Exam2_8(A, B, C):
    a = b = c = 0 # inizializziamo i contatori
    while a < len(A) and b < len(B) and c < len(C): # finché non abbiamo finito
        if A[a] == B[b] == C[c]: # se troviamo un elemento comune
            print(A[a]) # stampiamo l'elemento
        m = min(A[a], B[b], C[c]) # calcoliamo il minore
        if m == A[a]: # se il minore è in A
            a += 1 # incrementiamo l'indice di A
        if m == B[b]: # se il minore è in B
            b += 1 # incrementiamo l'indice di B
        if m == C[c]: # se il minore è in C
            c += 1 # incrementiamo l'indice di C
```

c)

Il costo computazionale dipende dal num. di cicli while, il quale si ferma quando almeno uno degli indici non ha superato n. Ad ogni iterazione almeno uno degli indici incrementa o incrementano tutti e tre insieme.

Quindi il num. di iterazioni va da 0 a  $n-1$  nel caso migliore o da 0 a  $3n-3$  nel caso peggiore.

Ogni iter. del while richiede tempo costante, quindi il costo è  $\Theta(n)$ .

d)

Ad ogni iter. almeno uno dei  $\Theta(n)$  indici che tengono traccia della posizione all'interno dei vari array si incrementa.

Quindi il num. di iter. prima che uno degli indici superi la grandezza dell'array è **almeno n e al più  $n^2\Theta(n)$** .

Ogni iter. del while richiede tempo  $\Theta(n)$  (tempo richiesto per il calcolo dell'indice posizionato sull'elem. di val. minimo).

La complessità dell'algoritmo proposto in precedenza è dunque  $\Theta(n^3)$

### Esercizio 3

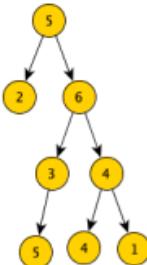
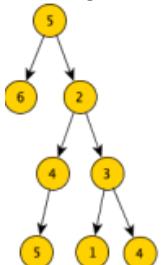
Si consideri un albero binario radicato T, i cui nodi hanno un campo valore contenente un intero. Bisogna modificare l'albero in modo che i nodi fratelli scambino tra loro il valore.

Si consideri ad esempio l'albero T in figura a sinistra, a destra viene riportato il risultato della modifica di T.

Progettare un algoritmo che, dato il puntatore r alla radice di T memorizzato tramite record e puntatori, effettui l'operazione di modifica in tempo  $O(n)$  dove n è il numero di nodi presenti nell'albero. Ogni nodo dell'albero è memorizzato in un record contenente il campo val con il valore del nodo e i campi left e right con i puntatori ai figli di sinistra e destra, rispettivamente.

Dell'algoritmo proposto:

- si dia la descrizione a parole
- si scriva lo pseudocodice
- si giustifichi formalmente il costo computazionale.



#### Soluzione

a)

Parto dalla radice ed eseguo il controllo sugli eventuali sottoalberi.

Ad ogni nodo che non sia una foglia controllo:

- Se ha due figli, eseguo lo scambio dei val tra di loro
- Se i figli esistono, continuo il controllo sugli eventuali figli

b)

```
def Exam3_8(r):
    if r.left and r.right: # se i figli esistono
        r.left.val, r.right.val = r.right.val, r.left.val # scambio i val tra di loro
    if r.left: # se il figlio SX esiste
        Exam3_8(r.left) # continuo il controllo sul figlio SX
    if r.right: # se il figlio DX esiste
        Exam3_8(r.right) # continuo il controllo sul figlio DX
```

c)

Il costo dell'algoritmo è  $\Theta(n)$  in quanto consiste in una semplice visita in preordine dell'albero

# Esame 13/1/22

giovedì 13 febbraio 2025 15:50

## Esercizio 1

funzione Exam( $n$ ):

```
tot ← n;  
if n <= 1: return tot;  
j ← 80;  
while j >= 3 do:  
    tot ← tot + j;  
    j ← j - 2;  
return tot+Exam(n - j)
```

- a) Si imposti la relazione di ricorrenza che ne definisce il tempo di esecuzione giustificando dettagliatamente l'equazione ottenuta.
- b) Si risolva la ricorrenza usando il metodo dell'albero dettagliando i passaggi del calcolo e giustificando ogni affermazione.

### Soluzione

a)

- linee 2-4: op. costanti  $\rightarrow \Theta(1)$
- linee 5-7: ciclo while, itera da 80 fino a 2 decrementando di 2 ad ogni iterazione. Viene iterato un num. costante di volte  $\rightarrow \Theta(1)$
- linea 8: chiamata ricorsiva Exam( $n-j$ ) dove  $j$  quando finisce il ciclo ha val. 2  $\rightarrow \text{Exam}(n-2) + \Theta(1)$
- Caso base: la ricorrenza finisce quando  $n \leq 1$  e si esegue un op. costante  $\rightarrow \Theta(1)$
- Eq. di ricorrenza:

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

b)

- Ad ogni livello abbiamo 1 solo nodo, poiché effettuiamo una singola chiamata ricorsiva
- Il contributo di ogni livello, è  $\Theta(1)$  (poiché c'è un solo nodo per livello)
- Si ferma quando  $n-2*k = 1 \Rightarrow k = \frac{n-1}{2} = \Theta(n)$
- Sommando i contributi dei vari livelli abbiamo un costo totale  $\Theta(n)$

## Esercizio 2

Abbiamo due array ordinati A e B di  $n$  interi distinti; si vuole sapere se esiste un valore  $x$  in A ed un valore  $y$  in B che differiscono al più 3 in valore assoluto (vale a dire  $|x - y| \leq 3$ ).

Ad esempio: per  $A = [1, 2, 20, 30]$  e  $B = [6, 7, 10]$  la risposta è negativa.

Per  $A = [1, 2, 9, 10, 12]$  e  $B = [6, 14, 16, 20]$  la risposta è positiva (grazie alla coppia (9, 6) o anche (12, 14)).

Progettare un algoritmo che risolva il problema restituendo 1 se la risposta è positiva, 0 altrimenti. Il costo computazionale dell'algoritmo deve essere asintoticamente strettamente inferiore a  $\Theta(n^2)$ .

Dell'algoritmo proposto

- a) si dia la descrizione a parole,
- b) si scriva lo pseudocodice
- c) si giustifichi il costo computazionale.

### Soluzione

a)

Creo due indici  $a$  e  $b$  che partono entrambi da 0 e utilizzero rispettivamente per A e B. Verifico se  $|A[a] - B[b]| \leq 3$ :

- Se è vero allora restituisco direttamente 1
- Se è falso allora incremento solo uno dei due indici:
  - Se  $A[a] < B[b]$  incremento  $a$
  - Se  $A[a] > B[b]$  incremento  $b$
  - Se  $A[a] = B[b]$  è impossibile perché  $A[a] - A[a] = 0$  che è  $\leq 3$

b)

```

def Exam2_13_1_22(A, B):
    a = b = 0 # inizializzo gli indici
    n = len(A)
    while a < n and b < n: # finchè non abbiamo scandito almeno un array
        if Abs(A[a] - B[b]) <= 3: # se la differenza è minore o uguale a 3
            return 1 # restituiamo 1
        if A[a] < B[b]: # se A[a] è minore di B[b]
            a += 1 # incrementiamo l'indice di A
        else: # altrimenti
            b += 1 # incrementiamo l'indice di B
    return 0 # se non abbiamo trovato val |A[a] - B[b]| <= 3 restituiamo 0

```

c)

Ogni iter. ha costo  $\Theta(1)$  e incrementa di 1 l'indice i o j. Quindi il costo del ciclo varia:

- da un minimo di n
- ad un massimo di  $2n$

Quindi il costo complessivo è  $O(n)$

### Esercizio 3

Si consideri una lista non vuota L, in cui ogni elemento `e un record a due campi, il campo **val** contenente un intero ed il campo **next** con il puntatore al nodo seguente (**next** vale **None** per l'ultimo record della lista).

Gli interi nella lista sono ordinati in modo non decrescente e bisogna eliminare dalla lista i record contenenti duplicati.

Si consideri ad esempio la lista L in figura; subito sotto viene riportato il risultato dell'operazione di cancellazione.



Progettare un **algoritmo iterativo** che, dato il puntatore r alla testa della lista effettui l'operazione di modifica in tempo  $\Theta(n)$  dove n è il numero 3 di elementi presenti nella lista. Lo spazio di lavoro dell'algoritmo deve essere  $O(1)$ .

Dell'algoritmo proposto

- si dia la descrizione a parole
- si scriva lo pseudocodice
- si giustifichi il costo computazionale.
- si scriva lo pseudocodice di un algoritmo ricorsivo che risolve il problema

### Soluzione

a)

Controlliamo ogni elem. finche non arriviamo alla fine della lista.

- Se esso ha lo stesso valore dell'elem. successivo allora dobbiamo eliminare l'elemento successivo, facendo in modo che il puntatore dell'elem. successivo punti all'elemento dopo ancora
- Altrimenti, se l'elemento corrente ha un valore diverso dall'elem. successivo continuiamo la ricerca sul prossimo elemento

b)

```

def Exam3_1_22_iter(r):
    p = r
    while p != None: # finche non arriviamo alla fine della lista
        if p.next.val == p.val:
            # se il val. del nodo successivo è uguale a quello del nodo corrente
            p.next = p.next.next # sostituiamo il puntatore al nodo successivo con quello dopo ancora
        else: # se il val. del nodo corrente è diverso da quello successivo
            p = p.next # continuiamo la ricerca

```

c)

Il costo varia in base al numero di iterazioni effettuate. Ad ogni iterazione o si avanza di un elem. o si cancella l'elemento successivo, quindi vengono eseguite esattamente n iterazioni

d)

```

def Exam3_1_22_ric(r):
    if r.next == None:
        return r
    p = Exam3_1_22_ric(r) # chiamata per il nodo successivo
    if r.val == p.val:
        # se il val. del nodo corrente è uguale a quello successivo
        r.next = p.next # saltiamo il nodo duplicato
    else: # se il val. del nodo corrente è diverso da quello successivo
        r.next = p # manteniamo il collegamento col nodo successivo
    return r

```

## Esercizio 1

funzione Exam( $n$ ):

```
tot ← n;  
if  $n \leq 1$ : return tot;  
 $j \leftarrow 512$ ;  
while  $j \geq 2$  do:  
     $k \leftarrow 0$ ;  
    while  $3 * k \leq n$  do:  $k \leftarrow k + 1$ ;  
    tot ← tot + Exam( $k$ );  
     $j \leftarrow j/2$ ;  
return tot
```

- a) Si imposta la relazione di ricorrenza che ne definisce il tempo di esecuzione giustificando dettagliatamente l'equazione ottenuta.
- b) Si risolva la ricorrenza usando il metodo principale (o un altro metodo, ricordando che  $\sum_{i=1}^k 3^i = \Theta(3^k)$ ) dettagliando i passaggi del calcolo e giustificando ogni affermazione.

### Soluzione

- Il ciclo while esterno viene eseguito un numero costante di volte (esattamente 9 volte)
- il ciclo while interno esegue  $\frac{n}{3}$  iterazioni, quindi ha costo  $\Theta(n)$ . A fine ciclo  $k = n/3$
- La chiamata ricorsiva Exam( $k$ ) viene eseguita nel ciclo esterno dopo la fine del ciclo interno  
Quindi viene eseguita 9 volte e quando  $k = n/3$
- La ricorrenza finisce quando  $n \leq 1$ .

Equazione ricorsiva:

$$T = \begin{cases} T(n) = 9T\left(\frac{n}{3}\right) + \Theta(n) \\ T(1) = \Theta(1) \end{cases}$$

Risoluzione col metodo principale:

- $a = 9, b = 3$
- $f(n) = \Theta(n)$
- $\Theta(n^{\log_3 9}) = \Theta(n^2)$
- Stiamo nel caso 1: Il costo è  $\Theta(n^2)$

## Esercizio 2

Sia A un array di dimensione  $n$  e B un array ordinato di dimensione  $m$ , contenenti entrambi numeri interi.

Si vuole trovare il numero di interi di A che sono presenti anche in B. Progettare un algoritmo ricorsivo che risolva il problema con un costo computazionale asintotico strettamente inferiore a  $\Theta(nm)$ .

Ad esempio: per  $A = [8, 1, 2, 12, 10, 11, 20, 2, 8]$  e  $B = [3, 3, 4, 8, 10, 10, 13, 20, 21, 22]$  l'algoritmo deve restituire 4 (i numeri di A che sono anche in B sono infatti 8, 10, 20, 8).

Dell'algoritmo proposto

- a) si dia la descrizione a parole
- b) si scriva lo pseudocodice
- c) si giustifichi il costo computazionale

### Soluzione

- a)

Per ogni elem. di A posso controllare se è contenuto in B tramite la ricerca binaria (poiché B è ordinato). Per controllare quanti numeri sono presenti in entrambi gli array posso usare un contatore, che verrà incrementato ad ogni val. trovato in B. Dopo aver controllato tutti gli elementi di A restituiamo il contatore.

- b)

```

def Exam2_31_1_22(A, B):
    i, n = 0, len(A)
    cont = 0 # inizializziamo il contatore
    while i < n:
        x = A[i] # prendiamo l'elem. corrente
        if Ric_Bin(B, x) != -1: # se l'elem è stato trovato
            cont += 1 # aumentiamo il contatore
    return cont

```

dove con `Ric_Bin(B, x)` indichiamo la funz. di ricerca binaria che cerca `x` in `B` e restituisce la posizione se viene trovato, -1 altrimenti.

c)

Il costo dipende dal ciclo while, che esegue `n` iterazioni e la ricerca binaria, che ha costo  $\log(m)$ , viene effettuata per ogni elemento di `A`.

Quindi il costo sarà  $n * O(\log(m)) = O(n * \log(m))$  che è inferiore a  $\Theta(n * m)$

### Esercizio 3

Si consideri una lista `L`, in cui ogni elemento `e un record a due campi, il campo **val** contenente un intero ed il campo **next** con il puntatore al nodo seguente (**next** vale **None** per l'ultimo record della lista).

Bisogna contare i record della lista contenenti numeri pari.

Si consideri ad esempio la lista `L`, per questa lista bisogna la risposta è 6



Progettare un algoritmo ricorsivo che, dato il puntatore `r` alla testa della lista effettui l'operazione di conteggio in tempo  $\Theta(n)$  dove `n` è il numero di elementi presenti nella lista.

Dell'algoritmo proposto

- si dia la descrizione a parole
- si scriva lo pseudocodice
- si giustifichi il costo computazionale risolvendo la ricorsione che viene fuori dall'algoritmo utilizzando uno dei metodi di soluzione visti a lezione.

### Soluzione

a)

Abbiamo 3 casi:

- Se la lista è vuota allora ritorno 0
- Se il valore dell'elem. corrente è dispari allora posso continuare la ricerca negli altri elementi
- Se invece il val. è pari allora continuo la ricerca ma aggiungo 1 incrementare il conto dei num. pari

b)

```

def Exam3_31_1_22(r):
    if r == None: # se abbiamo finito la lista o essa è vuota
        return 0 # ritorniamo 0
    if r.val%2 == 1: # se il val. corrente è dispari
        return Exam3_31_1_22(r.next) # continuiamo la ricerca
    return Exam3_31_1_22(r.next) + 1 # se è pari continuiamo la ricerca aggiungendo 1

```

c)

La chiamata ricorsiva viene effettuata su una lista con un elem. in meno, quindi:

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

Posso risolvere usando il metodo iterativo.

Sviluppo  $T(n)$ :

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

Generalizzo:

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

Continuo finchè  $n - k = 0$  quindi  $n = k$

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

# Esame 31/3/22

giovedì 13 febbraio 2025 19:44

## Esercizio 1

funzione Exam( $n$ ):

```
tot ← 1;
if n <= 1: return tot;
j ← 63;
while j > 0 do:
    k ← 0;
    while 3 * k <= n do: k ← k + 1;
    tot ← tot + 2 * Exam(k);
    j ← j - 7;
while k > 0 do:
    for i = 1 to n do: tot ← tot - 1
    k ← k - 1;
return tot
```

- a) Si imposti la relazione di ricorrenza che ne definisce il tempo di esecuzione giustificando dettagliatamente l'equazione ottenuta.  
b) Si risolva la ricorrenza usando il **metodo dell'albero** dettagliando i passaggi del calcolo e giustificando ogni affermazione.

## Soluzione

a)

- il primo ciclo while esterno viene iterato un numero costante di volte (esattamente 9).
- Il ciclo while interno invece varia in base a  $n$ , itera da 0 a  $n/3$ , quindi ha fine ciclo  $k = n/3$  e ha costo  $\Theta(n)$   
Costo:  $9 * \Theta(n) = \Theta(n)$   
Viene poi eseguita una chiamata ricorsiva all'interno del ciclo esterno dopo la fine del ciclo interno.  
Quindi la chiamata viene effettuata 9 volte con  $k = n/3$ .
- L'ultimo while invece varia in base al valore di  $k$  che parte da  $n/3$  fino ad arrivare a 0  
All'interno del ciclo vi è un ciclo for che itera da 1 a  $n$  che richiede tempo  $\Theta(n)$   
Costo:  $\Theta(n^2)$
- La ricorsione termina quando  $n \leq 1$
- Equazione di ricorrenza:

$$T = \begin{cases} T(n) = 9T\left(\frac{n}{3}\right) + \Theta(n^2) \\ T(1) = \Theta(1) \end{cases}$$

b)

- La chiamata viene effettuata 9 volte quindi ad ogni livello i avremmo  $9^i$  nodi.
- Un nodo al livello  $i$  avrà costo  $\left(\frac{n}{3^i}\right)^2 = \frac{n^2}{9^i}$
- Il contributo di ogni livello sarà  $9^i * \frac{n^2}{9^i} = n^2 = \Theta(n^2)$
- L'albero è completo ed ogni nodo ha 9 figli quindi la sua altezza è logaritmica in  $n$ .

Quindi il costo totale è  $\Theta(n^2 * \log(n))$

## Esercizio 2

Dato un array **ordinato** A di  $n$  interi ed un intero  $k$  vogliamo sapere quante coppie in A hanno somma  $k$ .

Si progetti un algoritmo **iterativo** che risolva il problema in tempo  $\Theta(n)$ .

Ad esempio:

- se  $A = [1, 2, 2, 3, 4, 5, 5, 5, 8, 9, 9]$  e  $k = 7$  l'algoritmo deve restituire 7 (le coppie a somma 7 sono infatti (1, 5), (1, 6), (1, 7), (2, 5), (2, 6), (2, 7) e (3, 4)).
- se  $A = [1, 5, 5, 5, 9]$  e  $k = 10$  l'algoritmo deve restituire 4 (le coppie a somma 10 sono infatti (0, 4), (1, 2), (1, 3), (2, 3)).

Dell'algoritmo proposto:

- si dia la descrizione a parole
- si scriva lo pseudocodice
- si giustifichi il costo computazionale

## Soluzione

Utilizzo un contatore per contare le coppie e due indici **I** e **r** che partono rispettivamente dal primo e dall'ultimo elem. dell'array.

Eseguo un controllo incrociato sulla lista per trovare le coppie valide, se la somma degli elementi puntati è:

- superiore a k**, la somma è troppo grande quindi **decremento r** per trovare un num. più piccolo
- inferiore a k**, la somma è troppo piccola quindi **incremento I** per trovare un num. più grande
- uguale a k**, abbiamo trovato la coppia valida però dobbiamo **gestire le occorrenze** di  $A[I]$  e  $A[r]$  perché non sono elementi distinti

Se troviamo una coppia, **aumentiamo sia I che r** e teniamo conto dei **passi** con cui entrambi **avanzano** con le var. **I\_cont** e **r\_cont**.

Se gli elem. di  $A[I]$  e  $A[r]$ :

- Sono **uguali** ( $A[I] = A[r]$ ) allora il num. di coppie valide è il **num. di modi in cui possiamo scegliere 2 elem. tra I\_cont e r\_cont**:

$$\binom{l_{cont} + r_{cont}}{2} = \frac{(l_{cont} + r_{cont}) * (l_{cont} + r_{cont} - 1)}{2}$$

Perché ogni coppia è formata da due elem. della stessa sequenza

- Sono **diversi** ( $A[l] \neq A[r]$ ), il num. di coppie valide è dato da:

$$cont += l_{cont} * r_{cont}$$

Perché ogni copia  $A[l]$  può formare una coppia con ogni copia di  $A[r]$

L'algoritmo si ferma quando l'indice l raggiunge l'indice r.

```
def Exam2_31_3_22(A, k):
    l, r = 0, len(A)-1 # inizializzo gli indici Sx e Dx
    cont = 0 # inizializzo il contatore delle coppie
    while l < r: # finchè l'indice Sx non raggiunge quello Dx
        x = A[l] + A[r] # eseguo la somma degli elem. puntati
        if x > k: # se è più grande dell'elem. richiesto
            r -= 1 # cerco un num. più piccolo
        if x < k: # se è più piccola dell'elem. richiesto
            l += 1 # cerco un num. più grande
        if x == k: # se è uguale a k
            lc, rc = 1, 1 # inizializzo i contatori dei passi degli indici
            while l+1 < r and A[l+1] == A[r-1]:
                # finchè gli elem. successivi a l sono uguali e l'ind. Sx non supera quello Dx
                l += 1 # incremento l'indice Sx per controllare i prossimi elem.
                lc += 1 # incremento il contatore dei passi Sx
            while r-1 > l and A[r-1] == A[l]:
                # finchè gli elem. successivi a r sono uguali e l'ind. Dx non supera quello Sx
                r -= 1 # decremento l'indice Dx per controllare i prossimi elem.
                rc += 1 # incremento il contatore dei passi Dx
            if A[l] == A[r]: # se i due elem. sono uguali
                cont += ((lc+rc)*(lc+rc-1))//2 # sommo al contatore i modi per scegliere 2 elem. tra lc e rc
            else: # se i due elem. sono diversi
                cont += lc*rc # sommo al contatore il loro prodotto
            l += 1 # incremento l'indice Sx per andare avanti sugli elem.
            r -= 1 # decremento l'indice Dx per andare avanti sugli elem.
    return cont
```

Il while esterno e interni vengono eseguiti sempre n volte, poiché se tutti gli elem. sono coppie valide gli indici si incontrano a metà, invece se sono tutti uguali uno dei due indici arriverà a fine array.

Quindi ogni elem. viene visitato unna sola volta, quindi il costo è  $\Theta(n)$

### Esercizio 3

Si consideri una lista a puntatori L, in cui ogni elemento `e un record a tre campi: il campo **val** contenente un **bit** (cio`è un valore 0 o 1), il campo **next** con il puntatore al nodo seguente (**next** vale **None** per l'ultimo record della lista) ed il campo **prec** con il puntatore al nodo precedente (**prec** vale **None** per il primo record della lista).

Bisogna verificare se la stringa che si ottiene considerando i bit dei vari nodi della lista `e palindroma.

Ad esempio, se la lista L in input `e quella di sinistra nella figura che segue, la risposta `e NO (la stringa binaria 010110 non `e palindroma, mentre se L `e la lista di destra la risposta `e SI (la stringa binaria 110111 `e palindroma).



Progettare un algoritmo (iterativo o ricorsivo) che, dato il puntatore s alla testa della lista, risolve il problema in tempo  $\Theta(n)$ , dove n `e il numero di nodi della lista a puntatori.

Lo spazio di lavoro dell'algoritmo proposto deve essere  $O(1)$  (in altri termini NON `e possibile definire e utilizzare altre liste).

Dell'algoritmo proposto:

- si dia la descrizione a parole
- si scriva lo pseudocodice
- si giustifichi il costo computazionale.

### Soluzione

Utilizzo due puntatori, uno che parte dal primo elem. (s) e si sposta verso Dx e uno che parte dall'ultimo elem. (d) e si sposta verso Sx.

Per ricavare la pos del puntatore d scorriamo tutta la lista fino ad arrivare all'ultimo elem, quindi s e d saranno diametralmente opposti.

Se s e d sono uguali, quindi puntano entrambi allo stesso elem. allora c'è un solo elem. nella lista (quindi è un palindromo)

Se non hanno lo stesso val. allora non `e un palindromo poiché la stringa non `e simmetrica

Altrimenti se sono uguali, controllo se il prossimo puntatore di s è uguale a d (quindi abbiamo controllato tutti gli elem.) e in quel caso abbiamo trovato un palindromo.

Altrimenti se il prossimo puntatore di s non è uguale a d sposto s e d al loro elem. successivo per continuare il controllo.

```

def Exam3_31_3_22(s):
    d = s
    while d.next != None: # andiamo alla fine della lista
        d = d.next
    # ora d punta all'ultimo elem. nella lista
    # e i puntatori sono diametralmente opposti
    while True: # finchè non finisce l'algoritmo
        if s == d: # se la lista è formata da un elem.
            return "SI" # allora è un palindromo
        if s.val != d.val: # se gli elem. sono diversi
            return "NO" # allora non è un palindromo
        if s.next == d: # se i due puntatori si incontrano
            return "SI" # allora è un palindromo
        s = s.next # altrimenti continuo il controllo
        d = d.prev

```

il ciclo while viene eseguito n volte e nel secondo while, ad ogni iterazione s e d si avvicinano, quindi dopo  $n/2$  iter. finisce il ciclo.  
Quindi il costo è  $\Theta(n) + \Theta(n) = \Theta(n)$

# Esame 6/22

venerdì 14 febbraio 2025 11:23

## Esercizio 1

Per la soluzione di un certo problema disponiamo di un algoritmo iterativo con costo computazionale  $\Theta(n^2)$ . Ci viene proposto in alternativa un algoritmo ricorsivo il cui costo è catturato dalla seguente ricorrenza:

$$T = \begin{cases} T(n) = a * T\left(\frac{n}{4}\right) + \Theta(1) & \text{se } n \geq 4 \\ T(n) = \Theta(1) & \text{se } n < 4 \end{cases}$$

Dove a una certa costante intera positiva con  $a \geq 2$

Determinare qual è il valore massimo che la costante intera a può avere perché l'algoritmo ricorsivo risulti asintoticamente più efficiente dell'algoritmo iterativo di cui disponiamo. **Motivate bene la vostra risposta.**

### Soluzione

- Cominciamo col risolvere la ricorrenza inserendo y al posto di a:

Applicando il metodo principale abbiamo  $f(n) = \Theta(1)$  e  $n^{\log_4(a)} \geq n^{\log_4(2)} = n^{\frac{1}{2}}$  si ha quindi  $f(n) = O(n^{\log_4(a-\varepsilon)})$   
Siamo quindi nel **primo caso** e il costo è  $\Theta(n^{\log_4(a)})$

- Troviamo il massimo di a

Se  $a = 16$  ( $4^2$ ) la ricorrenza ha soluzione  $\Theta(n^{\log_4(16)}) = \Theta(n^2)$  quindi perché l'algoritmo ricorsivo abbia costo inferiore bisogna avere  $a \leq 15$

## Esercizio 2

Sia A un array di n interi. Con la coppia ordinata (i, j),  $0 \leq i \leq j < n$ , rappresentiamo il suo sottoarray che parte dall'elemento in posizione i e termina con l'elemento in posizione j, definiamo valore di un sottoarray come la somma dei suoi elementi.

Progettare un algoritmo che, dato un array A di interi positivi ed un intero positivo s, restituisce la coppia ordinata che rappresenta il sottoarray di A più a sinistra che ha valore s. Se un tale sottoarray non esiste, la funzione deve restituire None. L'algoritmo deve avere costo computazionale  $O(n)$ .

Ad esempio, per  $A = [1, 3, 5, 2, 9, 3, 3, 1, 6]$

- con  $s = 7$  l'algoritmo deve restituire la coppia (2, 3) (ci sono infatti in A tre sottoarray con valore 7 le cui coppie nell'ordine da sinistra a destra sono (2, 3), (5, 7), (7, 8)).
- con  $s = 21$  l'algoritmo deve restituire None in quanto A non ha sottoarray con valore 21 .

Dell'algoritmo proposto:

- si dia la descrizione a parole
- si scriva lo pseudocodice
- si giustifichi il costo computazionale.

### Soluzione

Utilizziamo gli indici i e j per indicare i due massimi del sottoarray (entrambi inizializzati a 0) e una var. tot per indicare la somma dei val. del sottoarray. Incrementiamo gli indici con 3 possibili casi:

- Se tot < s, dobbiamo continuare a cercare un val. che dia s come somma. Quindi incrementiamo j e aggiungiamo A[j] al tot
- Se tot = s, abbiamo trovato il nostro sottoarray, quindi ritorniamo i due indici
- Se tot > s, abbiamo un sottoarray troppo grande. Quindi incrementiamo i e rimuoviamo A[i-1] da tot.

Se abbiamo controllato tutti gli elem. (j ha superato len(A)) allora ritorniamo None

```
def Exam2_6_22(A, s):  
    i, j, tot = 0, 0, 0 # inizializziamo gli indici e tot  
    while j < len(A): # finché non finisce l'array  
        tot += A[j] # aggiungiamo gli elem. alla somma  
        while tot > s: # finché il totale è maggiore di s  
            i += 1 # incrementiamo l'indice Sx  
            tot -= A[i-1] # e rimuoviamo gli elem. precedenti  
        if tot == s: # se il totale è uguale a s  
            return i, j # ritorniamo gli indici  
        j += 1 # controlliamo il prossimo elem.  
    return None
```

Il costo dipende dal num. di iterazioni del while. Nel while interno ad ogni iter. si incrementa i e nel while esterno ad ogni iter. si incrementa j, poiché  $i \leq j < n$  quindi ogni elem. dell'array può essere visitato al massimo una volta da i e una volta da j. Quindi il num. tot di incrementi di i e j è al massimo  $2n$ , quindi il costo è  $\Theta(n)$

## Esercizio 3

Si consideri una lista concatenata dove ogni nodo ha 2 campi, il campo key contenente un intero ed il campo next con il puntatore al nodo seguente (next vale None per l'ultimo nodo della lista).

Bisogna aggiornare i puntatori della lista in modo da creare una nuova lista priva dei nodi con valore superiore a 10 e in cui i nodi rimanenti appaiono in ordine inverso rispetto all'originale.

Ad esempio per la lista di seguito a sinistra la funzione deve restituire la lista di seguito a destra:



Progettare un algoritmo che, dato il puntatore  $p$  alla testa della lista, risolve il problema in tempo  $\Theta(n)$  dove  $n$  è il numero di nodi della lista originaria. Lo spazio di lavoro dell'algoritmo proposto deve essere  $\Theta(1)$  (in altri termini non è possibile definire e utilizzare altre liste o nodi).

Dell'algoritmo proposto:

- a) si dia la descrizione a parole
- b) si scriva lo pseudocodice,
- c) si giustifichi il costo computazionale

### Soluzione

Uso un puntatore  $q$  per puntare alla lista da restituire. Scorro la lista e ad ogni nodo se il suo val è  $> 10$  allora lo salto, altrimenti aggiungo il nodo in testa alla lista  $q$ . Per aggiungere un nodo alla nuova lista devo prima salvare temporaneamente il suo prossimo nodo (per continuare il controllo), aggiungere il nodo in testa alla lista (aggiornando il suo prossimo nodo a quello della testa di  $q$ ), spostare il puntatore di  $q$  al nodo corrente e poi continuare il controllo sul prossimo nodo salvato in precedenza.

```
def Exam3_6_22(p):
    q = None
    while p != None: # finche non finisco gli elem.
        if p.key > 10: # se il val è > 10
            p = p.next # lo salto
        else: # se è un nodo che ci serve
            tmp = p.next # salvo temporaneamente il suo prossimo nodo
            p.next = q # modifico il puntatore del nodo per indicare alla testa di q
            q = p # aggiorno il puntatore di q per indicare alla nuova testa
            p = tmp # continuo il controllo sul prossimo nodo
    return q
```

Ogni nodo viene visitato una sola volta, quindi il costo è  $\Theta(n)$

## Esercizio 1

```
def Exam(A, n):
    b=1
    if n <= 2:  return b
    i = 1
    while i <= 8:
        b=b*Exam(A, n//2)
        i+=1
    for i in range(n-1):
        A[i]+=A[i+1]
    return b
```

che viene richiamata la prima volta cos'è: Exam(A, len(A)).

- Si imposta la relazione di ricorrenza che ne definisce il tempo di esecuzione giustificando dettagliatamente l'equazione ottenuta.
- Si risolve la ricorrenza usando due metodi a scelta, dettagliando i passaggi del calcolo e giustificando ogni affermazione.

### Soluzione

a)

- Il primo ciclo while viene eseguito un numero costante di volte (esattamente 8 volte).  
Nel while viene chiamata ricorsivamente la funzione con l'array originale ma con n dimezzato.  
La chiamata ricorsiva viene effettuata 8 volte quindi il while ha costo  $8T(n/2) + \Theta(1)$
- Il ciclo for viene eseguito da 8 a n-1 volte e nel suo interno viene effettuata una op. costante  
Quindi il suo costo sarà  $\Theta(n)$
- La chiamta ricorsiva si ferma quando  $n \leq 2$

L'equazione di ricorrenza sarà:

$$T = \begin{cases} T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n) & \text{se } n > 2 \\ T(n) = \Theta(1) & \text{altrimenti} \end{cases}$$

b)

Risolvendo col metodo principale abbiamo:

- $a = 8, b = 2$
- $f(n) = \Theta(n)$
- $n^{\log_2(8)} = \Theta(n^3)$
- Ci troviamo nel primo caso quindi il costo sarà  $\Theta(n^3)$

Risolvendo col metodo iterativo abbiamo:

- $8T\left(\frac{n}{2}\right) + \Theta(n)$
- $8\left(8T\left(\frac{n}{4}\right) + \Theta\left(\frac{n}{2}\right)\right) + \Theta(n) = 8^2T\left(\frac{n}{2^2}\right) + 8\Theta\left(\frac{n}{2}\right) + \Theta(n)$
- $8\left(8\left(8T\left(\frac{n}{8}\right) + \Theta\left(\frac{n}{4}\right)\right) + \Theta\left(\frac{n}{2}\right)\right) + \Theta(n) = 8^3T\left(\frac{n}{2^3}\right) + 8^2\Theta\left(\frac{n}{2^2}\right) + 8\Theta\left(\frac{n}{2}\right) + \Theta(n)$

Generalizzo:

$$\bullet 8^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 8^i \Theta\left(\frac{n}{2^i}\right) = 8^k T\left(\frac{n}{2^k}\right) + n * \sum_{i=0}^{k-1} \Theta(4^i)$$

$$2^{2 \log(n)} = 2^{\log(n^2)} = n^2$$

$$8^{\log(n)} = 2^{3 \log(n)} = 2^{\log(n^3)} = n^3$$

oppure

$$8^{\log(n)} = n^{\log(8)} = n^3$$

Mi fermo quando  $\frac{n}{2^k} = 1 \Rightarrow k = \log(n)$  quindi

$$8^{\log(n)} T(1) + n * \sum_{i=0}^{\log(n)-1} \Theta(4^i) = n^3 * \Theta(1) + n * \Theta\left(\frac{4^{\log(n)} - 1}{4 - 1}\right) = n^3 \Theta(1) + n * \Theta(n^2) = \Theta(n^3)$$

Per provare ho risolto anche col metodo dell'albero ma non è richiesto dall'esercizio

- Ad ogni liv. i abbiamo  $8^i$  nodi
- Ogni nodo ha un costo  $\frac{n}{2^i}$
- Il contributo di ogni liv. è  $8^i * \frac{n}{2^i} = 4^i * n$
- Ci fermiamo quando  $\frac{n}{2^i} = 1 \Rightarrow i = \log(n)$ , quindi

$$\sum_{i=0}^{\log(n)} 4^i * n = n * \sum_{i=0}^{\log(n)} 4^i = n * \left( \frac{4^{\log(n)} - 1}{4 - 1} \right) = n * \frac{2^{2\log(n)} - 1}{3} = n * \frac{n^2 - 1}{3} = \frac{n^3}{3} - \frac{n}{3} = \Theta(n^3)$$

## Esercizio 2

Un array A ordinato di  $n > 1$  interi distinti ha subito una rotazione di  $k$  posizioni verso sinistra,  $1 \leq k < n$ .

Ad esempio, per  $A = [5, 7, 9, 2, 3]$  il valore di  $k$  è 2 mentre per  $A = [9, 2, 3, 5, 7]$  è 4.

Progettare un'algoritmo che, dato l'array A, in tempo  $O(\log n)$  restituisca il valore di  $k$ .

Dell'algoritmo proposto:

- a) si dia la descrizione a parole
- b) si scriva lo pseudocodice
- c) si giustifichi il costo computazionale.

### Soluzione

Dopo una rotazione  $k$ , il val. minimo dell'array si trova nella posizione  $n - k$ , quindi il val. massimo calcolo poi la grandezza dell'array meno l'indice trovato ( $-1$  perché parte da 0) che mi dà come risultato il num. degli elem. che hanno subito la rotazione.

Con la ricerca binaria trovo l'indice del valore più piccolo nell'array. Poiché l'array è diviso in due parti devo trovare il punto in cui la sequenza cambia. Sia  $m$  la pos. in mezzo dell'array:

- Se  $A[m-1] > A[m]$ : allora siamo nel punto in cui si incontrano le due parti, quindi  $m$  sarà il val. max e  $m+1$  il val. min, in questo caso ritorno il calcolo di  $\text{len}(A) - m$
- Altrimenti continuo la ricerca:
  - Se  $A[s] < A[m]$ : se stiamo ancora nella parte sinistra (crescente), allora cerchiamo nella parte dopo il punto medio ( $A[m+1:]$ )
  - Altrimenti abbiamo superato il val. minimo e dobbiamo cercare nella parte precedente al punto medio ( $A[:m]$ )

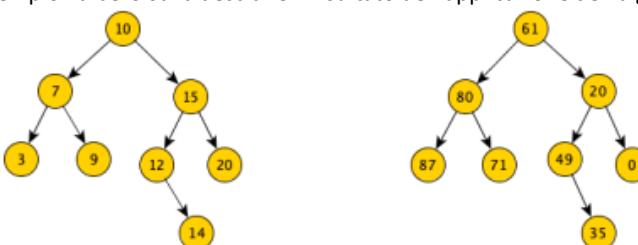
```
def Exam2_27_6_22(A):
    s = 0
    d = len(A) - 1
    while True:
        m = (s+d)//2
        if A[m-1] > A[m]: # abbiamo trovato il punto min
            return len(A) - m
        if A[s] < A[m]: # ancora dobbiamo trovare il punto min
            s = m+1 # ricontrolliamo da [m+1, d]
        else: # abbiamo superato il punto min
            d = m # ricontrolliamo da [s, m]
```

Ad ogni iter. del ciclo, l'intervallo di ricerca si dimezza, quindi dopo al più  $\log(n)$  iter. verrà trovato il val. richiesto e l'algoritmo termina. Quindi il costo è  $O(\log(n))$

## Esercizio 3

Progettare un algoritmo che, dato il puntatore alla radice di un albero binario di ricerca T, modifica il valore di ciascun nodo di T in modo che il nuovo valore del nodo risulti la somma di tutte le chiavi (che, in quanto tali, sono tutte distinte) che in T avevano un valore maggiore della sua chiave originaria.

Ad esempio l'albero sulla destra è il risultato dell'applicazione dell'algoritmo sull'albero binario di ricerca T riportato a sinistra.



Il costo computazionale dell'algoritmo proposto deve essere  $\Theta(n)$  dove  $n$  è il numero di nodi dell'albero.

Dell'algoritmo proposto:

- a) si dia la descrizione a parole
- b) si scriva lo pseudocodice
- c) si giustifichi il costo computazionale.

### Soluzione

Poiché è un albero di ricerca, per ogni nodo posso sommare tutti i nodi alla sua destra (che sia figlio, padre, sottoalbero del padre, e così via).

Per fare in modo di non controllare più volte l'albero (ad esempio se sono nel nodo 7 dovrei risalire al padre 10 e controllare il suo sottoalbero destro, cosa che poi dovrò rifare quando sono nel nodo 3, aggiungendo però anche il nodo 7 e 9), possiamo eseguire una visita inordine però controllando prima il nodo destro, ritorniamo la somma del sottoalbero destro, modifichiamo la chiave del nodo in base al val. della somma e poi controlliamo il nodo sinistro.

```
def Exam3_27_6_22(p):
    if p: # se p è un nodo
        Exam3_27_6_22(p.right) # controlliamo il sottoalbero destro
        global somma # prendiamo il val. somma aggiornato
        somma = somma + p.key # aggiungiamo il val. del nodo corrente
        p.key = somma - p.key # modifichiamo la chiave del nodo
        Exam3_27_6_22(p.left) # e controlliamo poi il sottoalbero sinistro
```

Il costo è quello di una visita dell'albero quindi  $\Theta(n)$  (poiché ogni nodo viene visitato una sola volta)

## Esercizio 1

Si supponga di avere un algoritmo speciale in grado di eseguire la fusione di due sottoarray ordinati di  $n/2$  elementi ciascuno in  $O(\sqrt{n})$  operazioni. Quanto sarebbe, in questo caso, il costo computazionale dell'algoritmo di Merge Sort?

- a) Si imposti la relazione di ricorrenza che definisce il tempo di esecuzione giustificando dettagliatamente l'equazione ottenuta.
- b) Si risolva la ricorrenza usando due metodi a scelta, dettagliando i passaggi del calcolo e giustificando ogni affermazione.

### Soluzione

$$T = \begin{cases} 2T\left(\frac{n}{2}\right) + O(\sqrt{n}) \\ T(1) = \Theta(1) \end{cases}$$

Risolvendo col metodo principale abbiamo

- $a = 2, b = 2$
- $f(n) = O(\sqrt{n})$
- $n^{\log_2(2)} = \Theta(n)$
- Siamo nel caso 1: quindi il costo è  $\Theta(n)$

Risolvendo col metodo iterativo abbiamo

- $2T\left(\frac{n}{2}\right) + \sqrt{n}$
- $2\left(2T\left(\frac{n}{4}\right) + O\left(\sqrt{\frac{n}{2}}\right)\right) + O(\sqrt{n}) = 4T\left(\frac{n}{4}\right) + 2O\left(\sqrt{\frac{n}{2}}\right) + O(\sqrt{n})$
- $2\left(2\left(2T\left(\frac{n}{8}\right) + O\left(\sqrt{\frac{n}{4}}\right)\right) + O\left(\sqrt{\frac{n}{2}}\right)\right) + O(\sqrt{n}) = 8T\left(\frac{n}{8}\right) + 4O\left(\sqrt{\frac{n}{4}}\right) + 2O\left(\sqrt{\frac{n}{2}}\right) + O(\sqrt{n})$

Generalizzo

$$2^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 2^i O\left(\sqrt{\frac{n}{2^i}}\right) = 2^k T\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} O\left(2^i * \frac{\sqrt{n}}{\sqrt{2^i}}\right) = 2^k T\left(\frac{n}{2^k}\right) + O\left(\sqrt{n} * \sum_{i=0}^{k-1} \sqrt{2^i}\right)$$

Mi fermo quando  $\frac{n}{2^k} = 1 \Rightarrow k = \log(n)$ , quindi:

$$2^{\log(n)} T(1) + O\left(\sqrt{n} * \sum_{i=0}^{\log(n)-1} \sqrt{2^i}\right) = n * \Theta(1) + \sqrt{n} * O(\sqrt{n}) = \Theta(n)$$

## Esercizio 2

Sia dato un array A contenente n interi distinti e ordinati in modo crescente. Progettare un algoritmo che, in tempo  $O(\log n)$ , individui la posizione più a sinistra nell'array per cui si ha  $A[i] \neq i$ , l'algoritmo restituisce  $-1$  se una tale posizione non esiste.

Ad esempio, per  $A = [0, 1, 2, 3, 4]$  l'algoritmo deve restituire  $-1$ , per  $A = [0, 5, 6, 20, 30]$  la risposta deve essere  $1$  e per  $A = [-3, 1, 2, 3, 6]$  la risposta deve essere  $0$ .

Dell'algoritmo proposto:

- a) si dia la descrizione a parole
- b) si scriva lo pseudocodice
- c) si giustifichi il costo computazionale

### Soluzione

Possiamo usare una versione modificata della ricerca binaria. Per prima cosa controlliamo se il primo elem. ha val. diverso da 0. Se si allora restituiamo quel valore, altrimenti abbiamo che  $A[0] = 0$  quindi i numeri successivi sono positivi.

Iniziamo la ricerca binaria controllando se il val. a sinistra del sottoarray è valido altrimenti controlliamo il valore centrale m:

- Se  $A[m] < m$ , impossibile perché l'array in questo caso partirà da 0 e sono valori distinti, quindi nel caso peggiore ogni  $A[i] = i$
- Se  $A[m] = m$ , allora dobbiamo continuare la ricerca nella parte successiva dell'array ( $A[m:d]$ )
- Se  $A[m] > m$ , allora dobbiamo vedere se magari vi è un val valido più a sinistra tenendo compresa la posizione trovata( $A[s:m]$ )

```

def Exam2_15_9_22(A):
    if A[0] != 0: # se il primo val è diverso da 0
        return 0 # ritorniamo l'indice del primo val
    s, d = 0, len(A)-1 # altrimenti cerchiamo nell'array
    while s <= d: # finchè l'indice Sx non supera quello Dx
        if A[s] != s: # se l'elem. a sinistra è diverso dal suo indice
            return s # allora restituiamo il suo indice
        m = (s+d)//2 # altrimenti calcoliamo il punto medio del sottoarray
        if A[m] == m: # se il val. a metà è uguale all'indice
            s = m+1 # dobbiamo continuare la ricerca più avanti
        else: # altrimenti dobbiamo vedere se c'è un val. valido più piccolo
            d = m
    return -1 # se non abbiamo trovato un val. valido restituiamo -1

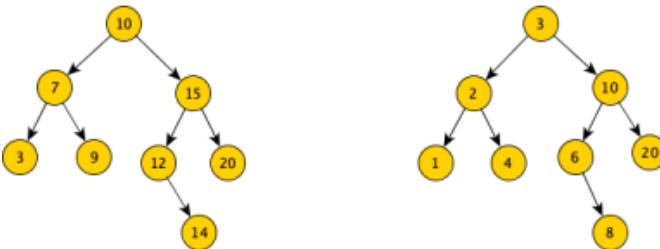
```

Come nella ricerca binaria classica, ad ogni iterazione del ciclo, la grandezza del sottoarray viene dimezzata quindi il costo è  $O(\log(n))$

### Esercizio 3

Progettare un algoritmo che, dato il puntatore alla radice di un albero binario T avente per chiavi degli interi, verifica se l'albero è un albero binario di ricerca.

Ad esempio, l'algoritmo per l'albero sulla sinistra deve restituire True mentre per l'albero sulla destra deve restituire False (infatti nel sottoalbero di sinistra del nodo con chiave 3 c'è presente un nodo con chiave 4)



Il costo computazionale dell'algoritmo proposto deve essere  $\Theta(n)$  dove  $n$  è il numero di nodi dell'albero.

Dell'algoritmo proposto

- si dia la descrizione a parole
- si scriva lo pseudocodice
- si giustifichi il costo computazionale

### Soluzione

Per controllare se è un albero di ricerca per ogni nodo  $x$  devo vedere se tutti gli elem. nel sottoalbero a sinistra siano  $< x$  e se tutti gli elem. nel sottoalbero a destra siano  $> x$ .

Per fare ciò posso:

- Eseguire una visita inordine e inserire ogni elem. in un array
- Scorrere l'array e se è crescente allora restituisco True, altrimenti False

```

def Exam3_15_9_22_1(p):
    A = [] # inizializziamo l'array
    Visita_inordine(p, A) # eseguiamo la visita inordine
    for i in range(1, len(A)):
        if A[i-1] >= A[i]: # se il val. precedente è più grande o uguale al corrente
            return False # allora non è un albero binario di ricerca
    return True # Se non ci sono stati intoppi restituiamo True

def Visita_inordine(p, A):
    if p != None: # se è un nodo
        Visita_inordine(p.left, A) # aggiungiamo gli elem. sinistri
        A.append(p.key) # poi il nodo corrente
        Visita_inordine(p.right, A) # e poi quelli destri

```

Un altro modo per progettare l'algoritmo senza l'uso di un array ausiliario, è di effettuare una visita postordine e restituire ad ogni nodo due val. che rappresentano l'intervallo dei val. contenuti nei suoi sottoalberi.

È un albero di ricerca se la chiave di ogni nodo è  $>$  dell'intervallo del figlio sinistro e  $<$  dell'intervallo restituito dal figlio destro.

Il costo è quello di una visita dell'albero, quindi  $\Theta(n)$ , e di un ciclo sull'array di  $n$  elem., quindi anche esso  $O(n)$ . Costo tot.  $\Theta(n)$   
L'eq. di ricorrenza della visita è:

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

dove  $k$  è il num. di nodi del sottoalbero sinistro della radice, e  $n-k-1$  il num. di nodi di quello destro.

Come abbiamo già visto nella sezione delle visite dell'albero il costo è  $\Theta(n)$

# Esame 25/10/22

mercoledì 12 febbraio 2025 18:01

## Esercizio 1

```
def es1(n):
    if n <= 1:  return 5
    a = es1(n//2)
    i = j = 1
    while j < n:
        j*= 2
        i+= 1
    u, j = 1, n
    while j > 1:
        j-= i
        u+= 1
    return a + es1(n//2) + u
```

- a) Si imposti la relazione di ricorrenza che ne definisce il tempo di esecuzione giustificando dettagliatamente l'equazione ottenuta.
- b) Qualora sia possibile, risolvere la ricorrenza utilizzando il teorema principale dettagliando il caso del teorema ed i passaggi logici. Se il teorema principale non è applicabile spiegarne il motivo.

### Soluzione

- Il primo ciclo while viene eseguito da  $j=1$  fino ad  $n$  e ad ogni iter.  $j$  raddoppia, quindi il ciclo viene eseguito  $\Theta(\log(n))$  volte. Alla fine del ciclo  $i$  avrà val.  $\log(n)$
- Il secondo ciclo viene eseguito da  $j=n$  a 1 e ad ogni iter.  $j$  diminuisce di  $\log(n)$ , quindi il ciclo viene eseguito  $\Theta(\frac{n}{\log(n)})$
- Vengono effettuate due chiamate ricorsive entrambe con input  $n/2$ .
- La ricorsione finisce quando  $n \leq 1$

L'eq di ricorrenza è

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

Usando il metodo principale abbiamo:

- $a = 2, b = 2$
- $f(n) = \Theta\left(\frac{n}{\log(n)}\right)$
- $n^{\log_2 2} = \Theta(n)$
- Poiché  $n$  cresce più velocemente di  $\frac{n}{\log(n)}$  ci ritroveremmo nel primo caso.
- Però poiché  $f(n)$  non è polinomialmente più piccolo di  $n^{\log_b(a)}$  non possiamo applicare il primo caso

## Esercizio 2

Sia  $A$  un array di  $n$  elementi con  $n$  pari, contenente uno stesso numero di interi pari ed interi dispari. Un riarrangiamento degli interi in  $A$  è valido se nelle posizioni ad indice pari compaiono interi pari e in quelle ad indice dispari interi dispari (l'indice 0 è considerato pari). Ad esempio, per  $A = [7, 3, 1, 8, 8, 2, 1, 4]$  esistono diversi riarrangiamenti validi come:  $[8, 7, 2, 3, 8, 1, 4, 1]$ , oppure  $[4, 1, 2, 1, 8, 3, 8, 7]$  o anche  $[2, 3, 8, 1, 8, 7, 4, 1]$ .

Progettare un algoritmo che, preso l'array  $A$ , produca un riarrangiamento valido.

L'algoritmo deve avere costo computazionale  $O(n)$  e deve utilizzare uno spazio di lavoro costante (in altri termini, non è possibile utilizzare liste concatenate o array di appoggio).

Dell'algoritmo proposto:

- a) si dia la descrizione a parole
- b) si scriva lo pseudocodice
- c) si giustifichi il costo computazionale.

### Soluzione

Utilizziamo due indici,  $p$  per scorrere gli indici pari e  $d$  per quelli dispari, che partono rispettivamente da 0 e da 1, quindi le successive posizioni pari e dispari per entrambi saranno  $p+2$  e  $d+2$ .

Ad ogni iterazione:

- Se il val. in  $p$  è pari: allora va bene e passiamo alla pos. pari successiva ( $p+2$ )
- Se il val. in  $d$  è dispari: allora va bene e passiamo alla pos dispari successiva( $d+2$ )
- Altrimenti dobbiamo scambiare i due elem. e poi passiamo alle posizioni pari e dispari successive

Continuo l'array finchè uno dei due indici non supera la fine dell'array

```

def Exam2_25_10_22(A):
    p, d = 0, 1
    while p < len(A) and d < len(A):
        if A[p] % 2 == 0: # se l'elem. in pos pari è pari
            p += 2 # vado alla pos. pari successiva
        elif A[d] % 2 == 1: # se l'elem. in pos dispari è dispari
            d += 2 # vado alla pos. dispari successiva
        else: # altrimenti devo scambiare i due elem.
            A[p], A[d] = A[d], A[p]
            p += 2 # e vado alle pos. pari e dispari successive
            d += 2
    return A

```

L'array ha sempre  $n/2$  elem. dispari e  $n/2$  pari.

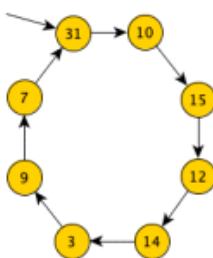
Ad ogni iter. viene aumentato di due almeno uno dei due indici, ed entrambi gli indici possono essere aumentati al più  $n/2$  volte, quindi il num. massimo di iter. del while è  $n$ . Quindi il costo è  $\Theta(n)$

### Esercizio 3

Si consideri una lista a puntatori circolare  $L$  data tramite un puntatore  $p$  ad un suo elemento. In  $L$  ogni nodo ha 2 campi: il campo key contenente un intero ed il campo next con il puntatore al nodo seguente.

Sappiamo che gli interi dei vari nodi sono tutti distinti e bisogna trovare il valore minimo tra questi.

Ad esempio per la lista circolare di seguito il valore cercato `e 3:



Progettare un algoritmo iterativo che, dato il puntatore  $p$  ad un nodo della lista circolare, restituisce il valore cercato in tempo  $\Theta(n)$  dove  $n$  `e il numero di nodi della lista.

Dell'algoritmo proposto:

- si dia la descrizione a parole
- si scriva lo pseudocodice
- si giustifichi il costo computazionale.

### Soluzione

Inizializzo una var.  $min$  per tenere conto del val. massimo che trovo. Scorro ogni puntatore e controllo se la sua chiave è minore della var.  $min$  allora aggiorno il val. di  $min$  con quello del nodo e poi passo al nodo successivo. Il ciclo finisce quando ritorno al nodo di partenza.

```

def Exam3_25_10_22(p):
    min = p.key
    # controllo la lista
    p_corr = p.next
    while p_corr != p:
        if p_corr.key < min:
            min = p_corr.key
        p_corr = p_corr.next
    return min

```

Ogni nodo viene visitato una volta sola nel ciclo, quindi il costo è  $\Theta(n)$

# Esame 17/1/23

mercoledì 12 febbraio 2025 18:01

## Esercizio 1

Per la soluzione di un certo problema disponiamo di un algoritmo iterativo con costo computazionale  $\Theta(n^3)$ . Ci viene proposto in alternativa un algoritmo ricorsivo il cui costo è catturato dalla seguente ricorrenza:

$$T = \begin{cases} T(n) = a * T\left(\frac{n}{2}\right) + \Theta(\sqrt{n}) & \text{se } n \geq 2 \\ T(n) = \Theta(1) & \text{se } n < 2 \end{cases}$$

dove  $a$  è una certa costante intera positiva con  $a \geq 2$

Determinare quale sia il valore massimo che la costante intera  $a$  può avere perché l'algoritmo ricorsivo risulti asintoticamente più efficiente dell'algoritmo iterativo di cui disponiamo. **Motivare bene la risposta**

### Soluzione

- Risolviamo la ricorrenza col val. minimo di  $a$  (2) usando il metodo principale

$$T(n) = 2 * T\left(\frac{n}{2}\right) + \Theta(\sqrt{n})$$

- $a = 2, b = 2$
- $f(n) = \Theta(\sqrt{n})$
- $n^{\log_2(2)} = \Theta(n)$

- Quindi il costo della ricorrenza è  $\Theta(n^{\log_2(2)})$  e per fare in modo di avere  $\Theta(n^3)$  come costo dell'algoritmo ricorsivo dovremmo avere  $a = 8$ .

Quindi il valore massimo che può avere  $a$  per fare in modo che il costo dell'algoritmo sia più efficiente è  $a = 7$

## Esercizio 2

Dati due arrays A e B, rispettivamente di  $n$  ed  $m$  interi distinti, con  $m < n$ , si vuole sapere se l'array A contenga l'array B come sottoarray. Ad esempio, se  $A = [5, 9, 1, 3, 4, 8, 2]$ , per  $B = [3, 4, 8]$  la risposta è SI mentre per  $B = [3, 8, 2]$  o  $B = [9, 6, 8]$  la risposta è NO

Progettare un algoritmo che, dati gli arrays A e B, restituisca 1 se la risposta al problema è SI, 0 altrimenti.

Il costo computazionale dell'algoritmo deve essere  $O(n)$ .

Dell'algoritmo proposto:

- si dia la descrizione a parole
- si scriva lo pseudocodice
- si giustifichi il costo computazionale.

### Soluzione

Utilizzo un indice  $b$  per scorrere negli elem. di B. Scorro gli elem. di A e appena incontro il primo elem. di B allora inizio a incrementare l'indice  $b$ . Per controllare se i prossimi elem. in A sono uguali a quelli in B e non sono stati saltati degli elem. utilizzo una var. di controllo `check` che inizializzo a 0 (False) e appena incontro un elem. di A che è anche in B la imposto a 1 (True).

- Nella iter. successiva, se l'elem. in A è diverso da quello in B e ho già incontrato qualche elem. che è in B allora B non è un sottoarray di A e posso uscire dall'algoritmo.
- Altrimenti aumento l'indice di B per controllare il suo elem. successivo nella prossima iter. di A.

Il ciclo finisce quando finisco gli elem. in A o quando finisco quelli in B.

Alla fine se abbiamo incontrato degli elem. che sono in B (quindi `check` è 1) possiamo ritornare 1 altrimenti 0.

```
def Exam2_17_1_23(A, B):  
    a, b, check = 0, 0, 0  
    check = False  
    while a < len(A) and b < len(B): # finché non finisce uno dei due array  
        if A[a] == B[b]: # se troviamo un elem. in A che è in B  
            b += 1 # aumentiamo l'indice di B  
            check = True # impostiamo il check a True  
        elif check == True:  
            # se troviamo un elem. in A che non è in B ma abbiamo già incontrato un elem. di B  
            return 0 # ritorniamo 0  
        a += 1 # aumentiamo l'indice di A  
    if b == len(B): # se abbiamo trovato tutti gli elem. di B in A  
        return 1 # allora B è sottoarray di A  
    return 0
```

Un altro metodo (usato dal prof) è quello di trovare prima la posizione in A di  $B[0]$  con un primo ciclo e poi eseguire un altro ciclo per controllare se gli elem. in A uguali a quelli in B sono consecutivi.

```

def es2(A, B):
    n, m = len(A), len(B)
    i = 0
    while i < n and A[i] != B[0]:
        i += 1
    if i == n: return 0
    j = 0
    while i < n and j < m and A[i] == B[j]:
        i += 1
        j += 1
    if j == m: return 1
    return 0

```

Nel mio codice ogni elem. di A viene visitato una sola volta, quindi ha costo O(n).

Nel codice del prof, il primo ciclo ha costo O(n), mentre il secondo ha costo O(min(n, m)) e poiché m < n ha costo O(n).

Quindi il costo complessivo è O(n)

### Esercizio 3

Sia dato un albero binario T, in cui ogni nodo p ha tre campi: il campo valore p.val, il campo col puntatore al figlio sinistro p.sx e il campo col puntatore al figlio destro p.dx, in mancanza di figlio il puntatore vale None.

Progettare un algoritmo ricorsivo che, dato il puntatore p alla radice dell'albero binario T, restituisca 1 se tutti i nodi dell'albero hanno lo stesso valore, 0 altrimenti. Il costo computazionale dell'algoritmo deve essere O(n), dove n `e il numero di nodi dell'albero.

Dell'algoritmo proposto:

- a) si dia la descrizione a parole
- b) si scriva lo pseudocodice
- c) si giustifichi il costo computazionale.

### Soluzione

Eseguo la visita postordine dell'albero e controllo se tutti i nodi nel sottoalbero sinistro e destro hanno lo stesso val di radice.

La funzione principale chiama la funzione ricorsiva che ha lo scopo di contare i nodi tot. dell'albero e il num. di nodi con chiave uguale a quella della radice.

La funzione ricorsiva prende in input il puntatore di un nodo e il val. della radice.

Eseguo la visita nei nodi figli che ritorneranno il num. di nodi visitati (n) e il num. di nodi con val uguale a quella della radice (c).

Aggiungo 1 al num. dei nodi visitati (Sx, Dx e quello corrente) e se il nodo ha val uguale a quello della radice allora aggiungo 1 al contatore.

Alla fine se il num. dei nodi (n) è uguale al num. di volte in cui si è incontrato il val. uguale a quello della radice (c) allora restituisco 1. 0 altrimenti

```

def Exam3_17_1_23(p):
    n, c = Controllo_val_uguali(p, p.key)
    if n == c:
        return 1
    return 0

def Controllo_val_uguali(p, r):
    if p == None:
        return 0, 0
    n_l, c_l = Controllo_val_uguali(p.left, r) # controllo il sottoalbero Sx
    n_r, c_r = Controllo_val_uguali(p.right, r) # controllo il sottoalbero Dx
    # i due sottoalberi ritornano il num. di nodi visitati e il num. di nodi con val = r
    n = n_l + n_r + 1 # aumento di 1 il num. di nodi visitati (nodi Sx, Dx e quello corrente)
    c = c_l + c_r # sommo gli elem. uguali alla radice
    if p.key == r: # se la chiave del nodo corrente è uguale a quella della radice
        c += 1 # aumento di 1 il contatore degli elem. uguali alla radice
    return n, c # ritorno i due contatori

```

Il costo è una normale visita dell'albero quindi  $\Theta(n)$

Il prof ha scritto un algoritmo più corto del mio e che non utilizza una funzione separata o un contatore per il num. di nodi uguali alla radice.

```

def es3(p):
    if p == None: # se non è un nodo
        return 1 # è comunque valido
    # se il sottoalbero sinistro o destro ha val. diversi
    if es3(p.sx) == 0 or es3(p.dx) == 0:
        return 0 # restituisco 0
    # se il nodo Sx/Dx non esiste oppure il suo valore è uguale a quello del nodo corrente
    if (p.sx == None or p.sx.val == p.val) and (p.dx == None or p.dx.val == val):
        return 1 # allora per ora l'albero ha tutti val. uguali
    return 0 # altrimenti abbiamo incontrato val. diversi tra padre e figli

```

## Esercizio 1

Si consideri il seguente algoritmo ricorsivo che, dato un array A di dimensione n, verifica se esistono due indici diversi i e j compresi nell'intervallo  $[0, n - 1]$  tali che  $A[i] = j$  e  $A[j] = i$ :

```
def IndiciValori(A, sx, dx):
    if (sx >= dx): return False
    else:
        trovato = False
        centro = (sx + dx)//2
        for i in range(sx, centro + 1):
            for j in range(centro + 1, dx + 1):
                if (A[i] == j) and (A[j] == i): trovato = True
        trovato1 = IndiciValori(A, sx, centro)
        trovato2 = IndiciValori(A, centro + 1, dx)
        return trovato or trovato1 or trovato2
```

- a) Si imposti la relazione di ricorrenza che definisce il tempo di esecuzione giustificando dettagliatamente l'equazione ottenuta.
- b) Si risolva la ricorrenza usando **due metodi** a scelta, dettagliando i passaggi del calcolo e giustificando ogni affermazione.

### Soluzione

- Caso base se l'indice sinistro raggiunge o supera quello destro (quindi siamo rimasti con un elem.) allora si esce dalla ricorsione
- Altrimenti si continua la ricorsione
  - Ci sono due for innidati in cui il primo itera dall'indice di sinistra a quello al centro e il secondo dal centro all'indice di destra.
  - Iterano entrambi  $n/2$  volte quindi hanno costo  $\Theta\left(\frac{n}{2} * \frac{n}{2}\right) = \Theta(n^2)$
- Infine si eseguono due chiamate ricorsive entrambe con input metà dell'array di partenza

L'equazione di ricorrenza è:

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

Usando il metodo principale abbiamo:

- $a = 2, b = 2$
- $f(n) = \Theta(n^2)$
- $n^{\log_2(2)} = \Theta(n)$
- Siamo quindi nel primo caso, quindi il costo sarà  $\Theta(n^2)$

Usando il metodo iterativo abbiamo

$$\begin{aligned} & \bullet \quad 2T\left(\frac{n}{2}\right) + \Theta(n^2) \\ & \bullet \quad 2\left(2T\left(\frac{n}{4}\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right)\right) + \Theta(n^2) = 2^2T\left(\frac{n}{2^2}\right) + 2\Theta\left(\frac{n^2}{2^2}\right) + \Theta(n^2) \\ & \bullet \quad 2\left(2\left(2T\left(\frac{n}{8}\right) + \Theta\left(\left(\frac{n}{4}\right)^2\right)\right) + \Theta\left(\left(\frac{n}{2}\right)^2\right)\right) + \Theta(n^2) = 2^3T\left(\frac{n}{2^3}\right) + 2^2\Theta\left(\frac{n^2}{2^4}\right) + 2\Theta\left(\frac{n^2}{2^2}\right) + \Theta(n^2) \end{aligned}$$

Generalizzo:

$$2^kT\left(\frac{n}{2^k}\right) + \sum_{i=0}^{k-1} 2^i * \Theta\left(\frac{n^2}{2^{i+2}}\right) = 2^kT\left(\frac{n}{2^k}\right) + n^2 \sum_{i=0}^{k-1} \Theta\left(2^i * \frac{1}{2^{2i}}\right) = 2^kT\left(\frac{n}{2^k}\right) + n^2 \Theta\left(\sum_{i=0}^{k-1} \frac{1}{2^i}\right)$$

L'eq. si ferma quando  $\frac{n}{2^k} = 1 \Rightarrow k = \log(n)$ , quindi:

$$2^{\log(n)}T(1) + n^2 * \Theta\left(\sum_{i=0}^{\log(n)} \frac{1}{2^i}\right) = n * \Theta(1) + n^2 * \Theta\left(2 - \frac{1}{2^{\log(n)}}\right) = \Theta(n) + n^2 * \Theta(1) = \Theta(n^2)$$

per  $n$  grande  $\frac{1}{n} \rightarrow 0$ , quindi:

$$\Theta\left(2 - \frac{1}{2^{\log(n)}}\right) = \Theta\left(2 - \frac{1}{n}\right) = \Theta(2) = \Theta(1)$$

## Esercizio 2

Scrivere un algoritmo ElementoPiuFrequente che, dato un array A di n interi, compresi tra 1 e  $10n$  restituisce il valore più presente all'interno dell'array, a parità di occorrenze va restituito il valore minimo.

Ad esempio, se  $A = [2, 6, 8, 5, 2, 3, 6, 8, 9, 5, 8, 1, 2]$ , allora la risposta è 2 in quanto 2 ed 8 sono gli unici valori che compaiono 3 volte all'interno dell'array, mentre gli altri valori compaiono al più 2 volte.

Il costo computazionale dell'algoritmo proposto deve essere  $\Theta(n)$ .

Dell'algoritmo proposto:

- a) si scriva lo pseudocodice opportunamente commentato
- b) si giustifichi il costo computazionale.

### Soluzione

Conviene usare un algoritmo di ordinamento e poi controllare l'array una volta sola per controllare l'elem. più frequente.

Poiché gli elem. nell'array sono lineari su n, il Counting Sort permette un costo  $\Theta(n)$ .

Dopo aver ordinato l'array, scorriamo gli elem. e per ogni catena di elem. uguali teniamo conto della frequenza della catena corrente e della frequenza massima (quindi la lunghezza degli elem. più frequenti) e dell'elemento con la frequenza più alta. Scorriamo l'array per cercare una catena di elem. maggiore. Se troviamo una catena con frequenza maggiore allora aggiorniamo la frequenza massima e l'elem. con frequenza più alta.

```

def Exam2_31_1_23(A):
    CountingSort(A) # ordiniamo l'array
    ris = A[0] # in partenza prendiamo il primo elem.
    freq, freq_max = 1, 1 # che avrà frequenza 1 per ora
    # freq è la frequenza dell'elem. corrente
    # freq_max è la frequenza dell'elemento più frequente
    for i in range(1, len(A)):
        if A[i-1] == A[i]: # se l'elem. prec. è uguale a quello corrente
            freq += 1 # aumentiamo la freq dell'elem. corrente
            if freq > freq_max: # se abbiamo superato la freq max precedente
                freq_max = freq # aumentiamo la freq max
            ris = A[i] # aggiorniamo l'elem. più frequente a quello corrente
        else: # altrimenti stiamo controllando un'altra catena di elem.
            freq = 1 # resettiamo la freq. dell'elem. corrente
    return ris

```

Il costo dipende dal costo del counting sort e del num. di iter. del ciclo while. Poiché gli elem. nell'array sono lineare in n abbiamo che il costo è dell'ordinamento è  $\Theta(n)$ , mentre il ciclo while viene eseguito su ogni elem. dell'array una sola volta quindi anche esso ha costo  $\Theta(n)$ .

Il costo tot. è  $\Theta(n)$

### Esercizio 3

Sia L una lista concatenata semplicemente puntata data tramite il puntatore p alla sua testa e contenenti chiavi intere positive. Ogni record è composto da due campi: il campo key che contiene il valore del nodo ed il campo next che contiene il puntatore al nodo successivo della lista se questo esiste, il valore None altrimenti.

Si progetti un algoritmo ricorsivo con costo computazionale  $O(n)$  che restituisca un puntatore al primo elemento della lista la cui chiave sia esattamente uguale alla somma delle chiavi di tutti gli elementi precedenti; se un tale elemento non esiste, verrà ritornato None. Ad esempio, per la lista  $p \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 6$  verrà restituito un puntatore al record contenente l'informazione 3; si noti che anche il record contenente l'informazione 6 soddisfa la richiesta di avere la chiave pari alla somma dei precedenti, ma il record contenente 3 lo precede.

Dell'algoritmo proposto:

- si scriva lo pseudocodice opportunamente commentato
- si giustifichi il costo computazionale trovando e risolvendo l'equazione di ricorrenza.

### Soluzione

Utilizziamo una var somma per tenere conto della somma dei nodi visitati. Ad ogni nodo controlliamo:

- Se somma = nodo.key allora abbiamo trovato il nodo valido e possiamo uscire dalla ricorsione e ritornare il puntatore di questo nodo
- Se somma  $\neq$  nodo.key allora aggiungiamo la sua chiave a somma e continuamo la ricerca sul nodo successivo inviando anche la somma aggiornata

Chiamiamo inizialmente la ricorsione mandando il nodo della testa e 0 come val iniziale della somma

```

def Exam3_31_1_23(p, somma):
    if p != None: # se il nodo esiste
        if p.key == somma: # se la chiave è uguale alla somma
            return p # ritorniamo il puntatore valido
        somma += p.key # altrimenti aggiungiamo la chiave corrente alla somma
        return Exam3_31_1_23(p.next, somma) # continuiamo la ricerca
    return None # se abbiamo finito la lista ritorniamo None

Exam3_31_1_23(p, 0)

```

Nel caso migliore abbiamo che il primo nodo nella lista è 0 quindi è valido (perché la somma iniziale è 0), quindi costo  $\Omega(1)$ .

Nel caso peggiore abbiamo che non ci sono nodi validi quindi scorriamo tutta la lista, quindi costo  $O(n)$

Ad ogni ricorsione scorriamo di 1 elem. nella lista e eseguiamo op. costanti.

Il caso base è quando finiamo i nodi.

Eq. di ricorrenza:

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

Usando il metodo iterativo abbiamo

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

Generalizzo

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

Si ferma quando  $n - k = 1 \Rightarrow k = n$ , quindi

$$\Theta(1) + n * \Theta(1) = \Theta(n)$$

## Esercizio 1

```
def Es1(n):
    if n < 10:
        return n
    s = Es1(n//2) * Es1(n//2)
    for j in range(n):
        k = n
        while k > 1:
            s = s + 1
            k = k//2
    return s + Es1(n//2)
```

- Si imposti la relazione di ricorrenza che ne definisce il tempo di esecuzione giustificando dettagliatamente l'equazione ottenuta.
- Qualora sia possibile, risolvere la ricorrenza utilizzando il teorema principale, dettagliando il caso del teorema ed i passaggi logici. Se il teorema principale non `e applicabile spiegarne il motivo

### Soluzione

- Il primo ciclo esterno itera da 0 a n, quindi ha costo  $\Theta(n)$
- Il ciclo while interno itera da n a  $\log(n)$  (perche ad ogni ricorsione viene diviso per 2), quindi ha costo  $\Theta(\log(n))$   
Il costo dei due cicli è  $\Theta(n \cdot \log(n))$
- Vengono eseguite 3 chiamate iterative fuori dai cicli, tutte e tre con chiamata di  $n/2$
- Il caso base si incontra quando  $n < 10$  ed ha costo costante

L'eq. di ricorrenza è:

$$T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n \cdot \log(n))$$

Usando il teorema principale abbiamo:

- $a = 3, b = 2$
- $f(n) = \Theta(n \cdot \log(n))$
- $n^{\log_2(3)} = n^{1.585} = n^{1+\epsilon}$
- Poiché  $n^{1+\epsilon}$  domina  $n \cdot \log(n)$  (siccome  $\log(n)$  cresce più lentamente rispetto a  $n^\epsilon$  con  $\epsilon > 0$ ), quindi siamo nel primo caso e il costo è  
 $T(n) = \Theta(n^{\log_2(3)})$

## Esercizio 2

Dato un array A di n interi non negativi distinti, si vuole determinare se esistono almeno tre numeri consecutivi di valore inferiore a 100. Ad esempio, se  $A = [101, 5, 9, 31, 33, 10, 100, 4, 8, 32, 500, 11, 99]$ , gli elementi 8, 9 e 10 così come gli elementi 31, 32 e 33 rispettano la proprietà mentre 99, 100 e 101 no.

Progettare un algoritmo che, dato A, in tempo  $\Theta(n)$  restituisce il valore dell'elemento centrale della terna se questa `e presente, -1 altrimenti. Se esistono piu' terne allora bisogna restituire l'elemento centrale di valore massimo (nell'esempio sopra, l'algoritmo dovrebbe restituire 32; se nell'array ci fossero 4 numeri consecutivi andrebbe restituito il terzo, ad esempio se l'array contenesse soltanto 5, 6, 7 e 8, andrebbe restituito il 7).

Dell'algoritmo proposto:

- si scriva lo pseudocodice opportunamente commentato;
- si giustifichi il costo computazionale.

### Soluzione

Un idea sarebbe di usare il counting sort per ordinare l'array e poi scorrere l'array per trovare una terna valida, solo che poiché gli elem. di A non sono lineari su n il suo costo peggiore sarebbe  $O(k)$  dove k è l'elem più grande di A.

Però possiamo usare un sistema simile al counting sort dove in un array aggiorniamo l'indice i solo per gli elem. di  $A[i]$  minori di 100 (limite lineare su n) e poi eseguiamo lo scorrimento nella lista al contrario per trovare la prima terna valida, quindi se gli elem in pos i-1, i e i+1 sono uguali a 1 (poiché sono elem. distinti ci saranno una sola volta in A).

Altrimenti se non abbiamo trovato una terna valida restituiamo -1

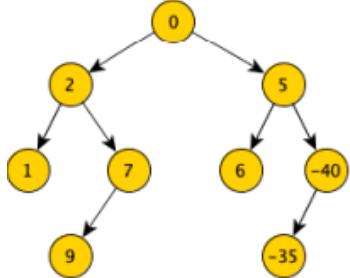
```
def Exam2_20_3_23(A):
    C[0]*100 # inizializziamo le 100 pos di C (che va da 0 a 99)
    for i in range(len(A)): # scorriamo i val di A
        if A[i] < 100: # inseriamo solo i val. minori di 100
            C[A[i]] += 1 # aumentiamo l'elem. di C in pos A[i] di 1
    for i in reversed(range(1, 99)): # scorriamo la lista al contrario dalla pos 98 a 1
        if C[i-1] == 1 and C[i] == 1 and C[i+1] == 1: # se la terna è valida
            return A[i] # ritorniamo l'elem. centrale
    return -1 # altrimenti se non abbiamo trovato una terna valida ritorniamo -1
```

Il costo tot. dipende dal costo massimo dei due cicli. Il primo ciclo scorre una sola volta l'array A, quindi ha costo  $\Theta(n)$ , mentre il secondo array nel caso peggiore (quindi la terna sono i primi tre elem. dell'array) ha costo  $O(1)$  (poiché esegue un num. costante di iterazioni). Quindi il costo tot è  $\Theta(n)$

### Esercizio 3

Un nodo di un albero a valori interi si dice nodo valido se la somma dei valori dei suoi antenati è uguale al valore del nodo. Ad esempio nell'albero binario in figura, risultano validi 3 nodi (quello con valore 0, quello con valore 9 e quello con valore -35).

Dato il puntatore r al nodo radice di un albero binario non vuoto, progettare un algoritmo ricorsivo che in tempo  $\Theta(n)$  calcoli il numero di nodi validi dell'albero. L'albero è memorizzato tramite puntatori e record a tre campi: il campo key contenente il valore ed i campi left e right con i puntatori ai figlio sinistro e al figlio destro, rispettivamente (questi puntatori valgono None in mancanza del figlio).



Dell'algoritmo proposto:

- si scriva lo pseudocodice opportunamente commentato;
- si giustifichi il costo computazionale.

NOTA BENE: nello pseudocodice dell'algoritmo ricorsivo non si deve far uso di variabili globali.

### Soluzione

Eseguo una visita preordine dell'albero e utilizzo una var. somma per indicare la somma dei nodi che si incontrano.

Ad ogni nodo eseguo il controllo ai nodi figli sommando il valore dei nodi validi che andranno ad incontrare con la differenza che:

- Se la somma dei suoi antenati è uguale alla chiave del nodo allora aggiungo 1 alla chiamata ricorsiva (per indicare che è il nodo corrente è valido)
- Altrimenti eseguo la normale chiamata ricorsiva sui figli.

Eseguo la chiamata della funzione inviando il puntatore della radice e la var somma inizializzata a 0

```

def Exam3_20_3_23(r, somma):
    if r == None: # se il nodo non esiste
        return 0 # restituisco 0
    if somma == r.key: # se il nodo è valido (somma degli antenati = chiave corrente)
        somma += r.key # incremento la somma col nodo corrente e aggiungo +1 alla chiamata ricorsiva sui figli
        return Exam3_20_3_23(r.left, somma) + Exam3_20_3_23(r.right, somma) + 1
    somma += r.key # altrimenti incremento la somma e chiamo ricorsivamente i figli
    return Exam3_20_3_23(r.left, somma) + Exam3_20_3_23(r.right, somma)
  
```

`Exam3_20_3_23(r, 0)`

Il costo dell'algoritmo è una normale visita dell'albero quindi ha costo  $\Theta(n)$ .

Svolgimento con metodo della sostituzione

L'eq. di ricorrenza è:

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

Rimuoviamo la notazione asintotica

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

Ipotizziamo che  $T(n) \leq O(n)$ , cioè  $T(n) \leq a^*n$  per qualche costante a

- Sostituiamo nel Caso Base  $T(1)$ :  
 $T(1) \leq a^*1 \Rightarrow d \leq a^*$
- Sostituimmo nel caso generico  $T(n)$ :

$$T(k) + T(n-k-1) + c \leq ak + a(n-k-1) + c \leq an \Rightarrow ak + an - ak - a + c \leq an \Rightarrow an + c \leq an + a \Rightarrow c \leq a$$

Poiché esistono due costanti  $c \leq a$  e  $d \leq a$ , abbiamo che  $T(n) = O(n)$

Ora ipotizziamo che  $T(n) \geq \Omega(n)$ , cioè  $T(n) \geq b^*n$  per qualche costante b

- Sostituiamo nel Caso Base  $T(1)$ :  
 $T(1) \geq b^*1 \Rightarrow d \geq b$
- Sostituimmo nel caso generico  $T(n)$ :

$$T(k) + T(n-k-1) + c \geq bk + b(n-k-1) + c \geq bn \Rightarrow bk + bn - bk - b + c \geq bn \Rightarrow bn + c \geq bn + b \Rightarrow c \geq b$$

Poiché esistono due costanti  $c \geq b$  e  $d \geq b$ , abbiamo che  $T(n) = \Omega(n)$

Quindi poiché  $T(n)$  è in  $O(n)$  e in  $\Omega(n)$  abbiamo che è anche in  $\Theta(n)$

## Esercizio 1

Siano

$$T(n) = 8T\left(\frac{n}{2}\right) + \Theta(n^2)$$

la funzione di costo di un algoritmo ricorsivo A, e

$$T(n) = bT\left(\frac{n}{4}\right) + \Theta(n^2)$$

la funzione di costo di un altro algoritmo ricorsivo A', dove b è una costante intera positiva, e per entrambe le ricorrenze vale  $T(1) = \Theta(1)$ . Qual è il minimo valore intero della costante b che rende A asintoticamente più veloce di A'? Dettagliare il ragionamento ed i passaggi del calcolo, giustificando ogni affermazione.

### Soluzione

Risolviamo intanto l'equazione di A col metodo principale:

- $a = 8, b = 2$
- $f(n) = \Theta(n^2)$
- $n^{\log_2(8)} = \Theta(n^3)$
- Siamo quindi nel terzo caso, quindi il costo sarà  $\Theta(n^3)$

Se avessimo  $b = 64$  avremmo che il costo di A' sarà anche esso  $\Theta(n^3)$ , poiché

- $a = 64, b = 4$
- $f(n) = \Theta(n^2)$
- $n^{\log_4(64)} = \Theta(n^3)$
- Siamo anche qui nel terzo caso, quindi il costo sarà  $\Theta(n^3)$

Per fare in modo che A sia asintoticamente più veloce di A' dovremmo avere come minimo  $b = 65$

## Esercizio 2

Dato un array A di n interi, si scriva un algoritmo iterativo MaxSequenzaElementiUguali che calcoli il numero di elementi della più lunga porzione di A costituita interamente da elementi consecutivi uguali tra loro.

Ad esempio, se  $A = [5, 7, 3, 3, 8, 9, 9, 9, 5, 3, 2, 2]$ , allora la risposta è 3 in quanto la porzione  $[9, 9, 9]$  è la più lunga formata da elementi consecutivi tutti uguali.

Dell'algoritmo proposto:

- si scriva lo pseudocodice opportunamente commentato;
- si calcoli formalmente il costo computazionale.

### Soluzione

È un esercizio simile a quello del 31/1/23

Devo tenere conto della frequenza massima di una sequenza precedente (freq\_max) e della frequenza degli elem. della sequenza corrente.

Ad ogni elem. controllo se il precedente è uguale a esso e aumento la frequenza corrente

Se l'elem. precedente è diverso da quello corrente (quindi stiamo controllando un'altra sequenza) allora resetto la freq. corrente ma prima controllo se la frequenza degli elem. precedenti supera la frequenza massima, e in quel caso aggiorno la freq. massima a quella della freq. precedente.

```
def Exam2_12_9_23(A):
    freq_max, freq = 1, 1 # partiamo già con la freq. di A[0]
    for i in range(1, len(A)):
        if A[i-1] == A[i]: # se l'elem. prec. è uguale al corrente
            freq += 1 # aumentiamo la sua freq. corrente
        else: # se stiamo controllando un'altra sequenza
            if freq > freq_max: # e la freq. precedente supera quella massima
                freq_max = freq # allora aggiorniamo la freq. massima con quella precedente
            freq = 1 # e resettiamo la freq. corrente per controllare la nuova sequenza
        if freq > freq_max: # se la freq. finale supera quella massima
            return freq # ritorniamo la freq. finale
    return freq_max # altrimenti quella massima
```

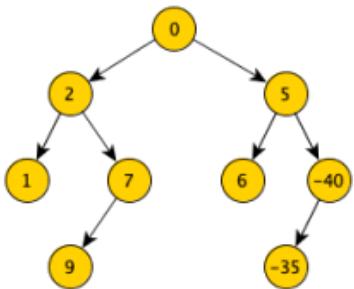
Il ciclo controlla gli elem. di A una sola volta e il resto delle op. sono costanti, quindi ha costo  $\Theta(n)$

## Esercizio 3

Dato un albero binario non vuoto a valori interi T ed un suo nodo v, il costo del cammino radice-v è definito come la somma dei valori dei nodi nel percorso che va dalla radice al nodo v (estremi inclusi).

Vogliamo calcolare il costo del massimo cammino radicefoglia di T.

Ad esempio nell'albero binario in figura, la risposta è 18, infatti nell'albero sono presenti quattro diversi cammini radice-foglia di costo 3, 18, 11 e -70, rispettivamente.



Dato il puntatore  $r$  al nodo radice di un albero binario non vuoto a valori interi  $T$ , progettare un algoritmo ricorsivo che, in tempo  $\Theta(n)$ , risolva il problema.

L'albero è memorizzato tramite puntatori e record a tre campi: il campo key contenente il valore ed i campi left e right con i puntatori ai figli sinistro e al figlio destro, rispettivamente (questi puntatori valgono None in mancanza del figlio).

Dell'algoritmo proposto:

- si scriva lo pseudocodice opportunamente commentato;
- si giustifichi formalmente il costo computazionale.

NOTA BENE: nello pseudocodice dell'algoritmo ricorsivo non si deve far uso di variabili globali.

### Soluzione

Eseguo una visita postordine e per ogni nodo devo contare il massimo della somma dei suoi sottoalberi.

Se i nodi figli esistono prendo la somma dei loro sottoalberi e ritorno il valore massimo tra i due aggiungendo la chiave del nodo corrente

```

def Exam3_12_9_23(r):
    somma_l, somma_r = 0, 0 # inizializzo la somma dei figli Sx e Dx
    if r.left != None: # se il figlio Sx esiste
        somma_l = Exam3_12_9_23(r.left) # prendo la somma del suo sottoalbero
    if r.right != None: # se il figlio Dx esiste
        somma_r = Exam3_12_9_23(r.right) # prendo la somma del suo sottoalbero
    # ritorno la somma tra la chiave corrente e il massimo dei suoi sottoalberi
    return r.key + max(somma_l, somma_r)

```

Il costo è quello di una visita di un albero con  $n$  nodi.

L'eq. di ricorrenza è:

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

Dove  $k$  è il num. di nodi nel sottoalbero sinistro e  $n-k-1$  il num. di nodi nel sottoalbero destro.

- Nel caso migliore abbiamo un albero binario completo in cui ogni nodo ha esattamente due figli. Quindi entrambi i sottoalberi hanno  $n/2$  nodi.

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

Possiamo usare il metodo principale per risolvere l'equazione:

- $a = 2, b = 2$
- $f(n) = \Theta(1)$
- $n^{\log_2(2)} = \Theta(n)$
- Ci troviamo nel terzo caso, quindi il costo è  $T(n) = \Theta(n)$

- Nel caso peggiore abbiamo che è un albero totalmente sbilanciato con i nodi solo a destra o solo a sinistra.

- Nel caso in cui i nodi sono totalmente a sinistra abbiamo  $n-k-1 = 0 \Rightarrow k = n-1$ :

$$T(n) = T(n - 1) + T(0) + \Theta(1) = T(n - 1) + \Theta(1)$$

- Nel caso in cui i nodi sono totalmente a destra abbiamo  $k = 0$ :

$$T(n) = T(0) + T(n - 0 - 1) + \Theta(1) = T(n - 1) + \Theta(1)$$

In entrambi i casi abbiamo, quindi possiamo usare il metodo iterativo e trovare che il suo costo è  $\Theta(n)$

Poiché in entrambi abbiamo che il costo è  $\Theta(n)$ , possiamo ipotizzare che il costo medio sia  $\Theta(n)$

Possiamo verificare questa ipotesi usando il metodo sostitutivo.

- Rimuoviamo la notazione asintotica
 
$$T = \begin{cases} T(n) = T(k) + T(n - k - 1) + c \\ T(1) = d \end{cases}$$
- Tentiamo la sol.  $T(n) \leq O(n)$ , quindi  $T(n) \leq an$  per qualche costante  $a$  da determinare.
  - Sostituendo nel caso base  $T(1)$  abbiamo:
 
$$T(1) = d \leq a * 1 \Rightarrow d \leq a$$
  - Sostituendo nel caso generico  $T(n)$  abbiamo:
 
$$T(n) \leq ak + a(n - k - 1) + c \leq an \Rightarrow ak + an - ak - a + c \leq an \Rightarrow -a + c \leq 0 \Rightarrow c \leq a$$
- Poiché entrambe le diseguaglianze sono vere, deduciamo che  $T(n) = O(n)$
- Tentiamo la sol.  $T(n) \geq \Omega(n)$ , quindi  $T(n) \geq bn$  per qualche costante  $b$  da determinare.
  - Sostituendo nel caso base  $T(1)$  abbiamo:
 
$$T(1) = d \geq b * 1 \Rightarrow d \geq b$$
  - Sostituendo nel caso generico  $T(n)$  abbiamo:
 
$$T(n) \geq bk + b(n - k - 1) + c \geq bn \Rightarrow bk + bn - bk - b + c \geq bn \Rightarrow -b + c \geq 0 \Rightarrow c \geq b$$
- Poiché entrambe le diseguaglianze sono vere, deduciamo che  $T(n) = \Omega(n)$

- Siccome  $T(n) = \Omega(n)$  e  $T(n) = O(8n)$ , possiamo dire che  $T(n) = \Theta(n)$

# Esame 10/23

mercoledì 12 febbraio 2025 18:01

## Esercizio 1

Siano:

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

una funzione di costo di un algoritmo ricorsivo A, e

$$T(n) = aT(n/2) + \Theta(1)$$

una funzione di costo di un altro algoritmo ricorsivo A', dove a è una costante intera positiva maggiore di 1, e per entrambe le ricorrenze vale  $T(1) = \Theta(1)$ .

Qual è il minimo valore intero della costante a che rende A asintoticamente più veloce di A'?

Dettagliare il ragionamento ed i passaggi del calcolo, giustificando ogni affermazione.

### Soluzione

Prima risolviamo l'eq. di A col metodo iterativo:

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

Generalizzo

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

Si ferma quando  $n - k = 1 \Rightarrow k = n$ , quindi

$$\Theta(1) + \Theta(n^2) = \Theta(n^2)$$

Calcoliamo ora l'eq di A':

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

Usando il metodo principale abbiamo:

- $a = a, b = 2$
- $f(n) = \Theta(1)$
- Poiché  $n^{\log_2(a)}$  domina sempre  $f(n)$  ci troviamo quindi nel caso terzo caso che ha costo  $\Theta(n^{\log_2(a)})$

Per fare in modo che A' abbia un costo uguale a quello di A, bisogna avere  $n^2 < n^{\log_2(a)} \Rightarrow 2 < \log_2(a) \Rightarrow 4 < a$ , quindi il val. min. di a che rende A più efficiente di A' è 5

## Esercizio 2

Un array A di interi positivi si dice valido se i numeri nelle posizioni potenza di due non superano i numeri nelle posizioni potenza di tre ed inoltre i numeri in queste posizioni sono tutti pari.

Ad esempio:

- l'array A = [1, 50, 20, 70, 6, 11, 10, 21, 40, 80, 1, 1, 13, 1, 22, 64, 30, 1] è valido in quanto  $A[2^0] = 50, A[2^1] = 20, A[2^2] = 6, A[2^3] = 40, A[2^4] = 30, A[3^0] = 50, A[3^1] = 70, A[3^2] = 80$ . Inoltre i numeri nelle posizioni potenza di due non superano i numeri nelle posizioni potenza di tre.
- l'array B = [1, 50, 20, 70, 6, 11, 10, 21, 40, 85, 1, 1, 13, 1, 22, 64, 30, 1] non è valido in quanto  $B[3^2] = 85$  che è dispari
- l'array C = [1, 50, 20, 70, 6, 11, 10, 21, 40, 80, 1, 1, 13, 1, 22, 64, 90, 1] non è valido in quanto  $C[2^4] = 90 > C[3^0] = 50$

si scriva un algoritmo iterativo che, dato un array di n elementi, in tempo O(log n) restituisce 1 se A è valido, 0 altrimenti.

Dell'algoritmo proposto:

- si scriva lo pseudocodice opportunamente commentato;
- si calcoli formalmente il costo computazionale.

### Soluzione

Siccome dobbiamo controllare le posizioni  $2^k$  e  $3^k$  possiamo eseguire due scorrimenti dell'array solo su quelle posizioni usando:

- una var. exp per tenere conto della potenza di due o di tre corrente
  - due var. per indicare il val. massimo incontrato nelle pos. potenze di due e il val. minimo incontrato nelle potenze di tre
- Scorriamo entrambe le posizioni delle potenze di due e di tre in due cicli separati in cui controlliamo che le potenze di due/tre non superino la grandezza dell'array.

Controlliamo sempre in entrambi se il val. nella pos. potenza di due/tre non sia dispari (altrimenti l'array non è valido)

Scorrendo l'array nelle posizioni con potenza di due prendiamo il massimo elem. incontrato.

In modo analogo, scorrendo l'array nelle pos. con potenza di tre prendiamo il min. elem. incontrato.

Alla fine se il max elem. in potenza di due supera il min. elem. in potenza di tre allora l'array non è valido, altrimenti lo è.

Nell'array il num. di pos. con potenza di due sarà  $\lceil \log_2(n) \rceil$  quindi termina dopo massimo  $\log_2(n)$  iter. quindi ha costo  $O(\log(n))$

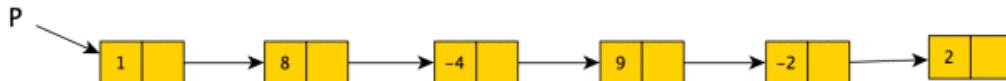
Allo stesso modo, il num. di pos con potenza di tre sarà  $\lceil \log_3(n) \rceil$  quindi termina dopo massimo  $\log_3(n)$  iter. quindi ha costo  $O(\log(n))$ .

Quindi lo scorrimento avrà costo  $O(\log(n))$  poiché le altre op. sono costanti.

## Esercizio 3

Abbiamo una lista a puntatori contenente nodi con campo chiave contenente interi e vogliamo sapere il valore massimo ed il valore minimo delle chiavi dei nodi della lista.

Ad esempio per la lista a puntatori in figura la risposta è la coppia di interi 9 e -4.



Dato il puntatore  $p$  al nodo testa della lista di  $n \geq 1$  nodi, progettare un algoritmo **ricorsivo** che, in tempo  $\Theta(n)$ , risolva il problema. Ciascun nodo della lista ha due campi: il campo **key** contenente il valore chiave ed il campo **next** al nodo seguente (**next** è pari a **None** per l'ultimo nodo della lista).

Dell'algoritmo proposto:

- si scriva lo pseudocodice opportunamente commentato;
- si giustifichi formalmente il costo computazionale.

NOTA BENE: nello pseudocodice dell'algoritmo ricorsivo non si deve far uso di variabili globali.

### Soluzione

Per salvare i dati dei val. min. e max nella lista potremmo usare due var. globali o due var. locali da inviare in ogni chiamata ricorsiva.

Usiamo due var min e max per salvare rispettivamente la chiave min e max trovata nella lista.

Scorriamo gli elem. nella lista e controlliamo se il nodo corrente ha un val più piccolo del min o più grande del max e nel caso fosse così li aggiorniamo al val. della chiave corrente. Chiamiamo poi recursivamente il prossimo nodo inviando come var. il val min e max.

Un altro modo che non utilizza nessuna variabile aggiuntiva è usare le funzioni **max** e **min** tra gli elem. di ritorno dalla chiamata ricorsiva e il valore della chiave corrente.

Quindi la chiamata ricorsiva tornerà il val min e max degli elem. successivi al nodo corrente.

Se abbiamo finito la lista ritorniamo **None** per il min e il max. Se invece siamo all'ultimo elem. ritorniamo per min e per max la sua chiave.

```
def Exam3_10_23(p):  
    if p == None: # se non è un nodo  
        return None, None # ritorniamo nulla  
    if p.next == None: # se è l'ultimo elem  
        return p.key, p.key # ritorniamo solo la sua chiave  
    min_p, max_p = Exam3_10_23(p.next) # altrimenti prendiamo il min e max del resto della lista  
    return min(min_p, p.key), max(max_p, p.key) # e ritorniamo il min e max tra i val. ritornati e la chiave corrente
```

La chiamata ricorsiva viene chiamata una sola volta e scorre un singolo elem. della lista alla volta.

La sua equazione di ricorrenza sarà:

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

Col metodo iterativo il suo costo sarà  $\Theta(n)$  (nell'esame va scritto per intero)

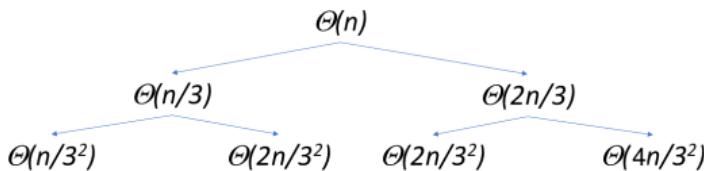
## Esercizio 1

Si risolva con il metodo dell'albero la seguente equazione di ricorrenza:

$$T = \begin{cases} T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + \Theta(n) \\ T(1) = \Theta(1) \end{cases}$$

Si dettaglino i ragionamenti ed i passaggi del calcolo, giustificando ogni affermazione.

### Soluzione



Facilmente vediamo che il contributo di ogni è sempre lo stesso, pari a  $\Theta(n)$ . Però calcolare l'albero è un po' più complicato poiché non è un albero binario completo e il ramo sinistro finirà prima di quello destro.

Infatti il ramo sinistro finirà quando  $\frac{n}{3^k} = 1 \Rightarrow k = \log_3(n)$ , mentre quello destro quando  $\frac{2^kn}{3^k} = 1 \Rightarrow k = \log_{3/2}(n)$ .

Ogni ramo avrà un num. di liv. compreso tra i due, quindi dobbiamo maggiorare e minorare il val. di  $T(n)$ , ricordando che il contributo di ciascun livello è  $\Theta(n)$ :

$$\log_3(n) * \Theta(n) \leq T(n) \leq \log_{\frac{3}{2}}(n) * \Theta(n)$$

Con la formula per la trasformazione della base dei log  $\log_a(n) = \log_b(n) * \log_a(b)$  otteniamo che sia il primo che il terzo membro della prec. catena di disegualanze sono pari a  $\Theta(n * \log(n))$ , per cui in questo caso siamo riusciti a trovare due limitazioni.

## Esercizio 2

Si scriva lo pseudocodice, opportunamente commentato, di una funzione **iterativa** che, preso in input un array A di interi non nulli (positivi e negativi), ne sposti gli elementi in modo che gli interi negativi precedano gli interi positivi e restituisca poi l'array così modificato.

Ad esempio per  $A = [3, -5, -7, 1, -8]$  due possibili risposte corrette (ma ve ne sono altre) sono  $A = [-8, -7, -5, 1, 3]$  o  $A = [-5, -8, -7, 3, 1]$ .

La funzione deve avere costo computazionale  $O(n)$ , dove n è il numero di elementi presenti nell'array. Il costo in termini di spazio oltre l'array A deve essere  $\Theta(1)$  (in pratica non può far uso di array di appoggio).

Si motivi in dettaglio il costo computazionale dell'algoritmo proposto. Si risponda, inoltre, alle seguenti domande:

- qual è il costo computazionale se l'array in input è ordinato in modo non decrescente?
- qual è il costo computazionale se l'array in input è ordinato in modo non crescente?
- qual è il costo computazionale se l'array in input è costituito da elementi tutti uguali?

### Soluzione

Utilizzo due indici s e d che partono rispettivamente dal primo e dall'ultimo elem. nell'array dove s scorre in avanti nell'array e d all'indietro. In un ciclo while scorro la lista in base a queste regole:

Se l'elem. puntato da s è positivo e quello puntato da d è negativo, li scambio tra di loro e incremento entrambi gli indici

Altrimenti

- Se l'elem puntato da s è negativo, incremento s per passare all'elem. successivo
- Se l'elem. puntato da d è positivo, decremento d per passare all'elem. successivo

```

def Exam2_17_1_24(A):
    s, d = 0, len(A)-1
    while s < d: # finchè l'indice Sx non supera quello Dx
        if A[s] >= 0 and A[d] < 0:
            # se l'elem. a Sx è >= 0 e quello a Dx è < 0
            A[s], A[d] = A[d], A[s] # li scambio tra di loro
            s += 1 # incremento/decremento entrambi gli indici
            d -= 1 # per passare all'elem. successivo
        else:
            if A[s] < 0: # se l'elem. a Sx è negativo
                s += 1 # passo all'elem. Sx successivo
            if A[d] >= 0: # se l'elem. a Dx è positivo
                d -= 1 # passo all'elem. Dx successivo
    return A
  
```

Ad ogni iter. almeno uno dei due indici passa all'elem. successivo.

Quindi nel caso migliore in cui gli elem. negativi sono già a destra e quelli positivi già a sinistra (tipo in ordine decrescente) o se tutti gli elem. negativi sono a sinistra e i positivi a destra (tipo in ordine crescente) gli indici aumenterebbero insieme e si incontrerebbero a metà array, quindi vengono eseguite  $n/2$  iter., costo  $\Omega(n)$ .

Nel caso peggiore in cui gli elem. sono tutti uguali, allora solo uno dei due indici scorrerà per tutta la lista eseguendo n iter., quindi costo

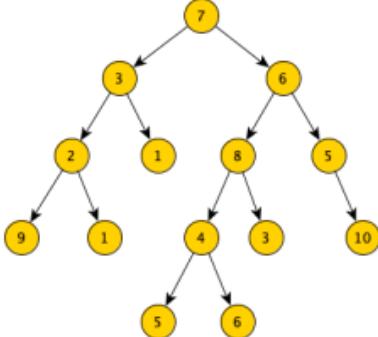
$O(n)$ . Quindi il costo medio è  $\Theta(n)$ .

### Esercizio 3

Si consideri un albero binario radicato  $T$ , i cui nodi abbiano un campo  $val$  contenente un intero e i campi  $left$  e  $right$  con i puntatori ai figli. Un nodo  $v$  dell'albero si dice valido se si verificano le seguenti tre condizioni:

- $v$  ha entrambi i figli;
- il campo  $val$  di uno dei due figli è minore di quello padre;
- il campo  $val$  dell'altro figlio è maggiore di quello del padre.

Ad esempio nell'albero in figura ci sono esattamente 2 nodi validi (quelli con valore 6 e 2).



Si progetti una funzione **ricorsiva** che, dato il puntatore  $r$  alla radice di  $T$  restituisca il numero di nodi validi di  $T$  in tempo  $O(n)$  dove  $n$  è il numero di nodi presenti nell'albero.

Della funzione proposta:

- si dia la descrizione a parole
- si scriva lo pseudocodice
- si giustifichi formalmente il costo computazionale.

NOTA BENE: nello pseudocodice della funzione ricorsiva non si deve far uso di variabili globali.

### Soluzione

Ad ogni nodo esistente, controllo che se nel caso ha figlio destro e sinistro, allora prendo il val. massimo e minimo tra le chiavi dei figli.

Poi se il nodo corrente ha la chiave minore del massimo dei figli e maggiore del minimo dei figli allora è un nodo valido.

Infine ritorno la somma delle chiamate sui figli (per controllare se ci sono altri nodi validi) e aggiungo 1 nel caso in cui il nodo corrente sia valido.

```

def Exam3_17_1_24(r):
    if r == None:
        return 0
    val = 0
    if r.left != None and r.right != None: # se ha entrambi i figli
        f_min = min(r.left.key, r.right.key) # prendi il val. minimo dei figli
        f_max = max(r.left.key, r.right.key) # e quello max
        if r.key > f_min and r.key < f_max:
            # se il nodo è minore del figlio min. e maggiore del figlio max
            val = 1 # allora è valido
    # ritorno i nodi validi nei figli e la validità del nodo corrente
    return Exam3_17_1_24(r.left) + Exam3_17_1_24(r.right) + val
  
```

Il costo dell'algoritmo è quello di una normale visita dell'albero binario che ha eq. di ricorrenza:

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

Dove  $k$  è il num. di nodi nel sottoalbero sinistro e  $n-k-1$  il num. di nodi nel sottoalbero destro.

- Nel caso migliore abbiamo un albero binario completo in cui ogni nodo ha esattamente due figli. Quindi entrambi i sottoalberi hanno  $n/2$  nodi.

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

Possiamo usare il metodo principale per risolvere l'equazione:

- $a = 2, b = 2$
- $f(n) = \Theta(1)$
- $n^{\log_2(2)} = \Theta(n)$
- Ci troviamo nel terzo caso, quindi il costo è  $T(n) = \Theta(n)$

- Nel caso peggiore abbiamo che è un albero totalmente sbilanciato con i nodi solo a destra o solo a sinistra.

- Nel caso in cui i nodi sono totalmente a sinistra abbiamo  $n-k-1 = 0 \Rightarrow k = n-1$ :

$$T(n) = T(n - 1) + T(0) + \Theta(1) = T(n - 1) + \Theta(1)$$

- Nel caso in cui i nodi sono totalmente a destra abbiamo  $k = 0$ :

$$T(n) = T(0) + T(n - 0 - 1) + \Theta(1) = T(n - 1) + \Theta(1)$$

In entrambi i casi abbiamo , quindi possiamo usare il metodo iterativo e trovare che il suo costo è  $\Theta(n)$

Poiché in entrambi abbiamo che il costo è  $\Theta(n)$ , possiamo ipotizzare che il costo medio sia  $\Theta(n)$

Possiamo verificare questa ipotesi usando il metodo sostitutivo.

- Rimuoviamo la notazione asintotica

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

- Tentiamo la sol.  $T(n) \leq O(n)$ , quindi  $T(n) \leq an$  per qualche costante  $a$  da determinare.
  - Sostituendo nel caso base  $T(1)$  abbiamo:  
 $T(1) = d \leq a * 1 \Rightarrow d \leq a$
  - Sostituendo nel caso generico  $T(n)$  abbiamo:  
 $T(n) \leq ak + a(n - k - 1) + c \leq an \Rightarrow ak + an - ak - a + c \leq an \Rightarrow -a + c \leq 0 \Rightarrow c \leq a$
  - Poiché entrambe le diseguaglianze sono vere, deduciamo che  $T(n) = O(n)$
- Tentiamo la sol.  $T(n) \geq \Omega(n)$ , quindi  $T(n) \geq bn$  per qualche costante  $b$  da determinare.
  - Sostituendo nel caso base  $T(1)$  abbiamo:  
 $T(1) = d \geq b * 1 \Rightarrow d \geq b$
  - Sostituendo nel caso generico  $T(n)$  abbiamo:  
 $T(n) \geq bk + b(n - k - 1) + c \geq bn \Rightarrow bk + bn - bk - b + c \geq bn \Rightarrow -b + c \geq 0 \Rightarrow c \geq b$
  - Poiché entrambe le diseguaglianze sono vere, deduciamo che  $T(n) = \Omega(n)$
- Siccome  $T(n) = \Omega(n)$  e  $T(n) = O(n)$ , possiamo dire che  $T(n) = \Theta(n)$

# Esame 14/2/24

mercoledì 12 febbraio 2025 18:01

## Esercizio 1

Dato un albero binario T con n nodi, memorizzato tramite record e puntatori, e dato tramite il puntatore p alla sua radice, si consideri il seguente algoritmo di visita in preordine:

```
def Visita_preordine (p):
    if (p != None):
        # qui istruzioni di costo costante di accesso al nodo
        Visita_preordine (p.left)
        Visita_preordine (p.right)
    return
```

Si scriva l'equazione di ricorrenza che ne descrive il costo computazionale e la si risolva in termini di notazione asintotica stretta.

Si dettaglino i ragionamenti ed i passaggi del calcolo, giustificando ogni affermazione.

### Soluzione

Questa è una normale visita dell'albero e l'ho appena scritta nell'esercizio 3 dell'esame precedente.

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

Dove k è il num. di nodi nel sottoalbero sinistro e n-k-1 il num. di nodi nel sottoalbero destro.

- Nel caso migliore abbiamo un albero binario completo in cui ogni nodo ha esattamente due figli. Quindi entrambi i sottoalberi hanno  $n/2$  nodi.

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

Possiamo usare il metodo principale per risolvere l'equazione:

- $a = 2, b = 2$
- $f(n) = \Theta(1)$
- $n^{\log_2(2)} = \Theta(n)$
- Ci troviamo nel terzo caso, quindi il costo è  $T(n) = \Theta(n)$

- Nel caso peggiore abbiamo che è un albero totalmente sbilanciato con i nodi solo a destra o solo a sinistra.

- Nel caso in cui i nodi sono totalmente a sinistra abbiamo  $n-k-1 = 0 \Rightarrow k = n-1$ :

$$T(n) = T(n - 1) + T(0) + \Theta(1) = T(n - 1) + \Theta(1)$$

- Nel caso in cui i nodi sono totalmente a destra abbiamo  $k = 0$ :

$$T(n) = T(0) + T(n - 0 - 1) + \Theta(1) = T(n - 1) + \Theta(1)$$

In entrambi i casi abbiamo, quindi possiamo usare il metodo iterativo e trovare che il suo costo è  $\Theta(n)$

Poiché in entrambi abbiamo che il costo è  $\Theta(n)$ , possiamo ipotizzare che il costo medio sia  $\Theta(n)$

Possiamo verificare questa ipotesi usando il metodo sostitutivo.

- Rimuoviamo la notazione asintotica

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

- Tentiamo la sol.  $T(n) \leq O(n)$ , quindi  $T(n) \leq an$  per qualche costante a da determinare.

- Sostituendo nel caso base  $T(1)$  abbiamo:

$$T(1) = d \leq a * 1 \Rightarrow d \leq a$$

- Sostituendo nel caso generico  $T(n)$  abbiamo:

$$T(n) \leq ak + a(n - k - 1) + c \leq an \Rightarrow ak + an - ak - a + c \leq an \Rightarrow -a + c \leq 0 \Rightarrow c \leq a$$

- Poiché entrambe le diseguaglianze sono vere, deduciamo che  $T(n) = O(n)$

- Tentiamo la sol.  $T(n) \geq \Omega(n)$ , quindi  $T(n) \geq bn$  per qualche costante b da determinare.

- Sostituendo nel caso base  $T(1)$  abbiamo:

$$T(1) = d \geq b * 1 \Rightarrow d \geq b$$

- Sostituendo nel caso generico  $T(n)$  abbiamo:

$$T(n) \geq bk + b(n - k - 1) + c \geq bn \Rightarrow bk + bn - bk - b + c \geq bn \Rightarrow -b + c \geq 0 \Rightarrow c \geq b$$

- Poiché entrambe le diseguaglianze sono vere, deduciamo che  $T(n) = \Omega(n)$

- Siccome  $T(n) = \Omega(n)$  e  $T(n) = O(n)$ , possiamo dire che  $T(n) = \Theta(n)$

## Esercizio 2

Sia dato un array A di n elementi memorizzati in ordine qualunque, nel quale si vogliono determinare i k elementi più grandi (dove k è un valore compreso tra 1 ed n).

Si considerino i seguenti due approcci:

- si ordini A e si prendano i primi k elementi;
- si costruisca uno heap massimo in tempo lineare, e per k volte si esegua l'estrazione del massimo.

Per ciascun approccio, si calcoli il costo computazionale stretto (cioè in termini di  $\Theta$ ) come funzione di k e di n; poi si discuta per quali valori di k ciascun approccio risulti asintoticamente migliore dell'altro in termini di costo computazionale; infine, si scriva lo pseudocodice della funzione di costruzione dello heap massimo (Build\_Heap).

### Soluzione

- 1) L'ordinamento con Merge Sort da costo  $\Theta(n \log(n))$ , mentre l'estrazione dei primi  $k$  elem. ha costo  $\Theta(k)$ . Quindi il costo tot è  $\Theta(n \log(n))$
- 2) La creazione di un heap ha costo  $\Theta(n)$ , mentre l'estrazione dei  $k$  elem. richiede ogni volta di ricostruire con Heapify l'heap quindi ha costo  $O(k \log(n))$ . quindi il costo tot. è  $\Theta(n+k \log(n))$  però poiché  $k = \Theta(n)$  abbiamo che il costo è  $O(n \log(n))$

Quindi i due approcci hanno lo stesso costo quando  $k = \Theta(n)$ . Se abbiamo un qualunque  $k$  più piccolo di  $\Theta(n)$  allora il secondo approccio è più veloce del primo.

```
def Build_heap(A):
    for i in reversed(range(len(A)//2)):
        Heapify(A, i, len(A))
    return A
```

### Esercizio 3

Si consideri una lista puntata  $L$  contenente chiavi intere positive, memorizzata tramite record e puntatori, ed accessibile tramite il puntatore  $p$  alla sua testa.

Definiamo  $k$  come un valore strettamente minore della somma di tutte le chiavi della lista puntata. Sia  $m$  un valore che rappresenta il minimo numero dei primi elementi della lista puntata la somma delle cui chiavi superi  $k$ .

Ad esempio, se la lista è la seguente:

$p \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$

e  $k$  vale 5, allora il valore restituito  $m$  sarà pari a 3, perché  $1 + 2 + 3 \geq k$  mentre  $1 + 2 < k$ .

Si progetti una funzione **ricorsiva** che, dato in input il puntatore  $p$  ed il parametro  $k$ , restituisca il valore  $m$ .

Della funzione proposta:

- si dia la descrizione a parole
- si scriva lo pseudocodice
- si giustifichi formalmente il costo computazionale.

NOTA BENE: nello pseudocodice della funzione ricorsiva non si deve far uso di variabili globali né di memoria aggiuntiva di dimensione non costante.

### Soluzione

Per poter calcolare che la somma delle chiavi precedenti superi  $k$  dobbiamo utilizzare una var somma da inviare ai nodi successivi durante la chiamata ricorsiva.

La var somma tiene conto delle somme dei nodi incontrati, e appena arriviamo un nodo la cui somma dei nodi precedenti più il suo supera o è uguale a  $k$  allora possiamo fermare la ricorsione e ritornare 1, altrimenti continuamo il controllo al nodo successivo e aggiungere 1 per indicare il nodo corrente controllato.

```
def Exam3_14_2_24(p, k, som = 0):
    som += p.key # altrimenti aggiungo la chiave corrente alla somma
    if som >= k: # se la somma ha superato k
        return 1 # inizio a ritornare il num. di nodi incontrati
    return Exam3_14_2_24(p.next, k, som) + 1 # ritorno il num. di nodi incontrati + 1 (nodo corrente)
```

Il prof ha fatto un codice leggermente più compatto che non utilizza la var. aggiuntiva per la somma.

Ad ogni chiamata ricorsiva rimuoveva il val. della chiave del nodo corrente da  $k$ . Se poi il nodo era maggiore o uguale al nuovo  $k$  allora ritornava 1, altrimenti eseguiva la chiamata ricorsiva rimuovendo la chiave da  $k$  e aggiungendo 1 per indicare il nodo corrente controllato.

```
def es3(p,k):
    if p.val >= k:
        return 1
    return 1 + es3(p.next, k-p.val)
```

Il costo di entrambi gli algoritmi è lo stesso. Nel caso peggiore dobbiamo scorrere tutta la lista e l'eq. di ricorrenza è:

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

Usando il metodo iterativo abbiamo che il costo è  $\Theta(n)$

## Esercizio 1

```
def Potenza(n, k):
    if k == 0: return 1
    if k == 1: return n
    if k % 2 == 0:
        pot = Potenza (n, k DIV 2)
        return pot * pot
    else:
        pot = Potenza (n, (k-1) DIV 2)
        return n * pot * pot
```

- a) Si imposti la relazione di ricorrenza che ne definisce il tempo di esecuzione giustificando dettagliatamente l'equazione ottenuta.
- b) La si risolva utilizzando **due dei metodi studiati**, dettagliando sia i passaggi matematici che quelli logici.

### Soluzione

Poiché tutte le op. sono costanti, il costo totale dipende solo dalla chiamata ricorsiva che viene effettuata solo in due casi.

- Se  $k$  è pari allora chiamiamo la funzione ricorsiva mandando  $k/2$
- Se  $k$  è dispari allora chiamiamo la funz. ricorsiva mandando  $(k-1)/2$

Viene quindi effettuata una sola chiamata ad ogni passo ricorsivo e viene dimezzato il suo input, quindi l'eq di ricorrenza sarà:

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

L'eq. ottenuta è uguale a quella della ricerca binaria.

Risolvendo col metodo principale abbiamo:

- $a = 1, b = 2$
- $f(n) = \Theta(1)$
- $n^{\log_2(1)} = n^0 = \Theta(1)$
- Ci troviamo quindi nel secondo caso, quindi abbiamo che il costo sarà  $\Theta(f(n)*\log(n)) = \Theta(\log(n))$

Risolvendo col metodo iterativo abbiamo:

- $T\left(\frac{n}{2}\right) + \Theta(1)$
- $T\left(\frac{n}{4}\right) + \Theta(1) + \Theta(1) = T\left(\frac{n}{2^2}\right) + 2\Theta(1)$
- $T\left(\frac{n}{8}\right) + \Theta(1) + \Theta(1) + \Theta(1) = T\left(\frac{n}{2^3}\right) + 3\Theta(1)$

Generalizzando abbiamo:

$$T\left(\frac{n}{2^k}\right) + k\Theta(1)$$

Si ferma quando  $\frac{n}{2^k} = 1 \Rightarrow k = \log(n)$ , quindi:

$$T(1) + \log(n) * \Theta(1) = \Theta(\log(n))$$

## Esercizio 2

Si definisce punto di sella di una matrice quell'elemento che gode della proprietà di essere simultaneamente minimo di riga e massimo di colonna. Si progetti un algoritmo che, data una matrice quadrata  $M$  di  $n \times n$  interi distinti, restituisca una coppia di indici  $(i, j)$  corrispondenti alla posizione del punto di sella in  $M$ , e `None` se la matrice non ha punti di sella. L'algoritmo deve avere costo computazionale  $\Theta(n^2)$ .

Dell'algoritmo proposto:

- a) si scriva lo pseudocodice opportunamente commentato
- b) si giustifichi il costo computazionale.

### Soluzione

Per trovare un punto di sella posso utilizzare due array, `min_r` in cui salvo gli elem. minimi di ogni riga e `max_c` in cui salvo gli elem. massimi di ogni colonna.

Il punto di sella sarà l'elem. in comune tra le due liste che posso trovare usando due cicli annidati.

Se l'elem. in comune si trova in  $(i, j)$  nella matrice, allora nelle due liste sarà in `min_r[i]` e `max_c[j]`

Se arriviamo alla fine del controllo delle due liste allora vuol dire che non ho trovato un punto di sella e restituisco `None`

```

# esame 12/6/24:
def Exam2_12_6_24(M):
    n = len(M[0])
    min_r = max_c = [0]*n
    for i in range(n):
        # ricerca val. minimi nella riga i
        # i = row, j = col
        min_m = M[i][0]
        for j in range(1, n):
            if M[i][j] < min_m:
                min_m = M[i][j]
        min_r.append(min_m)
        # ricerca val. massimi nella colonna i
        # j = col, i = row
        max_m = M[0][i]
        for j in range(1, n):
            if M[j][i] > max_m:
                max_m = M[j][i]
        max_c.append(max_m)
    # ricerchiamo l'elem. in comune nelle due liste
    for i in range(n):
        for j in range(n):
            if min_r[i] == max_c[j]:
                return i, j
    return None

```

Il codice del prof è 100 volte meglio (tacca sua) perché non utilizza array aggiuntivi.

Per ogni riga  $r$  trova la pos  $c$  in cui compare l'elem. minimo e poi per la colonna  $c$  trova la pos del val massimo della colonna  $c$  e se corrisponde a  $r$  allora è un punto di sella, altrimenti passa alla riga successiva.

```

def Exam2_12_6_24_prof(M):
    n = len(M)
    for r in range(n):
        # prendiamo la pos della colonna del val. min. nella riga corrente
        c_min = minC(r, M)
        # prendiamo la pos della riga del val max nella colonna trovata
        r_max = maxR(c_min, M)
        # se la pos del min della riga e del max. della colonna è uguale alla riga corrente
        if r == r_max: # allora è un punto di sella
            return r, c_min # ritorniamo le posizioni
    return None # se non abbiamo trovato nulla allora ritorniamo None

def minC(r, M):
    # cerchiamo il val. minimo nella riga r
    n, mc = len(M), 0
    for c in range(1, n):
        # se il val. corrente è più piccolo di quello min. prec.
        if M[r][c] < M[r][mc]:
            mc = c # aggiorniamo la pos del val. min
    return mc # ritorniamo la pos. della colonna del val. min. nella riga r

def maxR(c, M):
    # cerchiamo il val. minimo nella riga r
    n, mr = len(M), 0
    for r in range(1, n):
        # se il val. corrente è più grande di quello del max. prec.
        if M[r][c] > M[mr][c]:
            mr = r # aggiorniamo la pos. del val. max
    return mr # ritorniamo la pos. della riga del val. max della colonna c

```

Il costo della ricerca del min della riga e del max della colonna è per entrambi  $\Theta(n)$  che vengono ripetuti per  $n$  volte, quindi il costo è:  $n*\Theta(n) + n*\Theta(n) = \Theta(n^2)$

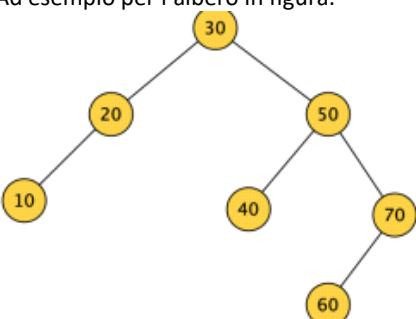
Perché il costo di due cicli annidati.

Nel mio codice poi viene eseguiti altri due cicli annidati per la ricerca degli elem. in comune quindi anche lì il costo è  $\Theta(n^2)$  ma non cambia il costo totale.

### Esercizio 3

Siano dati un intero positivo  $k$  ed un albero binario di ricerca  $T$  di altezza  $h$ ; si vuole individuare la  $k$ -esima chiave di  $T$  se queste fossero messe in ordine.

Ad esempio per l'albero in figura:



- con  $k = 1$  la risposta è 10
- per  $k = 5$  la risposta è 50
- per  $k = 9$  la risposta è None perché l'albero ha meno di  $k$  nodi.

L'albero è memorizzato tramite puntatori e record di quattro campi: il campo key contenente il valore, i campi left e right con i puntatori

al figlio sinistro e al figlio destro, rispettivamente (questi puntatori valgono None in mancanza del figlio), ed il campo num con l'indicazione del numero dei nodi nel sottoalbero in esso radicato.

Si progetti un algoritmo RICORSIVO che risolva il problema in un tempo computazionale  $O(h)$ .

Dell'algoritmo proposto:

- a) si scriva lo pseudocodice opportunamente commentato;
- b) si giustifichi il costo computazionale.

NOTA BENE: nello pseudocodice dell'algoritmo ricorsivo è preferibile **non** far uso di variabili globali.

**Soluzione**

# Esame 10/7/24

mercoledì 12 febbraio 2025 18:01

## Esercizio 1

```
def Es1(n):
    if n < 5:
        return n
    s = k = n
    while k > 1:
        s += k
        k = k//3
    return s + Es1(n - 1)
```

- Si imposti la relazione di ricorrenza che ne definisce il tempo di esecuzione giustificando dettagliatamente l'equazione ottenuta.
- Si risolva la ricorrenza utilizzando uno dei metodi studiati, dettagliando sia i passaggi matematici che quelli logici.

Soluzione

## Esercizio 2

Sia data una collezione C di n interi compresi tra 0 a 50 tra cui sono certamente presenti dei duplicati. Gli elementi della collezione sono memorizzati in un array A. Si vuole determinare la distanza massima tra le posizioni di due elementi duplicati in A, cioè il massimo al variare di  $x \in C$  di  $\max(j - i)$ , t.c.  $A[i] = x$  e  $A[j] = x$ .

Ad esempio per  $A = [3, 3, 4, 6, 6, 3, 5, 5, 5, 6, 6, 6, 9, 9, 1]$  i soli elementi che in A si ripetono sono 3, 5, 6 e 9.

- La distanza massima tra i duplicati del 3 è 5 ( $j = 5$  e  $i = 0$ )
- la distanza massima tra i duplicati del 5 è 2 ( $j = 8$ ,  $i = 6$ )
- la distanza massima tra i duplicati del 6 è 7 ( $j = 3$  e  $i = 10$ )
- la distanza massima tra i duplicati del 9 è 1 ( $j = 12$  e  $i = 11$ ).

quindi la risposta per l'array A è 7.

Progettare un algoritmo che, dato A, in tempo  $\Theta(n)$  restituisca la distanza massima tra le posizioni con elementi duplicati.

Dell'algoritmo proposto:

- si scriva lo pseudocodice opportunamente commentato
- si giustifichi il costo computazionale.

Soluzione

Utilizziamo un array C di 51 elem. inizializzati tutti a -1, dove andremmo ad inserire nella pos  $C[x]$  la prima posizione dove  $x$  viene trovato in A. Usiamo anche una var.  $mdis$ , inizializzata a 0, che indica la distanza massima tra le pos. di due elem. duplicati

Scorrendo l'array controlliamo che:

- se l'elem. in pos  $C[x] = -1$  allora ancora non abbiamo impostato la prima pos. di  $x$ , quindi aggiorniamo  $C[x]$  con la pos.  $i$  di  $x$
- Se l'elem. in pos  $C[x] \neq -1$  allora abbiamo già trovato la prima pos. di  $x$ , calcoliamo quindi la distanza tra l'elem. corrente (pos  $i$ ) e la distanza della sua prima pos. ( $C[x]$ ) calcolando semplicemente  $i - C[x]$  e poi se questa distanza è maggiore della distanza massima precedente ( $mdis$ ) allora aggiorniamo  $mdis$  con la nuova distanza maggiore

```
def Exam2_10_7_24(A):
    mdis, n = 0, len(A)
    C = [-1]*51 # inizializziamo C
    for i in range(n):
        if C[A[i]] != -1: # se abbiamo già trovato la prima pos. di A[i]
            # calcoliamo la distanza dell'elem dalla pos corrente a quella iniziale
            dis = i - C[A[i]]
            mdis = max(mdis, dis) # troviamo il massimo tra la distanza massima precedente e la distanza corrente
        else: # se non abbiamo ancora inserito la prima pos. di A[i]
            C[A[i]] = i # inseriamo in C nella pos. A[i] la sua prima pos.
    return mdis
```

C ha un num. costante di elem. quindi la sua creazione ha costo  $\Theta(1)$

Poiché  $\max$  è eseguito tra 2 elem. il suo costo è costante,  $\Theta(1)$

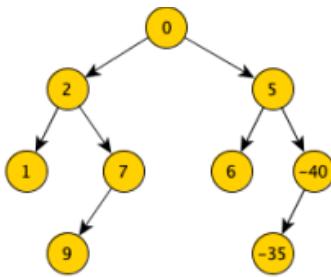
Il costo totale è principalmente dato dal num. di iterazioni del ciclo for, che esegue  $n$  iter. in cui vengono eseguite op. co stanti.

Quindi il costo totale è  $n * \Theta(1) = \Theta(n)$

## Esercizio 3

Dato il puntatore r al nodo radice di un albero binario non vuoto, progettare un algoritmo ricorsivo che in tempo  $\Theta(n)$  calcoli il numero di nodi che hanno esattamente 2 figli e chiavi pari.

Ad esempio, per l'albero in figura, l'algoritmo deve restituire 2, per la presenza dei nodi con chiavi 2 e 0.



L'albero è memorizzato tramite puntatori e record di tre campi: il campo key contenente il valore ed i campi left e right con i puntatori ai figli sinistro e al figlio destro, rispettivamente (questi puntatori valgono None in mancanza del figlio).

Dell'algoritmo proposto:

- si scriva lo pseudocodice opportunamente commentato;
- si giustifichi il costo computazionale.

NOTA BENE: nello pseudocodice dell'algoritmo ricorsivo **non** si deve far uso di variabili globali.

### Soluzione

Eseguo una visita preordine e ad ogni nodo controllo se i due figli sono esistenti e se la sua chiave è pari. Se è così allora aggiungo +1 alla chiamata dei nodi successivi

Altrimenti eseguo il controllo normale sui nodi figli ritornando la somma dei valori di ritorno delle loro chiamate.

```

def Exam3_10_7_24(r):
    if r == None:
        return 0
    # se il nodo è pari e ha due figli
    if r.key%2 == 0 and r.left != None and r.right != None:
        # allora aggiungo 1 alla chiamata ricorsiva sui figli
        return Exam3_10_7_24(r.left) + Exam3_10_7_24(r.right) + 1
    # altrimenti non aggiungo nulla alla chiamata ricorsiva sui figli
    return Exam3_10_7_24(r.left) + Exam3_10_7_24(r.right)
  
```

Il costo è quello di una visita dell'albero, quindi la sua eq. di ricorrenza è:

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

dove k è il num. di nodi nel sottoalbero sinistro e n-k-1 il num. di nodi nel sottoalbero destro.

Nel caso migliore abbiamo che l'albero è completo, quindi ogni nodo ha esattamente due figli, quindi:

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

Che possiamo risolvere col metodo principale:

- a = 2, b = 2
- f(n) = Θ(1)
- $n^{\log_2(2)} = \Theta(n)$
- Ci troviamo quindi nel terzo caso e il costo sarà  $\Theta(n)$

Nel caso peggiore abbiamo che l'albero è completamente sbilanciato quindi i nodi saranno tutti a destra o tutti a sinistra

- Se sono tutti a destra abbiamo che k = 0, quindi:  
 $T(n) = T(0) + T(n-0-1) + \Theta(1) = T(n-1) + \Theta(1)$
- Se sono tutti a sinistra abbiamo che  $n-k-1 = 0 \Rightarrow k = n-1$ , quindi:  
 $T(n) = T(n-1) + T(0) + \Theta(1) = T(n-1) + \Theta(1)$

Quindi in entrambi i casi abbiamo  $T(n) = T(n-1) + \Theta(1)$ , e usando il metodo iterativo troviamo che il suo costo è  $\Theta(n)$

Siccome in entrambi i casi abbiamo che il costo è  $\Theta(n)$ , possiamo ipotizzare che il costo medio sia  $\Theta(n)$ .

Per verificare l'ipotesi possiamo usare il metodo della sostituzione.

- Rimuoviamo la notazione asintotica:
- $T = \begin{cases} T(n) = T(k) + T(n-k-1) + c \\ \quad \quad \quad T(1) = d \end{cases}$
- Tentiamo la soluzione  $T(n) \leq a * n = O(n)$ , dove a è una costante indeterminata
  - Sostituiamo nel caso base  $T(1)$ :  
 $T(1) = d \leq a * 1 \Rightarrow d \leq a$
  - Sostituiamo nel caso generale  $T(n)$ :  
 $T(n) \leq ak + a(n-k-1) + c \leq an \Rightarrow ak + an - ak - a + c \leq an \Rightarrow -a + c \leq 0 \Rightarrow c \leq a$
  - Poiché le diseguaglianze  $d \leq a$  e  $c \leq a$  sono vere, possiamo dire che  $T(n) = O(n)$
- Tentiamo la soluzione  $T(n) \geq b * n = \Omega(n)$ , dove b è una costante indeterminata
  - Sostituiamo nel caso base  $T(1)$ :  
 $T(1) = d \geq b * 1 \Rightarrow d \geq b$
  - Sostituiamo nel caso generale  $T(n)$ :  
 $T(n) \geq bk + b(n-k-1) + c \geq bn \Rightarrow bk + bn - bk - b + c \geq bn \Rightarrow -b + c \geq 0 \Rightarrow c \geq b$
  - Poiché le diseguaglianze  $d \leq a$  e  $c \leq a$  sono vere, possiamo dire che  $T(n) = \Omega(n)$
- Siccome abbiamo visto che  $T(n) = O(n)$  e  $T(n) = \Omega(n)$ , possiamo dire che  $T(n) = \Theta(n)$

## Esercizio 1

Si consideri la seguente funzione, in cui i valori a e b in input sono due interi maggiori di 1:

```
def Es1(n, a, b):
    if n < 1:
        return 1
    c, t, s = n//b, a, 0
    while t > 0:
        for j in range(a):
            s += Es1(c, a, b)
        t -= 1
    for i in range(n):
        for j in range(n):
            s += i + j
    return s
```

- a) Si imposti la relazione di ricorrenza che ne definisce il tempo di esecuzione in termini di n, a e b giustificando dettagliatamente l'equazione ottenuta.
- b) Si risolva la ricorrenza utilizzando uno dei metodi studiati, nei tre casi possibili  $a > b$ ,  $a = b$  e  $a < b$ .

### Soluzione

- Il caso base si incontra quando  $n < 1$  ed ha costo costante  $\Theta(1)$
  - Il ciclo while viene eseguito da  $t = a$  fino a 0, quindi viene eseguito a volte.
  - Il ciclo for interno al while viene eseguito anche esso da 0 ad a, quindi a volte.
  - Infine abbiamo due cicli annidati che vengono eseguiti n volte, quindi insieme hanno costo  $\Theta(n^2)$
  - La chiamata ricorsiva viene effettuata all'interno del ciclo for e tra gli argomenti in input, c è l'unico che cambia durante tutta la ricorsione, poiché a e b non vengono modificati nella funzione, quindi l'eq.  $T(n, a, b)$  si deve solo calcolare  $T(n)$ .  
Il costo dei due cicli annidati è quindi  $a * a * \Theta(1) = \Theta(a^2)$ .
  - la chiamata ricorsiva viene anche essa effettuata  $a^2$  volte con  $n$  che viene diviso per  $b$  ad ogni iterazione. quindi l'eq. di ricorrenza è:
- $$T = \begin{cases} T(n) = a^2 T\left(\frac{n}{b}\right) + \Theta(n^2) \\ T(1) = \Theta(1) \end{cases}$$

con  $a > 1, b > 1$ :

Possiamo risolvere la ricorrenza col metodo principale.

Dobbiamo confrontare  $n^{\log_b(a^2)} = n^{2\log_b(a)}$  e  $f(n) = \Theta(n^2)$  tra di loro. Consideriamo i tre casi:

- Se  $a > b$  allora  $\log_b(a) > 1$  quindi  $n^{2\log_b(a)}$  è più grande e ci troviamo nel 3° caso, quindi il costo è  $\Theta(n^{2\log_b(a)})$
- Se  $a = b$  allora  $\log_b(a) = 1$  quindi  $n^{2\log_b(a)} = n^2$  è uguale a  $f(n)$  e ci troviamo nel 2° caso, quindi il costo è  $\Theta(n^2 * \log(n))$
- Se  $a < b$  allora  $\log_b(a) < 1$  quindi  $f(n)$  è più grande e ci troviamo nel 1° caso, quindi il costo è  $\Theta(n^2)$

## Esercizio 2

Diciamo che l'elemento di indice i in un array è un massimo parziale se è strettamente maggiore di tutti gli elementi che lo precedono nell'array, cioè di tutti gli elementi di indice tra 0 ed  $i - 1$ .

Dato un array A di n interi, si vuole determinare il numero di elementi di A che siano massimi parziali. Ad esempio per  $A = [3, 4, 2, 5, 1]$  la risposta è 3 (i massimi parziali sono infatti 3, 4 e 5).

Progettare un algoritmo che, dato A, in tempo  $\Theta(n)$  risolva il problema.

Dell'algoritmo proposto:

- a) si scriva lo pseudocodice opportunamente commentato
- b) si giustifichi il costo computazionale.

### Soluzione

Possiamo usare una var pmax per indicare l'elem. massimo trovato prima dell'elem. in indice i e una var. cont per tenere conto dei massimi parziali nella lista. Se un elem. è strettamente maggiore di pmax allora impostiamo pmax a questo elem. e aumentiamo il contatore.

```

def Exam2_18_9_24(A):
    pmax, cont, n = 0, 0, len(A)
    for i in range(n): # scorriamo l'array
        if A[i] > pmax: # se l'elem corrente è maggiore del max precedente
            pmax = A[i] # aggiorniamo il max
            cont += 1 # aumentiamo il contatore
    return cont

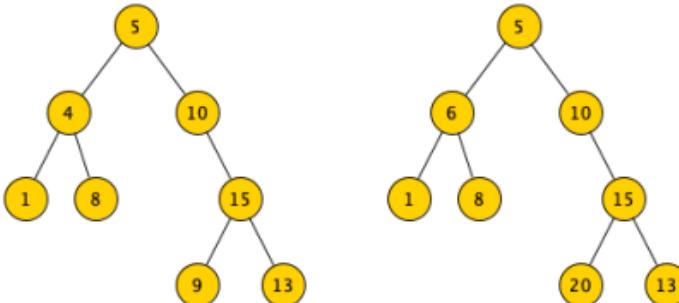
```

Il costo dipende dal ciclo che esegue n iter. di op. costanti. Quindi il costo è  $n * \Theta(1) = \Theta(n)$

### Esercizio 3

Sia dato un albero binario T non vuoto contenente interi, e questo sia memorizzato tramite puntatori e record di tre campi: il campo key contenente il valore ed i campi left e right contenenti i puntatori ai figli sinistro e al figlio destro, rispettivamente (questi puntatori valgono None in mancanza del figlio).

Dato il puntatore r al nodo radice di T, progettare un algoritmo ricorsivo che in tempo  $\Theta(n)$  restituisca T rue se in T esiste un cammino radice-foglia dove gli interi dei nodi che si incontrano lungo il cammino formano una sequenza strettamente crescente, F alse altrimenti. Ad esempio, per l'albero a sinistra in figura, l'algoritmo deve rispondere F alse perché nessuna delle sequenze di valori che si incontrano nei cammini radice-foglia dell'albero risulta crescente. Per l'albero a destra in figura, invece, l'algoritmo deve rispondere T rue, infatti esistono ben due cammini radicefoglia i cui valori formano sequenze strettamente crescenti (le sequenze sono 5, 6, 8 e 5, 10, 15 20).



Dell'algoritmo proposto:

- si scriva lo pseudocodice opportunamente commentato;
- Si ricavi il costo computazionale dell'algoritmo producendo la relazione di ricorrenza che cattura il tempo di calcolo dell'algoritmo e risolvendola.

NOTA BENE: nello pseudocodice dell'algoritmo ricorsivo **non** si deve far uso di variabili globali.

### Soluzione

Eseguo una visita preordine.

Ad ogni nodo incontro:

- Se è una foglia allora ritorno direttamente True
- Se ha un figlio sinistro/destro e il nodo è maggiore del figlio e la chiamata ricorsiva nel figlio non ha restituito False allora ritorniamo True
- Altrimenti restituisco False perché non abbiamo trovato cammini strettamente crescenti

```

def Exam3_18_9_24(r):
    # Se sono ad una foglia allora ritorno True
    if not r.left and not r.right:
        return True
    # se la chiave corrente è maggiore del figlio Sx/Ds e la chiamata ricorsiva sul figlio non ha dato problema
    if r.left and r.key > r.left.key and Exam3_18_9_24(r.left): # allora posso restituire True
        return True
    if r.right and r.key > r.right.key and Exam3_18_9_24(r.right):
        return True
    # altrimenti non abbiamo trovato cammini strettamente crescenti e restituiamo False
    return False

```

Il costo dell'algoritmo è quello di una visita di un albero con n nodi. L'eq. di ricorrenza è:

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

Dove k è il num. di nodi del sottoalbero sinistro e n-k-1 il num. di nodi del sottoalbero destro.

Il costo della visita l'abbiamo visto negli scorsi esami

# Esame 24/10/24

mercoledì 12 febbraio 2025 18:01

## Esercizio 1

Siano date le tre seguenti equazioni di ricorrenza:

- $T = \begin{cases} T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n) \\ T(1) = \Theta(1) \end{cases}$
- $T = \begin{cases} T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^2) \\ T(1) = \Theta(1) \end{cases}$
- $T = \begin{cases} T(n) = 4T\left(\frac{n}{2}\right) + \Theta(n^3) \\ T(1) = \Theta(1) \end{cases}$

Si risolvano con il teorema principale, dettagliando il ragionamento usato ed evidenziando le differenze.

### Soluzione

Tutte e tre le eq. hanno  $a = 4$  e  $b = 2$ , quindi  $n^{\log_2(4)} = n^2$

Nella prima eq. abbiamo che  $f(n) = \Theta(n) = O(n^{\log_2(4)-\varepsilon})$  per  $\varepsilon = 1$ , perciò siamo nel 1° caso e il costo è  $\Theta() = \Theta(n^2)$

Nella seconda abbiamo  $f(n) = \Theta(n^2) = \Theta(n^{\log_2(4)})$ , perciò siamo nel 2° caso e il costo è  $\Theta(f(n)*\log(n)) = \Theta(n^2*\log(n))$

Nella terza abbiamo  $f(n) = \Theta(n^3) = \Omega(n^{\log_2(4)+\varepsilon})$  per  $\varepsilon = 1$  e  $4 * \left(\frac{n}{2}\right)^3 \leq c * n^3 \rightarrow 4 * \frac{n^3}{8} \leq c * n^3 \rightarrow \frac{n^3}{2} \leq c * n^3$  con  $c = 1/2$ , perciò siamo nel 3° caso e il costo è  $\Theta(f(n)) = \Theta(n^3)$

## Esercizio 2

Si scriva lo pseudocodice, opportunamente commentato, di una funzione iterativa che, preso in input un array A di interi, trovi la lunghezza massima delle sequenze crescenti presenti nell'array.

Ad esempio:

- per  $A = [3, 1, 5, 2, 6, 8, 7, 1]$  la risposta è 3, che è la lunghezza della sequenza 2, 6, 8;
- per  $A = [10, 7, 6, 1]$  la risposta è 1 perché non ci sono in A sequenze crescenti di lunghezza maggiore ad 1.

La funzione deve avere costo computazionale  $O(n)$ , dove  $n$  è il numero di elementi presenti nell'array. Il costo in termini di spazio oltre l'array A deve essere  $\Theta(1)$  (in pratica non si può far uso di array di appoggio).

Il costo computazionale dell'algoritmo proposto va dettagliato valutando il contributo di ogni istruzione.

### Soluzione

Se la lista è vuota ritorniamo direttamente 0.

Utilizzo una var. l per tenere traccia della lunghezza massima e lc per tenere traccia della lunghezza della seq. corrente.

Ad ogni elem. controllo:

- Se il precedente è minore dell'elem. corrente allora aumento la lungh. della sequenza corrente (lc)
- Altrimenti abbiamo terminato la sequenza corrente, quindi resettiamo lc ma controlliamo prima che se supera la lunghezza massima precedente allora essa diventerà la nuova lunghezza massima

Alla fine ritorniamo il val. massimo tra l e lc, nel caso in cui la sequenza più lunga finisce con l'ultimo elem.

```
def Exam2_10_24(A):  
    if len(A) == 0: return 0  
    l, lc, n = 1, 1, len(A) # partiamo dal primo elem.  
    for i in range(1, n):  
        # se il prec. è minore del corrente  
        if A[i-1] < A[i]:  
            # allora la sequenza continua  
            lc += 1  
        else: # altrimenti abbiamo finito la sequenza prec.  
            l = max(l, lc) # nel caso aggiorniamo la lungh. max con quella prec.  
            lc = 1 # e ricominciamo un'altra sequenza  
    return max(l, lc) # prendiamo il massimo tra la seq. corrente e quella max
```

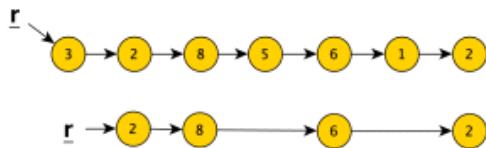
Il ciclo for esegue sempre per n volte diverse op. costanti (il max è eseguito su 2 val. quindi anche esso è costante), quindi il costo tot. è  $\Theta(n)$

## Esercizio 3

Si consideri una lista concatenata, i cui nodi abbiano un campo key contenente un intero ed un campo next al nodo successivo (next vale None se si tratta dell'ultimo nodo).

Si progetti una funzione **ricorsiva** che, dato il puntatore r alla testa della lista restituisca il puntatore alla testa della lista modificata in modo che vi compaiano solo i nodi contenenti gli interi pari.

Ad esempio, data la lista concatenata r della figura, la funzione deve restituire il puntatore alla nuova lista concatenata contenente i soli nodi con valore pari.



La funzione deve avere costo computazionale  $O(n)$  dove  $n$  è il numero di nodi presenti nella lista originaria e l'algoritmo non può generare nuovi nodi né utilizzare array di appoggio.

Della funzione proposta:

- si dia la descrizione a parole
- si scriva lo pseudocodice
- si giustifichi formalmente il costo computazionale dando la ricorrenza che lo caratterizza e poi la sua soluzione.

NOTA BENE: nello pseudocodice della funzione ricorsiva **non** si deve far uso di variabili globali.

### Soluzione

Se il nodo corrente è dispari allora eseguo la chiamata ricorsiva sul nodo successivo che mi restituirà il puntatore del prossimo nodo pari, così posso fare in modo di cambiare il puntatore al nodo corrente con quello del nodo pari successivo.

Se il nodo è pari allora continuo la ricorsione sul nodo successivo

```

def Exam3_21_1_25(r):
    if not r.left and not r.right: # se è una foglia
        return r.key # ritorno la chiave
    sx, dx = float('inf'), float('inf') # inizializzo a infinito il cammino dei figli
    if r.left: # se il figlio esiste
        sx = Exam3_21_1_25(r.left) # controllo la lunghezza del cammino nodo-foglia
    if r.right:
        dx = Exam3_21_1_25(r.right)
    return min(sx, dx) + r.key # ritorno il min cammino dei sottoalberi + la chiave corrente

```

## Esercizio 1

Supponiamo di disporre di un processore speciale in grado di fondere due array ordinati, aventi complessivamente n elementi, in tempo  $O(\sqrt{n})$ .

Si determini il costo computazionale di un algoritmo di ordinamento, che chiamiamo NEW-MERGESORT, che sia identico all'algoritmo di ordinamento MERGESORT classico, salvo che la fase di fusione dei sotto-array ordinati viene realizzata mediante il processore speciale di cui sopra.

Si scriva l'equazione di ricorrenza di NEW-MERGESORT, spiegandola in dettaglio, e la si risolva con uno dei metodi studiati, specificando i passaggi.

### Soluzione

La nuova eq. sarà

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

Usando il metodo principale abbiamo

- $a = 2, b = 2$
- $f(n) = \Theta(\sqrt{n})$
- $n^{\log_2(2)} = \Theta(n)$
- Siccome  $\Theta(\sqrt{n}) = O(n^{\log_2(2)-\varepsilon})$  per  $\varepsilon = 1/3$  quindi siamo nel primo caso e il costo sarà  $\Theta(n)$

## Esercizio 2

Sia dato un array E di n coppie di interi del tipo (i, e), in cui il primo elemento di ciascuna coppia i è un indice tra 0 ed  $n-1$  assegnato ad una persona, ed il secondo elemento e rappresenta l'età della persona di indice i;

sia poi data una matrice quadrata e simmetrica A di dimensione  $n \times n$  a valori 0 ed 1 in cui  $A[j, k] = A[k, j]$  è uguale ad 1 se e solo se la persona di indice j conosce la persona di indice k.

Si scriva un algoritmo il più efficiente possibile che verifichi se esiste almeno una coppia di conoscenti coetanei.

L'algoritmo dovrebbe dare in output una coppia con gli indici dei due conoscenti coetanei se ce ne sono, oppure la coppia **(0, 0)**.

Ad esempio, sia dato il seguente array  $E: \begin{matrix} 2 & 1 & 3 & 0 \\ 35 & 31 & 35 & 35 \end{matrix}$  e la matrice A:

	0	1	2	3
0	0	1	0	1
1	1	0	1	0
2	0	1	0	0
3	1	0	0	0

L'output è la coppia (0, 3) in quanto queste due persone si conoscono ed hanno la stessa età; si osservi che le coppie (0, 2) e (1, 2) non sono soluzioni valide, infatti le persone nella prima coppia hanno la stessa età ma non si conoscono, mentre quelle nella seconda coppia si conoscono ma non hanno la stessa età.

Si scriva lo pseudocodice opportunamente commentato dell'algoritmo progettato e se ne calcoli formalmente il costo asintotico

### Soluzione

Una sol. è di ordinare E in base all'età e così scorrendo tra di loro gli elem. di E possiamo controllare che nel caso due elem. hanno la stessa età e il loro indice punta ad 1 nella tabella A, allora possiamo ritornare la coppia di conoscenti coetanei.

```
def Exam2_21_1_25(E, A):
    Heap_Sort(E) # ordiniamo E
    for i in range(n-1): # arrivo fino al penultimo elem.
        for j in range(i+1, n): # parto dall'elem dopo i
            if E[i][1] == E[j][1]: # se hanno la stessa età
                if A[ E[i][0] ][ E[j][0] ] == 1: # e sono conoscenti
                    return (E[i][0], E[j][0]) # ritorno il loro indice
    return (0, 0)
```

L'algoritmo di Heap Sort ha costo migliore  $\Omega(n^* \log(n))$  e peggiore  $O(n^2)$ , mentre l'esecuzione dei due array annidati ha costo peggiore  $\Theta(n^2)$  (nel caso non si trovi la coppia) e migliore  $\Theta(1)$  (nel caso sia nella prima pos).

Quindi nel caso migliore di entrambi il costo è  $\Omega(n^* \log(n)) + \Theta(1) = \Omega(n^* \log(n))$  e nel caso peggiore il costo è  $O(n^2) + \Theta(n^2) = O(n^2)$

## Esercizio 3

Sia T un albero binario radicato memorizzato tramite record e puntatori. L'albero è non vuoto e ogni nodo contiene un valore intero come chiave. Definiamo il costo di un cammino radice-foglia come la somma delle chiavi dei nodi che compongono il cammino.

Si progetti un algoritmo **ricorsivo** per trovare il valore minimo del costo di un cammino radice-foglia in un albero T dato in input tramite il puntatore alla sua radice.

Dell'algoritmo proposto:

- a) si dia la descrizione a parole
- b) si scriva lo pseudocodice
- c) si giustifichi formalmente il costo computazionale dando la ricorrenza che lo caratterizza e poi la sua soluzione.

NOTA BENE: nello pseudocodice della funzione ricorsiva **non** si deve far uso di variabili globali.

### Soluzione

Eseguo una visita postordine dell'albero.

Ad ogni nodo eseguo la chiamata ricorsiva per la visita sui figli del nodo che mi ritornerà la lunghezza del cammino dal nodo alle foglie dei sottoalberi.

Se siamo in una foglia ritorniamo direttamente il val. chiave. Altrimenti dobbiamo controllare la lunghezza del cammino nei due figli.

Alla fine prendiamo il val. minimo tra i due figli e aggiungiamo la chiave corrente e ritorniamo il val.

Poiché dobbiamo controllare il val. minimo il val. iniziale dei figli dovrà essere infinito che verrà poi eventualmente sovrascritto dalla chiamata ricorsiva nel caso in cui il figlio esista.

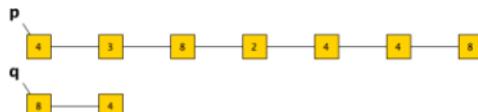
```
def Exam3_21_1_25(r):
    if not r.left and not r.right: # se è una foglia
        return r.key # ritorno la chiave
    sx, dx = float('inf'), float('inf') # inizializzo a infinito il cammino dei figli
    if r.left: # se il figlio esiste
        sx = Exam3_21_1_25(r.left) # controllo la lunghezza del cammino nodo-foglia
    if r.right:
        dx = Exam3_21_1_25(r.right)
    return min(sx, dx) + r.key # ritorno il min cammino dei sottoalberi + la chiave corrente
```

## Esercizio 1

**Esercizio 1 (15 punti):** Abbiamo una lista a puntatori, ciascun nodo della lista è stato generato tramite la classe *Nodo* vista a lezione. Ogni nodo ha quindi ha due campi: il campo *key* contenente interi ed il campo *next* con il puntatore al prossimo nodo della lista (o *None* per l'ultimo nodo).

Progettare un algoritmo che, dato il puntatore *p* alla testa della lista, senza modificarla restituisce il puntatore alla testa di una nuova lista *q* che contiene un nodo per ogni occorrenza che nella lista originaria appare più volte. Le chiavi della lista *q* devono risultare ordinate in modo decrescente.

Di seguito una lista *p* e la lista *q* che l'algoritmo deve restituire.



L'algoritmo deve avere complessità di tempo  $O(n \log n)$  dove  $n$  è il numero di nodi presenti nella lista. Dell'algoritmo proposto:

- si dia la descrizione a parole
- si scriva lo pseudocodice
- si giustifichi il costo computazionale

## Soluzione

Inserisco i valori della lista in un array di appoggio *A* che andrà ad ordinare in ordine crescente.

Creo la lista vuota *q* dandogli *None* come valore iniziale.

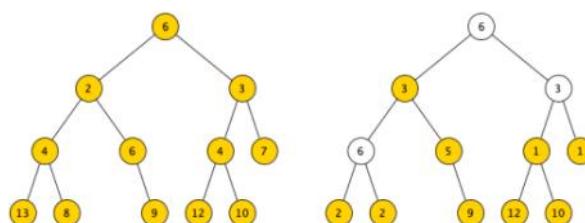
Scorro gli elem. di *A* e per ogni elem. che trovo uguale al successivo allora posso inserirlo nella lista *q*. Per fare in modo di non inserire sempre lo stesso elem. nel caso in cui ci siano più occorrenze, devo inserirlo solo quando passiamo ad un altro numero. Quindi inserisco solo i valori nel caso in cui il successivo è uguale e il precedente è diverso oppure è il primo.

Quindi se trovo un valore valido da poter inserire lo inserisco in testa alla lista di *q*

## Esercizio 2

**Esercizio 2 (15 punti):** Dato il puntatore *r* al nodo radice di un albero binario con nodi contenenti valori interi, progettare un algoritmo ricorsivo che in tempo  $\Theta(n)$  determini il numero di nodi dell'albero che hanno entrambi i figli con lo stesso valore.

Ad esempio, per l'albero in figura a sinistra l'algoritmo deve rispondere 0 perché nessun nodo ha due figli con lo stesso valore. Per l'albero in figura a destra la risposta deve essere 3 grazie alla presenza dei 3 nodi evidenziati in bianco.



L'albero è memorizzato tramite puntatori a record di tre campi: il campo *key* contenente un valore intero ed i campi *left* e *right* con i puntatori al figlio sinistro e al figlio destro, rispettivamente (questi puntatori valgono *None* in mancanza del figlio).

Dell'algoritmo proposto:

- si scriva lo pseudocodice opportunamente commentato;
- si giustifichi il costo computazionale risolvendo l'equazione di ricorrenza prodotta dal vostro algoritmo.

**NOTA BENE:** nello pseudocodice dell'algoritmo ricorsivo **non** si deve far uso di variabili globali.

## Soluzione

Per ogni nodo controlliamo intanto se ha due figli.

- Nel caso in cui abbia 1 o 0 figli allora non possiamo controllare se i figli hanno lo stesso valore, in quel caso continuiamo ritorniamo il valore che ci ritorna dalla ricorsione sull'unico figlio (se esiste).
- Altrimenti se ha 2 figli controlliamo direttamente dal nodo stesso se il valore dei due figli è uguale, in quel caso incrementiamo ritorniamo la somma dei

valori che ci ritorna dalla ricorsione sui figli + 1, per indicare che i figli del nodo hanno lo stesso valore.

# Esercizi Preparatori 2° Esonero

giovedì 29 maggio 2025 11:19

1) qual'è il numero di nodi foglia che puo' avere al massimo un albero quaternario? e qual'è il numero di nodi interni che può avere al massimo?

2) dato il puntatore ad un albero binario memorizzato tramite nodi e puntatori, progettare una funzione che restituisce la rappresentazione dell'albero tramite il vettore dei padri. La procedura deve avere complessità  $O(n)$  dove  $n$  è il numero di nodi.

Soluzione:

Inizializzo l'array vuoto A per contenere la chiave del nodo i e l'array vuoto P per contenere l'indice del padre del nodo i.

Eseguo una visita in preordine:

- Per l'indice utilizzo una variabile "cont" che indica l'indice del nodo i negli array e ad ogni visita sui nodi figli incremento di 1 l'indice che manderò al nodo figlio  
Poiché il nodo figlio sinistro potrebbe avere k con  $1 \leq k < n$  nodi nel suo sottoalbero, dovrà ritornare il contatore aggiornato dopo che abbiamo inserito tutti i nodi del sottoalbero sinistro prima di poter incrementare di 1 il contatore da mandare al figlio destro
- Per il nodo attuale x,

3) lo sbilanciamento di un nodo è il valore assoluto tra il numero di nodi presente nel suo sottoalbero di sinistra ed il numero di nodi presenti nel suo sottoalbero destro. Progettare un algoritmo che dato il puntatore alla radice di un albero rappresentato tramite nodi e puntatori restituisce lo sbilanciamento massimo tra quelli dei suoi nodi. L'algoritmo deve avere complessità  $O(n)$ .

4) dimostrare che in un albero binario di n nodi vale  $h = \Omega(\log n)$