

Algoritmi e Strutture Dati

• Algoritmo, quando e' efficiente?

I METODO: ANALISI Sperimentale, implemento l'algoritmo e vedo quanto impiega a restituire l'output desiderato per input di dimensioni via via crescenti.

SVANTAGGI:

- posso considerare solo un insieme limitato di input;
- perché l'analisi sia scientifica devo ripeterla sotto le medesime condizioni;
- devo prima di tutto implementare l'algoritmo (nel caso in cui l'algoritmo non sia valida ho perso tempo nell'implementazione)

Dunque, serve un **metodo teorico** che:

- permetta di valutare l'**efficienza temporale** di un certo algoritmo, indipendentemente dal software e dall'hardware;
- non richieda l'**implementazione dell'algoritmo**;
- tenga in considerazione tutti i possibili input.

II METODO: ANALISI ASINTOTICA (del caso peggiore)

① Stabilisco quali sono le **operazioni PRIMITIVE/ELEMENTARI** → circuito che COSTO COSTANTE (tempo di esecuzione costante)

② Sommando tutti i costi, ottengo il **TEMPO DI ESECUZIONE** → dipende dall'input

$T(m) \rightarrow$ rappresenta il tempo di esecuzione di un algoritmo su un input di dimensione m .

CASO PEGGIORE: $T(m) = \max$ tempo di esecuzione per un qualsiasi input di dimm m .

Insertion Sort

PSEUDO CODICE

	cost	times	
1	c ₁	1..m	4. $\sum_{j=2}^m t_j \times$
2	c ₂		7. m-1
3	i = j - 1		5. $\sum_{j=2}^m t_{j-1}$
4	while i > 0 and A[i] > key	c ₄	6. $\sum_{j=2}^m t_{j-1}$
5	A[i + 1] = A[i]	c ₅	
6	i = i - 1	c ₆	
7	A[i + 1] = key	c ₇	

- ogni riga di codice abbia costo costante ($c_1 \dots c_7$)
costanti additivi

- devo calcolare quante volte vado ad eseguire ciascuna operazione (times)

- ② while e' un ciclo innestato nel for principale, quindi viene eseguito t_j volte, $1 \leq t_j \leq j$ (lo eseguo al più j volte)
 $\sum_{j=2}^m t_j$ lo eseguo almeno una volta.

iterazioni del for principale

$$T(m) = c_1 m + c_2(m-1) + c_3(m-1) + c_4 \left(\sum_{j=2}^m t_j \right) + c_5 \left(\sum_{j=2}^m t_{j-1} \right) + c_6 \left(\sum_{j=2}^m t_{j-1} \right) + c_7(m-1)$$

- CASO MIGLIORE: eseguo il minimo numero di volte il loop, ovvero 1. da sequenza quindi e' già ordinata

$$t_j = 1 \quad \forall j \in \{2, \dots, m\}$$

$$\therefore T(m) = c_1 m + c_2(m-1) + c_3(m-1) + c_4(m-1) + c_5(m-1) = (c_1 + c_2 + c_3 + c_4 + c_5)(m-1) =$$

$$= a(m-1) \quad \text{per costanti } a, b \in \mathbb{R}$$

↓
fusione lineare

- CASO PEGGIORE: la sequenza di numeri e' ordinata in ordine inverso (decrecente)

$$t_j = j \quad \forall j \in \{2, \dots, m\}$$

$$\therefore \sum_{j=2}^m t_j = \sum_{j=2}^m j = \frac{m(m+1)}{2} - 1$$

$$\sum_{j=2}^m t_{j-1} = \frac{m(m+1)}{2}$$

$$T(m) = c_1 m + c_2(m-1) + c_3(m-1) + c_4 \left(\frac{m(m+1)}{2} - 1 \right) + c_5 \left(\frac{m(m+1)}{2} \right) + c_6 \left(\frac{m(m+1)}{2} \right) + c_7(m-1) = a m^2 + b m + c \quad \text{per costanti } a, b, c \in \mathbb{R}$$

↓
fusione quadratiche

è possibile semplificare: - è importante che gli input siano grandi, i valori minimi non ci interessano.

- 1° SEMPLIFICAZIONE: $a[m] \gg b[m] \gg c$

termine predominante che determina il comportamento del tempo di esecuzione

- 2° SEMPLIFICAZIONE: $O(m^2)$

costante che non influenza il $T(m)$ per valori grandi di m .

il comportamento di $T(m)$ è determinato da m^2 , $T(m) = \Theta(m^2)$.

INVARIANTE DI CICLO: ad ogni iterazione, tutto ciò che viene prima della mia chiave è stato ordinato.

- **INIZIAZIONE**, inizialmente è l'array e vuoto, non ho nessun elemento da confrontare quindi è "ordinato".

- **CONSERVAZIONE**, ad ogni iterazione ragiono tra $j \leq i$ mettendo in ordine senza modificare nulla, quindi è corretto.

- **CONCLUSIONE**, dopo aver finito correttamente tutti i cicli, che dalla CONSERVATION so che funzionano correttamente, ho ordinato gli m elementi.

più generalmente:

INVARIANTE DI CICLO è una proprietà che deve essere o sempre vera o sempre falsa. Viene utilizzata in presenza di cicli.

- cicli ANNOTATI, basta dimostrare la proprietà sul ciclo più esterno che funziona grazie al corretto funzionamento dei cicli più interni.
- cicli DISTINTI, verifico la proprietà su entrambi

Si divide in 3 stadi:

1- **INIZIAZIONE**, controllo che all'inizio la proprietà sia verificata.

2- **CONSERVAZIONE**, dopo aver eseguito il corpo del ciclo prima di ritornare, controllo

3- **CONCLUSIONE**, ho consumato il mio ciclo e ne sono uscito correttamente.

Somma Binaria

Somma di due numeri in una base qualsiasi, in questa forma: $a = (a_{m-1} \dots a_0)_2$ e $b = (b_{m-1} \dots b_0)_2$

oss' da somma di due numeri di m bit in base 2 è un numero di al più $m+1$ bit

oss² Sfruttando la divisione euclidea su a e b posso esprimere la somma del m -esimo elemento come $a_{m-1} + b_{m-1} + r_{m-1}$, dove r_{m-1} è il resto dell'operazione precedente.

Supponiamo gli addendi dati come array di dimensione m : $A[0 \dots m-1]$ e $B[0 \dots m-1]$. Il risultato sarà $C[0 \dots m]$

PSEUDOCODICE

COMPLESSITÀ: $\Theta(m)$

```
x = 0  
for i = 0 to n - 1  
    C[i] = (A[i] + B[i] + x) mod b  
    x = (A[i] + B[i] + x) / b  
C[n] = x
```

CORRENZIA: All'inizio di ogni iterazione del for il sottoarray $C[0 \dots i-1]$ è popolato

correttamente ed è il resto sul bit in posizione i (INVARIANTE)

1- **INIZIAZIONE**, all'inizio del ciclo sono in posizione zero e non ho nulla in posizione $i-1$, quindi è ordinato

2- **CONSERVAZIONE**, alla fine di un'iterazione del ciclo $C[i]$ è aggiornato sempre correttamente

x , inoltre non tocco ciò che si trova in posizione $0 \dots i-1$, quindi è invariante e corretta

3- **CONCLUSIONE**, uscito dal ciclo ho tutti gli elementi corretti da 0 a $m-1$, in posizione m colgo il valore di x evitando

overflow e aggiungendo correttamente la somma.

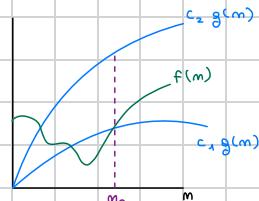
Notazione Asintotica

Utilizziamo la notazione asintotica per descrivere il tempo di esecuzione di un algoritmo, definito in termini di funzioni, il cui dominio è l'insieme dei numeri naturali \mathbb{N} . Osserviamo dunque, come si comporta la funzione di m per valori grandi di m .

- Dato una funzione $g(m)$,

$$\Theta(g(m)) = \left\{ f(m) : \begin{array}{l} \exists c_1, c_2, m_0 > 0 \text{ t.c.} \\ 0 \leq c_1 g(m) \leq f(m) \leq c_2 g(m) \quad \forall m \geq m_0 \end{array} \right\}$$

- Se $f(m) = \Theta(g(m))$, $g(m)$ è un **LIMITE ASINTOTICO STRETTO** per $f(m)$, ovvero $f(m) \sim g(m)$
hanno lo stesso comportamento (differiscono per valori costanti).

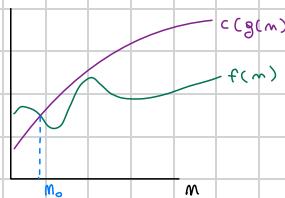


- Dato una funzione $g(m)$, definisco: $O(g(m)) \subsetneq$

$$f(m) : \left\{ \begin{array}{l} \exists c, m_0 > 0 \text{ t.c.} \\ 0 \leq f(m) \leq c \cdot g(m) \quad \forall m \geq m_0 \end{array} \right\}$$

- $O(g(m))$ è il **LIMITE SUPERIORE ASINTOTICO** per $f(m)$

- Dal grafico: per qualsiasi valore m a destra di m_0 , il valore della funzione $f(m)$ coincide, o sta sotto, $c \cdot g(m)$.



ES. 1 $2m^2 = O(m^3)$ Voglio $c, m_0 > 0$ t.c. $2m^2 \leq c m^3 \quad \forall m \geq m_0$
 $f(m) \leq g(m)$

$$\frac{2m^2}{m^3} \leq \frac{c m^3}{m^2} \rightarrow \text{divido per } m^2$$

$$\frac{2}{z} \leq \frac{c}{z} \rightarrow c \text{ è una costante positiva}$$

perciò posso dividere entrambi i membri

$$\frac{2}{c} \leq m \rightarrow \text{prendo } c=2$$

$$\frac{2}{2} \leq m$$

$1 \leq m \Rightarrow \text{prendo } m_0 = 1$

ES. 2 $m^3 \neq O(m^2) \rightarrow$ uso una dim. x ASSURDO per ottenere una contraddizione.

$$\downarrow$$

$$m^3 = O(m^2)$$

$$\Rightarrow \exists c, m_0 > 0 \text{ t.c. } m^3 \leq c m^2 \quad \forall m \geq m_0$$

$$\Rightarrow \frac{m^3}{m^2} \leq \frac{c m^2}{m^2} \quad m \leq c \quad \forall m \geq m_0 \quad \Rightarrow \text{l'assunto } m^3 = O(m^2) \text{ è falso}$$

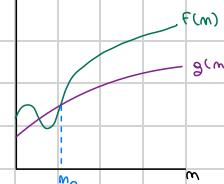
sono costanti

- Dato una funzione $\Omega(g(m))$ definisco:

$$\Omega(g(m)) = \left\{ f(m) : \begin{array}{l} \exists c, m_0 > 0 \text{ t.c.} \\ 0 \leq c \cdot g(m) \leq f(m) \quad \forall m \geq m_0 \end{array} \right\}$$

- $\Omega(g(m))$ è il **LIMITE INFERIORE ASINTOTICO** per $f(m)$

ES. $m^3 = \Omega(m^2)$ Voglio $c, m_0 > 0$ t.c. $\frac{c m^2}{m^3} \leq 1 \quad c \leq m$



Teorema: $\Theta(g(m)) = O(g(m)) \cap \Omega(g(m))$

Poi ogni coppia di funzioni $f(m)$ e $g(m)$ si ha $f(m) = \Theta(g(m))$, se e solo se $f(m) = O(g(m))$ e $f(m) = \Omega(g(m))$

OSS: Non tutte le funzioni sono asintoticamente confrontabili

$$\text{es. } m \neq O(m^{1+\sin m})$$

perché oscilla tra 0 e 2

$$m \neq \Omega(m^{1+\sin m})$$

Lemma: Siano $f(m)$ e $g(m)$ funzioni tali che $\lim_{m \rightarrow \infty} \frac{f(m)}{g(m)} = c$ per un certo numero reale $c > 0$.

Allora $f(m) = \Theta(g(m))$

E.S. $\lim_{m \rightarrow \infty} \frac{3m^3 + 180m^2 - m + 1000}{m^3} = \lim_{m \rightarrow \infty} \frac{m^3(3 + \frac{180}{m} - \frac{1}{m^2} + \frac{1000}{m^3})}{m^3} = 3$



• **NOTAZIONE O:** Data una funzione $g(m)$,

$$O(g(m)) = \left\{ f(m) : \forall c > 0 \exists m_0 > 0 \text{ t.c. } \begin{array}{l} 0 \leq f(m) \leq c \cdot g(m) \quad \forall m \geq m_0 \end{array} \right\}$$

Se $f(m) = O(g(m))$, $g(m)$ è LIMITE SUPERIORE NON ASINTOTICAMENTE STRETTO per $f(m)$

E.S. $2m = O(m^2)$ ma $2m^2 \neq O(m^2) \rightarrow$ significa che il tempo di esecuzione asintotico di $2m$ è strettamente inferiore a quello di m^2

$$\forall c > 0 \exists m_0 > 0 \text{ t.c. } \frac{2m}{m^2} \leq \frac{cm^2}{m^2} \Rightarrow 2 \leq cm \Rightarrow \frac{2}{c} \leq m \quad \forall m \geq m_0 \quad m_0 = \frac{2}{c}$$

OSS. - $\lim f(m) = O(g(m))$ il limite $0 \leq f(m) \leq c \cdot g(m)$ vale per qualche costante $c > 0$

- $\lim f(m) = O(g(m))$ il limite $0 \leq f(m) \leq c \cdot g(m)$ vale per tutte le costanti $c > 0$. da funzione $f(m)$ diventa insignificante rispetto a $g(m)$ quando m tende all'infinito.

• **NOTAZIONE W:** Data una funzione $g(m)$

$$W(g(m)) = \left\{ f(m) : \forall c > 0 \exists m_0 > 0 \text{ t.c. } \begin{array}{l} 0 \leq c \cdot g(m) \leq f(m) \quad \forall m \geq m_0 \end{array} \right\}$$

Se $f(m) = W(g(m))$, $g(m)$ è LIMITE INFERIORE NON ASINTOTICAMENTE STRETTO per $f(m)$.

E.S. $\frac{m^2}{m} = W(m)$ ma $\frac{m^2}{2} \neq W(m^2)$
 $\frac{m^2}{m} = W(m) \quad \forall c > 0 \exists m_0 > 0 \text{ t.c. } m < \frac{m^2}{m} \quad \frac{m}{m} < \frac{m^2}{m} \cdot M \quad \frac{M}{m} < m^2 \quad 1 < M$

Lemma: Se $\lim_{m \rightarrow \infty} \frac{f(m)}{g(m)} = 0$, allora $f(m) = O(g(m))$.
Se $\lim_{m \rightarrow \infty} \frac{f(m)}{g(m)} = \infty$, allora $f(m) = W(g(m))$. NON MEASSONTEMENTE

Proprietà utili: • Se $f(m) = O(g(m))$ & $g(m) = O(h(m))$, allora $f(m) = O(h(m))$.

• Se $f(m) = \Omega(g(m))$ & $g(m) = \Omega(h(m))$, allora $f(m) = \Omega(h(m))$.

• Sia k costante fissa $\epsilon f_1(m), \dots, f_k(m)$, $g(m)$ funzioni t.c. $f_i(m) = O(g(m)) \forall i$

Allora $f_1(m) + \dots + f_k(m) = O(g(m))$. (vale anche per Θ)



Suddividiamo i tempi di esecuzione in famiglie: POLINOMIALI, LOGARITMICHE, ESPONENZIALI

Lemma: Sia $f(m)$ un polinomio di grado d con coefficiente di grado massimo positivo $a_d m^d + a_{d-1} m^{d-1} + \dots + a_0$ con $a_d > 0$

Allora $f(m) = \Theta(m^d)$

Diciamo che un algoritmo ha tempo di esecuzione **POLINOMIALE** se (*nel caso peggiore*) $T(m) = O(m^d)$, per una certa costante d (indipendente da m).

Nota: Per dire che il t. di esecuzione è polinomiale non è necessario che $g(m)$ sia un polinomio

E.S. $T(m) = O(m \log m)$, allora $T(m) = O(m^2)$

e' un tempo polinomiale

Rule of thumb: Funzioni logaritmiche crescono più lentamente di polinomiali che a loro volta crescono più lentamente di esponenziali.

RECALL: . Funzioni logaritmiche $f(m) = \log_b m$ b, r costanti

. Funzioni esponenziali $f(m) = r^m$

costante > 0
se $a, b > 1$

Nota: la base del logaritmo non è importante in notazione asintotica. Inoltre, passo da base b a base a con $\log_a m = \frac{1}{\log_b a} \cdot \log_b m$

$$\Rightarrow \log_a m = O(\log_b m) \quad \forall a, b > 1$$

Mentre la base dell'esponenziale conta. Non è vero che, per $r > s > 1$, $r^m = O(s^m)$ $* r^m = O(s^m)$, $r^m = \Omega(s^m)$

• **DIMOSTRAZIONE A ASSURDO:** $r^m = O(s^m)$ (per $r > s$, $r^m = \Omega(s^m)$)

$$\text{Allora } \exists c, m_0 > 0 \text{ t.c. } r^m \leq c s^m \quad \forall m \geq m_0 \quad r^m \leq c s^m \quad \frac{r^m}{s^m} \leq c \quad \forall m \geq m_0$$

FALSO
Riesco sempre a trovare un m sufficientemente grande

$$\text{t.c. } \left(\frac{r}{s}\right)^m > c$$

Lemma: $\forall d, k > 0 \quad (\ln m)^k = O(m^d)$

polinomiale
 $m^k = O((1+d)^m)$

DIM. uso l'HOPITAL: $\begin{cases} \text{se } \lim_{m \rightarrow \infty} f(m) = \infty, \lim_{m \rightarrow \infty} g(m) = \infty \\ \text{allora } \lim_{m \rightarrow \infty} \frac{f(m)}{g(m)} = \lim_{m \rightarrow \infty} \frac{f'(m)}{g'(m)} \end{cases}$

$$\cdot \lim_{m \rightarrow \infty} \frac{(\ln m)^k}{m^d} = \frac{1}{m^d} \rightarrow \text{prop. limiti}$$

$$\lim_{m \rightarrow \infty} f(m)^k = \left(\lim_{m \rightarrow \infty} f(m) \right)^k \quad k > 0$$

$$= \left(\lim_{m \rightarrow \infty} \frac{\ln m}{m^{d/k}} \right)^k = \lim_{m \rightarrow \infty} \frac{1/m}{(d/k)m^{d/k-1}} = \left(\lim_{m \rightarrow \infty} \frac{1}{(d/k)m^{d/k}} \right)^k = 0$$

SCALA: $\log m = m - m - m \log m = m^2 - m^3 - 2^m - m!$

ricerca binaria

algo di scorr.
(max di array)
merge di due
sequenze ordinate

Multipl. Matr.

cicli
ognidai

moltiplicaz.
matrici
insieme
delle parti

permutez.

Analogia fra il confronto asintotico di due funzioni e il confronto fra due numeri:

$$f(m) = O(g(m)) \rightarrow a \leq b \quad f(m) = o(g(m)) \rightarrow a < b$$

$$f(m) = \Omega(g(m)) \rightarrow a \geq b \quad f(m) = \omega(g(m)) \rightarrow a > b$$

$$f(m) = \Theta(g(m)) \rightarrow a = b$$

Esercizio d'esame: ordinare la seguente sequenza di funzioni $f: \mathbb{N} \rightarrow \mathbb{R}$ in modo che ognuna sia O della successiva:

$$\sqrt{m}, m^2 \log m, 2^{\sqrt{m}}, 2^{\log m^2}, 10^m$$

Semplifico le $f(m)$ le più possibili:

$$2^{\log m^2} = 2^{2 \log m} = (2^{\log m})^2 = m^2$$

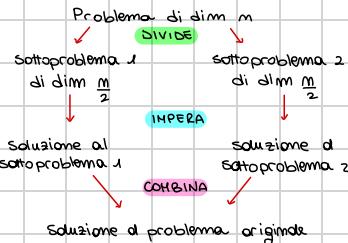
$$\square (a^{\log b} = b)$$

$$\sqrt{m}, m^2, m^2 \log m, 2^{\sqrt{m}}, 10^m$$

Divide et Impera

È un paradigma basato sulla **ricorsione**, ovvero l'algoritmo richiama se stesso per trattare sottoproblemi dello stesso tipo, infine ne combina le soluzioni per creare una soluzione unica del problema originale. Ad ogni livello di ricorsione sono presenti tre passi:

- **DIVIDE**, il problema viene diviso in sottoproblemi, ovvero istanze più piccole dello stesso problema iniziale;
- **IMPERA**, i sottoproblemi vengono risolti in modo ricorsivo.
(Caso Base): quando hanno una dimensione sufficientemente piccola vengono risolti direttamente.
- **COMBINA**, le soluzioni vengono combinate per generare la soluzione del problema originale.



da ricorsione finisce quando la sequenza da ordinare ha lunghezza 1, in quanto è già ordinata.

Metodi risolutivi: • METODO DI SOSTITUZIONE :

- DUE PASSI : - si stima l'ordine di grandezza asintotico per $T(m)$
- si dimostra, per induzione su m , la correttezza dell'ordine di grandezza stimato.

Il metodo si rivela utile quando si ha già un'idea della soluzione alla ricorrenza studiata.

• ALBERO DI RICORSIONE :

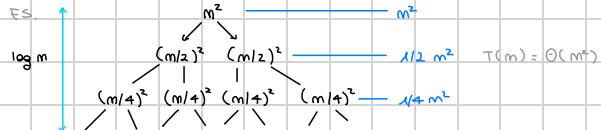
Sono un modo conveniente per visualizzare i passi di sostituzione necessari per risolvere una ricorrenza.

E' formato da : - una radice, ovvero l'input con dimensione m ;

- le foglie, rappresentanti i minimi sottoproblemi ottenibili;

- L'ALTEZZA, ovvero la distanza massima tra radice e foglia.

E' utile per semplificare i calcoli ed evidenziare le condizioni limite della ricorrenza. Ogni nodo rappresenta il costo di un sottoproblema. E' un ottimo metodo per ottenere una buona ipotesi, che poi viene verificata con il metodo di sostituzione.



Master Theorem

E' utilizzato per risolvere le ricorrenze della forma: $T(m) = aT(m/b) + f(m)$

$m \rightarrow$ dimensione del problema

$T(m/b) \rightarrow$ tempo nel quale vengono risolti i m modi ricorsivo i sottoproblemi

$a \rightarrow$ numero di sottoproblemi

$m/b \rightarrow$ dimensione dei sottoproblemi

$f(m) \rightarrow$ costo per il DIVIDE e COMBINA

• 3 CASI:

1) Se $f(m) = O(m^{\log_b \theta - \epsilon})$ (con $\epsilon > 0$ cost), allora $T(m) = \Theta(m^{\log_b \theta})$;

2) Se $f(m) = O(m^{\log_b \theta})$, allora $T(m) = \Theta(m^{\log_b \theta} \log_2 m)$

3) Se $f(m) = \Omega(m^{\log_b \theta + \epsilon})$ (con $\epsilon > 0$ cost) e se $a.f(\frac{m}{b}) \leq c.f(m)$ per $c > 1$ cost. e m sufficientemente grande, allora $T(m) = \Theta(f(m))$.

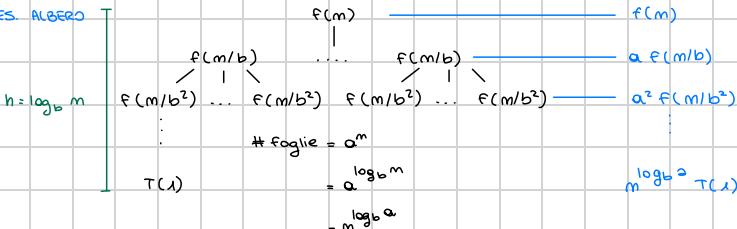
• $a \geq 1$, $b \geq 1$ costanti

• $f(m)$ FUNZIONE ASINTOTICAMENTE POSITIVA

- In tutti i casi confrontiamo $f(m)$ con $m^{\log_b 2}$, la soluzione è determinata dalla più grande tra le due funzioni.
 - Nel secondo caso, le due funzioni hanno la stessa dimensione, la soluzione allora viene moltiplicata per un fattore logaritmico ($\log m$)
- Nel primo caso $f(m)$ deve essere POLINOMIALMENTE/ASINTOTICAMENTE più piccola di $m^{\log_b 2}$ per un fisso $m^{\log_b 2}$
- Nel terzo caso $f(m)$ deve essere POLINOMIALMENTE/ASINTOTICAMENTE più grande di $m^{\log_b 2}$, in più deve soddisfare la condizione di "regolarità".

N.B. Questi tre casi non coprono tutte le funzioni possibili!

ES ALBERO



- caso 1** il costo totale è dominato dai costi delle foglie.
- caso 2** il costo totale è equamente distribuito tra i livelli dell'albero.
- caso 3** la maggior parte dei costi sono in testa all'albero, che ha costo $f(n)$.

MERGE-SORT (A, p, r)

```
if p < r
    q = [(p + r) / 2]
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q+1, r)
    MERGE(A, p, q, r)
```

MERGE(A, p, q, r)

```
n1 = q - p + 1 -> dim A[p..q]
n2 = r - q -> dim A[q+1..r]
create arrays L[1.. n1 + 1] and R[1.. n2 + 1]
for i = 1 to n1
    do L[i] = A[p + i - 1] copia A[p..q] in L[1..n1]
for j = 1 to n2
    do R[j] = A[q + j] copia A[q+1..r] in R[1..n2]
L[n1 + 1] = 00 sentinel, utilizzate per fare in modo che il numero in posizione n+1 in L e R non possa essere il numero più piccolo a meno che entrambi gli array R[n2 + 2] = 00 non abbiano esposto le loro sentinelle
i = 1
j = 1
for k = p to r
    do if L[i] <= R[j]
        then A[k] = L[i]
            i = i + 1
        else A[k] = R[j]
            j = j + 1
```

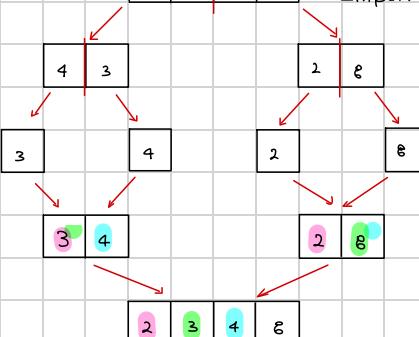
L'algoritmo opera nel seguente modo:

- **DIVIDE**, divide la sequenza degli elementi da ordinare in due sotto-sequenze di $n/2$ elementi ciascuna.
- **IMPERA**: ordina le sotto-sequenze in modo ricorsivo.
- **COMBINA**, fonde le due sotto-sequenze ordinate per generare un'unica sequenza ordinata del problema originale.

Invariante di ciclo: All'inizio di ogni iterazione **for**, $A[p..k-1]$ contiene, ordinati, i primi $k-p$ elementi più piccoli di $L[1..n1+1]$ e di $R[1..n2+1]$. Inoltre, $L[i]$ e $R[i]$ sono i più piccoli elementi del loro array non ancora copiati in A .

ES.

Input: $A:[4, 3, 2, 6]$ $p=1, r=4$



Correttezza di MERGE:

- INITIALIZATION:** alla prima iterazione del ciclo $for k = p$, quindi $A[p..k-1]$ è vuoto.
- CONSERVATION:** supponendo che $L[i] \leq R[j]$, $L[i]$ è l'elemento più piccolo che non è ancora stato copiato in A . Dopo che $L[i]$ viene copiato in $A[k]$, A contiene i $k-p+1$ elementi più piccoli. Successivamente viene incrementato k ristabilendo l'invariante di ciclo per la prossima iterazione. Vale la medesima cosa se $R[j] \leq L[i]$.
- CONCLUSION:** alla fine del ciclo $k = r+1$. Per l'invariante di ciclo il sotto array $A[p..k-1]$, ovvero $A[p..r]$, contiene $k-p = r-p+1$ elementi ordinati.

MERGE viene eseguito in $\Theta(m)$, con $m = r - p + 1$

- i cicli for che copiano $A[p \dots q]$ in $L[1 \dots m_1]$ e $A[q+1 \dots r]$ in $R[1 \dots m_2]$ impiegano un tempo $\Theta(m_1 + m_2) = \Theta(m)$
- il tutto ciclo for ha m iterazioni, quindi $\Theta(m)$

DIM. X INDUZIONE → CASO BASE: Se $p > r$, ha al massimo un elemento ed è quindi già ordinato

→ PASSO INDUTTIVO: il passo divide calcola un indice q che separa A : - $A[p \dots q]$ contiene $\lceil m/2 \rceil$ elementi
- $A[q+1 \dots r]$ contiene $\lfloor m/2 \rfloor$ elementi

T(m) MERGE-SORT

MERGE-SORT $A[1 \dots m]$

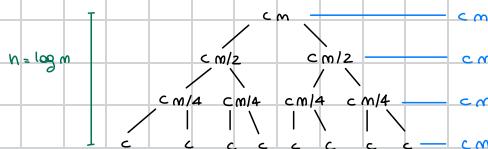
$\Theta(1)$ 1. IF $m=1$, done

$2\Theta(m/2)$ 2. Recursively sort $A[1 \dots \lceil m/2 \rceil]$ and $A[\lceil m/2 \rceil \dots m]$

$\Theta(m)$ 3. "MERGE" the 2 sorted lists.

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

ALBERO DI RICORSIONE



Il costo totale è $c'm(\log m + 1) = c'm\log m + cm = \Theta(m\log m)$

DIT PER SOSTITUZIONE

$$T(m) = \begin{cases} c & \text{se } m=1 \\ 2T(m/2) + cm & \text{se } m>1 \end{cases}$$

① $T(m) = \Theta(m \log m)$

② $T(m) = \Omega(m \log m)$

④ Claim: $T(m) \leq c'm\log m \quad \forall m \geq m_0$

Dim: $m=1, T(m)=c \rightarrow$ per def $T(m)=c$ se $m=1$

$$c'm\log m = 0 \quad [\log 1 = 0]$$

Non è un problema poiché $T(m) \leq c'm\log m$ deve essere vera solo per m sufficientemente grandi

\rightsquigarrow se $m_0=2$, il caso base diventa $m=2$

$$T(2) = 2T(2/2) + 2c = 2T(1) + 2c = 2c + 2c = 4c$$

$$4c \leq c'2\log 2 \quad 4c \leq 2c' \Rightarrow \text{sì, se } c'=2c$$

$$4c \leq 2(2c) \rightarrow 4c \leq 4c \checkmark$$

per ($m_0=2, c'=2c$):

$$T(m) \leq c'm\log m \rightarrow 2T(m/2) + cm \leq c'm\log m \rightarrow 2c + 2c \leq 2c \cdot 2\log 2 \rightarrow 4c \leq 4c \checkmark$$

P. INDUTTIVO: $c'=2c \quad m > 2$

$$T(m) = 2T(m/2) + cm \leq 2\left(c' \frac{m}{2} \log \frac{m}{2}\right) + cm \rightarrow = 2\left(2c \frac{m}{2} \log \frac{m}{2}\right) + cm$$

$$\rightarrow = 2\left(2c\left(\frac{m}{2} \log \frac{m}{2} - \frac{m}{2} \log \frac{m}{2}\right)\right) + cm \rightarrow = 4c\left(\frac{m}{2} \log \frac{m}{2} - \frac{m}{2}\right) + cm$$

$$\rightarrow \frac{2c}{2} \left(\frac{m}{2} \log \frac{m}{2}\right) - \frac{2c}{2} \frac{m}{2} + cm \rightarrow = 2c(m \log m) - 2cm + cm \rightarrow \leq 2c(m \log m) - cm \checkmark$$

$T(m) \leq O(m \log m)$

$$2c(m \log m) \leq c'm \log m$$

$$2c(m \log m) \leq 2c(m \log m) \checkmark \Rightarrow T(m) = \Theta(m \log m)$$

$$② T(m) = \Omega(m \log m)$$

claim¹: $T(m) \geq c'm \log m \quad \forall m > m_0 \rightarrow$ non sempre l'ipotesi induktiva è abbastanza forte

claim²: $T(m) \geq c'm \log m + c'm \quad \forall m > m_0$

↳ rafforzato il bound con un termine di stima inferiore

DIM: C. BASE) $m=1 \rightsquigarrow c = T(1) \quad c \geq c' \rightarrow$ si, se prendo $c=c'$

$$\text{P. INDUTTIVO) } m > 1 \quad T(m) \geq 2T\left(\frac{m}{2}\right) + cm \geq 2\left(c'\frac{m}{2} \log \frac{m}{2} + c\frac{m}{2}\right) + cm \rightarrow = 2\left(c'\frac{m}{2} \log m - c'\frac{m}{2} \log 2 + c\frac{m}{2}\right) + cm$$

$$\rightarrow = 2\left(c'\frac{m}{2} \log m - c'\frac{m}{2} + c\frac{m}{2}\right) + cm \rightarrow = 2c'\frac{m}{2} \log m - 2c'\frac{m}{2} + 2c\frac{m}{2} + cm$$

$$T(m) \geq c'm \log m + cm$$

$$\rightarrow = c'm \log m - c'm + cm + cm \rightarrow = c'm \log m - c'm + 2cm$$

$$c'm \log m - c'm + 2cm \geq c'm \log m + cm$$

$$-c'm + 2cm \geq cm$$

$$-c'm \geq cm - 2cm$$

$$\Rightarrow T(m) = \Omega(m \log m)$$

$$-c'm \geq -cm$$

$$\frac{c'm}{cm} \leq \frac{c'm}{cm}$$

$$c' \leq c \Rightarrow \text{Si, se prendo } c=c'$$

➡ MERGE-SORT ha complessità $\Theta(m \log m) \Rightarrow$ è un ALGORITMO OTTIMO!

Binary Search

La ricerca binaria è un algoritmo di ricerca veloce con complessità $O(\log n)$.

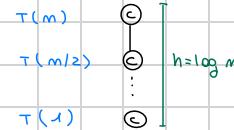
Funziona secondo il principio Divide et Impera e affinché funzioni correttamente la sequenza di dati fornita deve essere ordinata. Si occupa di trovare un elemento target nel seguente modo:

- 1) DIVIDE: controlliamo l'elemento di mezzo (mid);
- 2) IMPERA: se l'elemento target è minore del mid allora andremo a cercare il target nel sotto array di sinistra, mentre se il target è maggiore del mid, lo andremo a cercare nel sotto array di destra.
- 3) COMBINA: triviale.

```
BINARY-SEARCH (A, target, low, high)
if low > high
    return false
else
    mid = (low + high) / 2
    if target == A[mid]
        return mid
    else if target > A[mid]
        return BINARY-SEARCH(A, target, mid + 1, high)
    else
        return BINARY-SEARCH(A, target, low, mid - 1)
```

$$T(m) = T(m/2) + O(1)$$

DIM CON ALBERO DI RICORSIONE



Mi aspetto quindi $T(m) = O(\log m)$

DIM. x SOSTITUZIONE

$$T(m) = \begin{cases} c & \text{se } m=1 \\ T(m/2) + c & \text{se } m>1 \end{cases}$$

Claim: $T(m) = O(\log m) \Rightarrow ① T(m) \leq O(\log m)$

② $T(m) \geq \Omega(\log m)$

① Claim: $T(m) \leq c'(\log m) \quad \forall m \geq m_0$

DIM: prendo $m_0 = 2$ e c. base dell'induzione $m=2$

$$T(2) = T(1) + c = c + c = 2c \quad 2c \leq c' \log 2 \quad 2c \leq c' \rightarrow \text{Sì, per } c'=2c$$

P. INDUTTIVO) $m \geq 2 \Leftrightarrow c' \geq 2c$

$$\begin{aligned} T(m) &= T(m/2) + c \leq 2c \left(\log \frac{m}{2}\right) + c \\ &= 2c(\log m - \log 2) + c \\ &= 2c \log m - 2c + c \\ &\leq 2c \log m - c \end{aligned}$$

$\Rightarrow 2c \log m - c \leq c' \log m$
 $2c \log m - c \leq 2c \log m \quad \Rightarrow T(m) = O(\log m)$

② $T(m) = \Omega(\log m) \quad \forall m \geq m_0$

Claim: $T(m) \geq c' \log m$

Dim: $m = 1$

$$T(1) \geq c \quad c \geq c' \log 1 \quad c \geq 0 \quad /$$

P. INDUTTIVO: $m > 1$

$$\begin{aligned} T(m) &= T(m/2) + c \geq c' \log m \\ &= c' \left(\log \frac{m}{2}\right) + c = c' (\log m - \log 2) + c \\ &\geq c' \log m - c' + c \end{aligned}$$

$\Rightarrow c' \log m - c' + c \geq c' \log m$
 $c \geq c' \rightarrow \text{sì, per } c = c'$

$$\Rightarrow T(m) = \Omega(\log m)$$

$\Rightarrow T(m) = \Theta(\log m) \rightarrow \text{BINARY SEARCH è OTTIMO!}$

Potenza di un numero

Prende in input una base a e il suo esponente m , calcola a^m .

$T(m)$ BRUTE FORCE: $\Theta(m)$, con il principio dei passi possiamo ottimizzare a $T(m) = \Theta(\log m)$.

In effetti ad ogni iterazione osserviamo l'esponente e a seconda che sia pari o dispari iteriamo.

$$T(m) = \begin{cases} a^{\frac{m}{2}} \cdot a^{\frac{m}{2}} & \text{se } m \text{ pari} \\ a^{\frac{m-1}{2}} \cdot a^{\frac{m-1}{2}} \cdot a & \text{se } m \text{ dispari} \end{cases}$$

$$T(m) = T(m/2) + \Theta(1) \Rightarrow T(m) = \Theta(\log m)$$

l'algoritmo è ottimo!

```
POWER (a, n)
if n = 0
    return 1
if n % 2 == 0
    return POWER(a, n/2) * POWER(a, n/2)
else
    return POWER(a, (n-1)/2) * POWER(a, (n-1)/2) * a
```

Moltiplicazione Binaria

INIZIALMENTE l'algoritmo di moltiplicazione aveva limite inferiore $\Omega(m^2)$:

Siano x e y due interi positivi in binario

$$x = \boxed{x_L} \quad \boxed{x_R} = 2^{m/2} x_L + x_R \quad x \cdot y = 2^m (x_L y_L) + 2^{m/2} (x_L y_R + x_R y_L) + x_R y_R$$

$$y = \boxed{y_L} \quad \boxed{y_R} = 2^{m/2} y_L + y_R$$

$\Rightarrow T(m) = 4T(m/2) + \Theta(m)$, non otteniamo alcun guadagno e l'algoritmo è inefficiente.

KARATSUBA'S TRICK: ridurre il numero di sottoproblemi da 4 a 3.

KARATSUBA (X, Y, n)

```
if n = 1
    return X*Y
else
    m = n / 2
    compute XL, XR, YL, YR
    a = KARATSUBA(XL, YL, m)
    b = KARATSUBA(XR, YR, m)
    c = KARATSUBA(XL + XR, YL + YR, m)
    return  $2^m a + 2^m c - 2^m b$ 
```

$$T(m) = 3T(m/2) + \Theta(m)$$

$$T(m) = O(m^{\log 3}) \rightarrow$$
 ancora non è ottimo.

Inoltre è possibile moltiplicare in $O(m \log m)$

Max Sottoarray

Dato un array di dimensione m , trovare un sottoarray i cui valori hanno somma max.

- INIZIALE se A contiene solo valori positivi o negativi

- BRUTE FORCE: $T(m) = \Theta(m^2)$ prova tutti i Sottoarray $\binom{m}{2}$

Utilizzando D&I viene fatta una divisione bilanciata troncando il punto medio mid. $A[\text{low} \dots \text{high}]$ — $A[\text{low} \dots \text{mid}]$

Ogni sottoarray contiguo $A[i \dots j]$ deve tralasciare:

- interamente nel sottoarray $A[\text{low} \dots \text{mid}]$, quindi $\text{low} \leq i \leq \text{mid}$

$A[\text{mid}+1 \dots \text{high}]$

$\text{low} \leq i \leq j \leq \text{mid}$

- interamente nel sottoarray $A[\text{mid}+1 \dots \text{high}]$, quindi $\text{mid} < i \leq j \leq \text{high}$

- passante per il punto di mezzo, quindi $\text{low} \leq i \leq \text{mid} \leq j \leq \text{high}$

FIND-MAX-CROSSING-SUBARRAY ($A, \text{low}, \text{mid}, \text{high}$)

```
1. left-sum = -∞
2. sum = 0
3. for i = mid down to low
4.     sum = sum + A[i]
5.     if sum > left-sum
6.         left-sum = sum
7.         max-left = i
8. right-sum = -∞
9. sum = 0
10. for j = mid up to high
11.     sum = sum + A[j]
12.     if sum > right-sum
13.         right-sum = sum
14.         max-right = j
15. return (max-left, max-right, left-sum + right sum)
```

righe 1-7: trova il max-sottoarray della metà sinistra $A[\text{low} \dots \text{mid}]$

left-sum → somma più grande trovata finora

sum → somma elementi di $A[\text{low} \dots \text{mid}]$

righe 8-14: operano in modo analogo sulla metà destra $A[\text{mid}+1 \dots \text{high}]$

$A[\text{mid}+1 \dots \text{high}]$

righe 15: restituisce gli indici "max-left" e "max-right" che

delimitano un massimo sottoarray che include il punto di mezzo e la somma "left-sum + right-sum" dei valori del sottoarray $A[\text{max-left} \dots \text{max-right}]$

Se A contiene m elementi ($m = \text{high} - \text{low} + 1$) $\Rightarrow T(m) = \Theta(m)$

2 for $\rightarrow (3-7)$ $\text{mid-low} + 1$ iterazioni

(10-14) $\text{high}-\text{mid}$

$$\Rightarrow (\text{mid-low} + 1) + (\text{high}-\text{mid}) = \text{high} - \text{low} + 1 = m$$

```

FIND-MAXIMUM-SUBARRAY (A, low, high)
1. If high == low
2.   return (low, high, A[low])
3. else
4.   mid = (low+high)/2
5.   (left-low, left-high, left-sum) = FIND-MAXIMUM-SUBARRAY(A, low, mid)
6.   (right-low, right-high, right-sum) = FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
7.   (cross-low, cross-high, cross-sum) = FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
8.   if left-sum >= right-sum && left-sum >= cross-sum
9.     return (left-low, left-high, left-sum)
10.  else if right-sum >= left-sum && right-sum >= cross-sum
11.    return (right-low, right-high, right-sum)
12.  else
13.    return (cross-low, cross-high, cross-sum)

```

CORRETTEZZA: FIND-MAXIMUM-SUBARRAY(A, 1, n) trova un max sottoarray di A.

Per induzione sulla dimensione di A: se $n = 1$ ✓

Altrimenti, l'ipotesi induttiva implica che l'algoritmo trova un max sottoarray di $A[1..mid]$ e $A[mid + 1..n]$, mentre con FIND-MAX-CROSSING-SUBARRAY troviamo un max sottoarray che contiene $A[mid, mid + 1]$.

Viene alla fine restituito il max di questi tre sottoarray

COMPLESSITÀ: $T(m) = \begin{cases} O(1) & \text{se } m=1 \\ 2T(m/2) + O(m) & \text{se } m>1 \end{cases}$ \Rightarrow Stessa di MERGE-SORT, infatti $T(m) = O(m \log m)$

Counting Inversions

Dato una permutazione $\langle 1, 2, \dots, m \rangle$ determinare il numero di inversioni. Il costo corrisponde approssimativamente al numero di inversioni, cioè il numero di coppie i cui ordini sono scambiati nelle due permutazioni.

BRUTE FORCE: $T(m) = O(m^2)$ → controlla tutte le coppie

com Divide et Impera:

DIVIDE: separa la lista in 2 (costo: $O(1)$)

IMPERA: conta ricorsivamente il numero di inversioni in ciascuna delle metà ($2T(m/2)$) (e ordino l'array)

COMBINA: conta il numero di inversioni quando unisce le due metà, calcolando la somma di queste e di quelle ottenute nell'IMPERA.

x il COMBINA uso il MERGE di MERGE-SORT $\rightarrow (T(m) = O(m))$

mentre ordino conto → MERGE AND COUNT

Merge and Count



6	5	4	3	2	1	
8	3	7	10	14	18	19

6	2	11	16	17	23	25
6	3	2	2	0	0	0

total: $6+3+2+2+0+0$

SORT-AND-COUNT (L)

```

1. if list has one element
2.   return 0 and the list L
3. Divide the list into two halves A and B
4. (rA, A) = SORT-AND-COUNT(A)
5. (rB, B) = SORT-AND-COUNT(B)
6. (r, L) = MERGE-AND-COUNT(A, B)
7. return r = rA + rB + r and the sorted list L

```

PRE-CONDIZIONI x MERGE AND COUNT: A e B sono ordinati.

POST-CONDIZIONI x SORT AND COUNT: L è ordinato.

$$T(m) = 2T(m/2) + O(m) \Rightarrow T(m) = O(m \log m)$$

Closest Pair of Points

Dati n punti su un piano, trovare la coppia con la minima distanza euclidea fra loro, assumendo che nessuno coincida.

BRUTE FORCE: $T(n) = \Theta(n^2)$, controllo tutte le copie.

In una dimensione riesco ad arrivare a $(m \log m)$.

Dopo un primo tentativo in cui proviamo a dividere il nostro piano in quattro parti ci accorgiamo che è impossibile assicurarsi che ognuno di essi abbia $m/4$ punti in ogni parte.

- L'idea scelta del divide è quindi quella di disegnare una linea verticale che divide approssimativamente il piano in due parti aventi all'inizio $m/2$ punti. Il costo è di $\Theta(m \log m)$ poiché devo ordinare in maniera crescente le coordinate x dei punti (per poi dividere dato che il numero è più o meno $m/2$).
- L'IMPERA invece cerca la coppia più vicina in ciascuno dei sottoproblemi. (PATTINA ricorsiva)
- IL COMBINA troverà la coppia più vicina avendo un punto in una parte e un punto nell'altra, poi poi ritornare la migliore tra le tre soluzioni. Per funzionare bene COMBINA trova, se esiste, la distanza in modo tale che sia più piccola della minima tra le due restituite dalle due chiamate ricorsive a sx e dx. Ordino i punti in B in ordine crescente di coordinate x e controlla solo le coppie che hanno distanza minore di quella selezionata.

Closest-Pair(p_1, \dots, p_n) {

 Compute separation line L such that half the points are on one side and half on the other side.

$O(n \log n)$

$\delta_1 = \text{Closest-Pair(left half)}$

$2T(n/2)$

$\delta_2 = \text{Closest-Pair(right half)}$

$\delta = \min(\delta_1, \delta_2)$

$O(n)$

 Delete all points further than δ from separation line L

$O(n \log n)$

 Sort remaining points by y-coordinate.

$O(n \log n)$

 Scan points in y-order and compare distance between each point and next 11 neighbors.

$O(n)$

 If any of these distances is less than δ , update δ .

 return δ .

Quicksort

È basato sul paradigma D&I. Il tempo di esecuzione nel peggior caso è $\Theta(n^2)$. Il caso medio invece è $\Theta(n \log n)$. dovuto in loco, sull'array stesso.

- ① divide, partiziono l'array $A[p \dots r]$ in due regioni $A[p \dots q-1]$ e $A[q+1 \dots r]$, tali che ogni elemento di $A[p \dots q-1]$ sia minore o uguale a $A[q]$ che, a sua volta, è minore o uguale a ogni elemento di $A[q+1 \dots r]$
- ② IMPERA, ordino i 2 sottodarray chiamando ricorsivamente Quicksort.
- ③ COMBINA, gli array sono già ordinati

QUICKSORT(A, p, r)

```
1. if  $p < r$ 
2.     then  $q = \text{PARTITION}(A, p, r)$ 
3.     QUICKSORT( $A, p, q-1$ )
4.     QUICKSORT( $A, q+1, r$ )
```

PARTITION(A, p, r)

```
1.  $x = A[r]$ 
2.  $i = p-1$ 
3. for  $j = p$  to  $r-1$ 
4.     do if  $A[j] \leq x$ 
5.         then  $i = i + 1$ 
6.             scambia  $A[i]$  con  $A[j]$ 
7. scambia  $A[i+1]$  con  $A[r]$ 
8. return  $i+1$ 
```

- All'inizio di ogni iterazione del ciclo (3-6) per qualsiasi indice K dell'array:
- 1) Se $p \leq K \leq i$, allora $A[K] \leftarrow x$
 - 2) Se $i+1 \leq K \leq j-1$, allora $A[K] > x$
 - 3) Se $K = i$, allora $A[K] = x$

COMPLESSITÀ: • CASO PEGGIOR: $T(m) = T(m-1) + T(0) + \Theta(m) = T(m-1) + \Theta(m) \rightarrow T(m) = \Theta(m^2)$

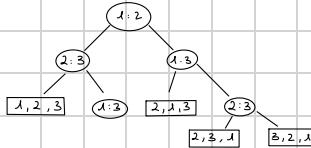
la routine di partizionamento produce un sottoproblema con $m-1$ elementi e uno con 0 elementi.

- CASO MIGLIORE: $T(m) = 2T(m/2) + \Theta(m) \rightarrow T(m) = O(m \log m)$
- CASO MEDIO: $T(m) = T(9m/10) + T(m/10) + cm \rightarrow T(m) = O(m \log m)$

Albero di Decisione

Un albero di decisione è un grafo di decisioni e delle loro possibili conseguenze, utilizzato per creare un "piano di azioni" mirato ad uno scopo. Un albero di decisione è costruito al fine di supportare l'azione decisionale.

- è un sistema con n variabili in input e m variabili in output
- Le variabili in input (attributi) sono derivate dall'osservazione dell'ambiente. Le variabili in output, invece, identificano la decisione/ azione da intraprendere.
- Il processo decisionale è rappresentato con un albero logico rovesciato dove ogni nodo è una funzione condizionale. Ogni nodo verifica una condizione (test) su una particolare proprietà dell'ambiente (variabile) e ha due o più diramazioni verso il basso.
- Il processo consiste in una sequenza di test. Comincia sempre dal nodo radice, il nodo genitore situato più in alto nella struttura, e procede verso il basso.
- A seconda dei valori rilevati in ciascun nodo, il flusso prende una direzione oppure un'altra e procede progressivamente verso il basso.
- La decisione finale si trova nei nodi foglia terminali, quelli più in basso.



STABILITÀ: Un algoritmo di ordinamento è stabile se preserva l'ordine iniziale tra due elementi dello stesso valore.

Counting Sort

È un algoritmo basato sul conteggio (non sul confronto dei valori).

Counting Sort suppone che ci sia un array di input compreso fra 1 e K (è necessario sapere il range).

L'algoritmo determina per ogni elemento di input x , il numero di elementi minori di x . Ad esempio, se ci sono 17 elementi minori di x , allora x dovrà essere inserito nella posizione 18 dell'array di output. Dovremo gestire il caso in cui più elementi hanno lo stesso valore per evitare che più valori vengono inseriti nella stessa posizione.

input: $A[1 \dots m]$ di appoggio: $C[0 \dots K]$

output: $B[1 \dots m]$ ordinato

STEP. → si contano le occorrenze di ogni valore del range $1 \dots K$ e si inserisce questo conteggio nell'array di appoggio C ,

in modo da sapere il numero di elementi che precedono ogni valore del range nell'array.

→ Si scelgono gli elementi del vettore di potenza A , dall'ultimo e si inseriscono nella corretta posizione in base alle info ottenute nello step precedente, scritte in C .

quante volte appare l'indice di C nell'array A

A	7	2	2	7	7	1	4	5	3	2

$K=7$ $[1 \dots 7]$

1	2	3	4	5	6	7
1	3	1	1	1	0	3

Dopo aver registrato le occorrenze di ogni elemento, in ogni posizione andremo a mettere la somma del valore della posizione corrente più quella della posizione precedente

1 2 3 4 5 6 7

C 1 4 5 6 7 7 10

→ ci dice esattamente per ogni posizione quanti elementi minori o uguali all'indice esistono all'interno di A, quindi ci dice dove posizionare Ciascun'indice

A 7 2 2 7 7 1 4 5 3 3 2

1 2 3 4 5 6 7

C 1° 4² 5³ 6⁴ 7⁵ 7⁶ 7 10⁷

↓
il valore corrispondente all'indice indica dove posizionare l'indice in B e andiamo a decrementare

1 2 3 4 5 6 7 8 9 10

B 1 2 2 2 3 4 5 7 7 7

• Counting Sort è **stabile**: riusciamo a mantenere l'ordine degli elementi con lo stesso valore.

```
COUNTING-SORT (A, B, k)
θ(k)
1. for i = 0 to k
   C[i] = 0 //inizializzazione di C
2. for j = 1 to A.length
   C[A[j]] = C[A[j]] + 1 //conteggio delle occorrenze di ogni elemento di A
3. for i = 2 to k
   C[i] = C[i] + C[i-1] //nella posizione i di C, si mette il numero di elementi <= a i
4. for j = A.length down to 1
   B[C[A[j]]] = A[j]
   C[A[j]] = C[A[j]] - 1 //scrittura del vettore ordinato B e decremento dei valori di C
```

$$T(m) = \Theta(m+k)$$

Radix Sort

E' un algoritmo di ordinamento basato sui seguenti concetti:

- ① non si basa sul confronto degli elementi
- ② parte dall'analisi della cifra meno significativa, raggruppando gli elementi in "contenitori" (bucket), in base al valore della cifra che si sta analizzando (bucket 0..9)
- ③ gli elementi vengono inseriti all'interno del bucket in ordine di appartenenza dell'array
- ④ effettua un numero di passaggi pari al numero di cifre dell'elemento maggiore
- ⑤ in termini di efficienza viene utilizzato quando l'array da ordinare ha dimensione estremamente elevata.

↓
V 85 71 56 34 107 211 86 64

$$1) V[i] \% m \quad 85 \% 10 = 5$$

$$2) (V[i] \% m) / q \quad 5 / 9 = 0 \rightarrow \text{bucket corrispondente}$$

m = q variabili: m = 10 e q = 1

bucket:

B[0]

B[2]

B[4] → 34 → 64

B[6] → 56 → 86

B[8]

B[1] → 71 → 211

B[3]

B[5] → 85

B[7] → 107

B[9]

↓
V 71 211 34 64 85 56 86 107

$$1) V[i] \% m \quad 71 \% 10 = 7$$

$$2) (V[i] \% m) / q \quad 7 / 9 = 0$$

$$m^k = 10 \rightarrow m = 100 \quad q = 10 \rightarrow q = 10 \Rightarrow \text{andiamo ad esaminare le decime}$$

i bucket precedenti possono essere svuotati così da poterli riempire nuovamente

$B[0] \rightarrow 107$ $B[2] \rightarrow$ $B[4] \rightarrow$ $B[6] \rightarrow 64$ $B[8] \rightarrow 85 \rightarrow 86$ $B[1] \rightarrow 211$ $B[3] \rightarrow 34$ $B[5] \rightarrow 56$ $B[7] \rightarrow 71$ $B[9]$

↓

\vee	107	211	34	56	64	71	85	86
--------	-----	-----	----	----	----	----	----	----

$1) V[i] \% m$

$107 \% 1000 = 107$

$2) (V[i] \% m) / q$

$107 / 100 = 1$

$m^* = 10 \rightarrow m = 1000$

$q^* = 10 \rightarrow q = 100$

 $B[0] \rightarrow 34, 56, 64, 71, 85, 86$ 

34	56	64	71	85	86	107	211
----	----	----	----	----	----	-----	-----

 $B[1] \rightarrow 107 \quad B[2] \rightarrow 211$ **RADIX-SORT (A, d)**

- for $i = 1$ to d
- usa un ordinamento stabile per ordinare l'array A sulla cifra i (es. Counting Sort)

LEMMA: Dati m numeri di d cifre, dove ogni cifra può avere fino a k valori possibili, la procedura Radix-Sort ordina correttamente i numeri nel tempo $\Theta(d(m+k))$, se l'ordinamento stabile utilizzato dalla procedura impiega un tempo $\Theta(m+k)$.

- # chiamate di counting-sort: d

\Rightarrow costo totale è $\Theta(d(m+k))$

- costo per chiamata: $\Theta(m+k)$

LEMMA: Dati m numeri di b bit e un intero positivo $r \leq b$, Radix-Sort ordina correttamente questi numeri nel tempo $\Theta((b/r)(m+2^r))$ se l'algoritmo di ordinamento stabile usato richiede tempo $\Theta(m+k)$ per input nell'intervallo da 0 a K .

DIH: per un valore $r \leq b$, consideriamo ogni chiave come se avesse $d = \lceil b/r \rceil$ cifre di r bit ciascuna. Ogni cifra è un numero intero compreso nell'intervallo da 0 a $2^r - 1$, quindi possiamo utilizzare counting sort con $K = 2^r - 1$. Ogni passaggio di counting sort richiede il tempo $\Theta(m+k) = \Theta(m+2^r)$, poiché ci sono d passaggi; il tempo di esecuzione totale è $\Theta(d(m+2^r)) = \Theta((b/r)(m+2^r))$.

Voglio $r \leq b$ che minimizzi $f(r) = \frac{b}{r} (m+2^r)$

con r grande, meno chiomate ma, se $r > \log m$, $T(m,b)$ esponenziale.

1 $b < \lfloor \log m \rfloor$. Allora $m+2^r = \Theta(m) \rightsquigarrow$ prendo $r = b \rightsquigarrow T(m,b) = \Theta(m)$ ottimo.

2 $b \geq \lfloor \log m \rfloor$. Prendo $r = \lfloor \log m \rfloor \rightsquigarrow T(m,b) = \Theta\left(\frac{bm}{\log m}\right) = \Theta\left(\frac{m \log K}{\log m}\right)$

$K \leq 2^b - 1$

Massimo Sottoarray in O(n)

Link video utile yt

oss. max Sottoarray ha un certo indice $dx \rightarrow V_d$, tanto max Sottoarray il cui indice dx è j .

$S[j] = \text{somma max Sottoarray che termina im } A[j]$

sottoproblema

$S[1] = A[1]$

max Sottoarray che termina im $A[j+1]$

contiene $A[j]$ (formato da $A[j+1]$ e max Sottoarray che termina in $A[j+1]$)

i		j	$j+1$		m		

$$\Rightarrow S(j+1) = \max \{ A[j+1], A[j+1] + S[j] \}$$

il valore massimo,

combinando il

sottoproblema

non contiene $A[j]$ (dato essere $A[j+1]$)

combinando la sommabile

(KADANE)

```
MAX-SUBARRAY (A)
1. S[1] = A[1]
2. max-right = 1 // indice dx del sottoarray massimo
3. for j = 2 to n
4.     if S[j-1] + A[j] > A[j]
5.         S[j] = S[j-1] + A[j]
6.     else
7.         S[j] = A[j]
8.     if S[j] > S[max-right]
9.         max-right = j
10.    max-right = j
11. return S[max-right]
```

Dynamic Programming

DeI: suddivido il problema in sottoproblemi INDEPENDENTI, risolvo in modo ricorsivo i sottoproblemi e combino. Quando però i sottoproblemi fanno in comune altri sottoproblemi, il DeI li risolve ripetutamente facendo più lavoro del necessario.

DP: crea sottoproblemi non indipendenti, avendo fanno sottoproblemi in comune. Costruisce soluzioni a sottoproblemi via via più grandi.

Infatti mentre DeI risolverebbe ripetutamente i sottoproblemi comuni, con DP risolve ciascun sottoproblema una volta e salva la soluzione in una tabella, evitando ricalcoli.

Fibonacci Sequence

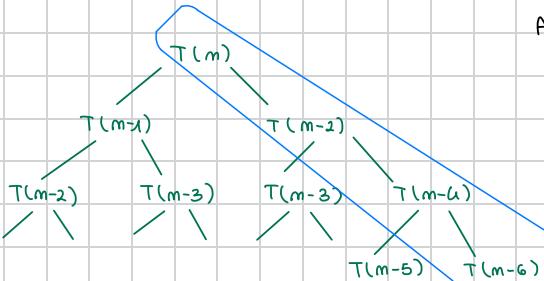
• Sequenza definita da: $a_1=1$ $a_2=1$ $a_m=a_{m-1}+a_{m-2}$

quindi otteniamo 1, 1, 3, 5, 8, 13, 21, 34...

• Algo ricorsivo Fib(n)

```
1. if n = 1 or n = 2, then
2.     return 1
3. else
4.     a = Fib(n-1)
5.     b = Fib(n-2)
6.     return a+b
```

$$T(m) = T(m-1) + T(m-2) + 1$$



Assumo per semplicità m dispari

$$m, m-2, m-4, m-6, \dots, 1$$

$$K \text{ t.c. } m-2k=1 \quad K = \frac{m-1}{2}$$

$$\Rightarrow h \geq \frac{m-1}{2}$$

$$\Rightarrow T(m) \geq 1 + 2 + 2^2 + \dots + 2^h$$

$$= 2^{\frac{m+1}{2}} - 1$$

$$= 2^{\frac{m}{2}} - 1$$

$$= (\sqrt{2})^m - 1$$

parte meno profonda dell'albero

esponentiale! :-)

oss. l'algoritmo ricorsivo risolve solo $m-1$ sottoproblemi, per ottimizzarlo l'algoritmo possiamo memorizzare i valori restituiti dalle chiamate ricorsive in una sottotabella.

Quindi otteniamo: Fib(n)

```
1. if n = 1 or n = 2, then
2.     return 1
3. else
4.     f[1] = 1; f[2] = 1
5.     for i = 3 to n
6.         f[i] = f[i-1] + f[i-2]
7.     return f[n]
```

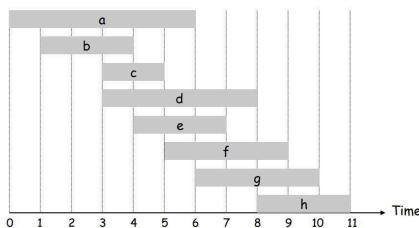
$$T(m) = O(m)$$

Weighted Interval Scheduling

PROBLEMA: Abbiamo una lista di lavori con un tempo d'inizio e di fine, ed un "peso" associato ad esso. L'obiettivo e' di

trovare il maximum weight subset di lavori reciprocamente compatibili.

due lavori sono compatibili se non si
sovrappongono



Da completare!

Tipo di dato ADT

INSIEMI DINAMICI: possono crescere, ridursi o cambiare

Operazioni tipiche:

- inserire elementi nell'insieme
- cancellare elementi dall'insieme
- verificare l'appartenenza di un elemento all'insieme

Un **dizionario** è un insieme dinamico che supporta le tre operazioni sopra citate.

È un insieme S di coppie (elem, key), dove key appartiene ad un insieme totalmente ordinato (per es. N). Le tre operazioni quindi diventano:

- INSERT(e, k), aggiunge (e, k) a S
- DELETE(k), rimuove da S coppia con chiave k
- SEARCH(k), se l'elemento non è presente ritorna NIL

STRUTTURA DATI: organizzazione che permette di supportare in modo efficiente le operazioni di un certo tipo di dato.

- strutture indirizzate (array)
- strutture collegate (liste)

ARRAY:

- collezione di celle numerate (con n celle gli indici vanno da 1 a n)
- accesso in lettura e scrittura ad una qualsiasi cella in tempo costante
- proprietà:
 - forte: gli indici delle celle sono indici consecutivi
 - debole: non è possibile aggiungere nuove celle ad un array a meno di riallocarlo

STACK o QUEUES: insiemi dinamici che consentono INSERT e DELETE. L'elemento rimosso con delete è predeterminato.

STACK:

- usa schema LIFO
- è possibile rimuovere solo l'oggetto inserito più recentemente
- pop: estrazione
- push: inserimento
- bisogna gestire i casi di underflow e overflow
- internet browser, editor di testo

QUEUE:

- usa schema FIFO
- è possibile rimuovere solo l'oggetto che si trova in coda da più tempo
- dequeue: estrazione
- enqueue: inserimento
- chiamate a centri di servizio, liste d'attesa

LINKED LIST: nodi non contigui in memoria, possono essere:

- liste semplicemente concatenate: ha un puntatore al nodo successivo della lista (next), è uno al primo elemento (1.head)
- liste doppiamente concatenate: oltre a next e head, ha prev, ovvero un puntatore al nodo precedente

Come gestire in modo efficiente insiemi di oggetti?

Siamo interessati ad insiemi dinamici: perché possono crescere, ridursi o combaciare

Operazioni tipiche su elementi: 1) inserire elem. in un insieme

2) cancellare gli elementi da un insieme

3) verificare l'appartenenza di un elemento all'insieme

DIZIONARIO → insieme dinamico che supporta i punti 1,2,3 (tipi di dato ADT)

☰

Insieme S di coppie (`elem, key`), dove ogni Key appartiene ad un insieme totalmente ordinato (per es. IN)

`INSERT(e, k)` aggiunge (`e, k`) ad S

`DELETE(k)` rimuove da S coppia con chiave k

`SEARCH(k)` se la chiave k è presente, restituisce l'elemento corrispondente, altrimenti NIL.

Strutture Dati: organizzazione che permette di supportare in modo efficiente le operazioni di un certo tipo di dato

① ARRAY strutture indirizzate

② LISTE strutture collegate

① ARRAY • collezione di celle numerate (n celle \Rightarrow indici tra 1 e n)

• accesso in lettura e scrittura ad una qualsiasi cella in tempo costante

Proprietà: • **FORTE**, indici delle celle sono numeri consecutivi

• **DEBOLE**, non è possibile aggiungere nuove celle ad un array, per farlo bisogna riallocare

Possiamo realizzare un dizionario con gli array?

S → array di dim m, contiene le coppie (`elem, key`) ordinate rispetto a key

`INSERT(e, k)`: 1) rialloca S aumentando la dimensione a $m+1$

2) trova il più piccolo i t.c. $K \leq S[i].key$

complessità: $O(m)$

3) $\forall j$ da $m+1$ a $i+1$, $S[j] \leftarrow S[j-1]$

4) $S[i] \leftarrow (e, k)$

`DELETE(k)`: 1) trova l'indice i della coppia con chiave k

2) $\forall j$ da i a $m-1$, $S[j] \leftarrow S[j+1]$

complessità: $O(m)$

3) rialloca S diminuendo la dimensione a $m-1$

`SEARCH(k)` BINARY-SEARCH su S per vedere se contiene la chiave k o meno

complessità: $O(\log m)$

- Operazioni tipiche su insiem dinamici*
- **SEARCH(S, k)**
A query that, given a set S and a key value k , returns a pointer x to an element in S such that $x.key = k$, or NIL if no such element belongs to S .
 - **INSERT(S, x)**
A modifying operation that augments the set S with the element pointed to by x . We usually assume that any attributes in element x needed by the set implementation have already been initialized.
 - **DELETE(S, x)**
A modifying operation that, given a pointer x to an element in the set S , removes x from S . (Note that this operation takes a pointer to an element x , not a key value.)
 - **MINIMUM(S)**
A query on a totally ordered set S that returns a pointer to the element of S with the smallest key.
 - **MAXIMUM(S)**
A query on a totally ordered set S that returns a pointer to the element of S with the largest key.
 - **SUCCESSOR(S, x)**
A query that, given an element x whose key is from a totally ordered set S , returns a pointer to the next larger element in S , or NIL if x is the maximum element.
 - **PREDECESSOR(S, x)**
A query that, given an element x whose key is from a totally ordered set S , returns a pointer to the next smaller element in S , or NIL if x is the minimum element.

① di modifica
② interrogazione (query)

Stack (pile) e Queue (code)

Imsiemi dinamici che consentono INSERT e DELETE. L'elemento uscito con DELETE è predeterminato

• **STACK** usa schema LIFO (last in first out)

↳ possibile rimuovere solo l'oggetto inserito più recentemente

• **QUEUE** usa schema FIFO (first in first out)

↳ possibile rimuovere solo l'oggetto che si trova in coda da più tempo

implementazione di **STACK** con **ARRAY**

STACK_EMPTY(S)

if $S.top == 0$

return true

else

return false

PUSH(S, x)

$S.top = S.top + 1$

$S[S.top] = x$

(insert)

POP(S)

if **STACK_EMPTY(S)**

error "underflow"

else

$S.top = S.top - 1$

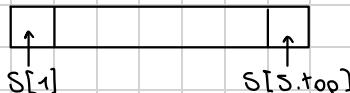
return $S[S.top + 1]$

(delete)

Stack di m elementi $S[1 \dots m]$

$S.top$ = indice dell'elemento inserito più recentemente

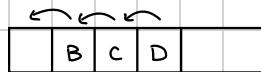
complessità $O(1)$



Implementazione di queue con array



DELETE:  $Q[1 \dots m]$



$O(m)$

Possibile farlo in $O(1)$?

Sì, introduco la variabile head che punta al primo elemento della coda.



$O(1)$ ma potrai "esaurire" array
con numero di m elementi
 \leadsto array circolare

ENQUEUE (Q, x)

```

 $Q[Q.\text{tail}] = x$ 
if  $Q.\text{tail} == Q.\text{length}$ 
     $Q.\text{tail} = 1$ 
else  $Q.\text{tail} = Q.\text{tail} + 1$ 

```

DEQUEUE (Q)

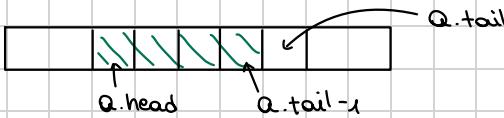
```

 $x = Q[Q.\text{head}]$ 
if  $Q.\text{head} == Q.\text{length}$ 
     $Q.\text{head} = 1$ 
else  $Q.\text{head} = Q.\text{head} + 1$ 
return x

```

Coda di $m-1$ elementi $Q[1 \dots m]$

$Q.\text{head}$ = inizio della coda ; $Q.\text{tail}$ = posizione in cui il prossimo elemento sarà inserito



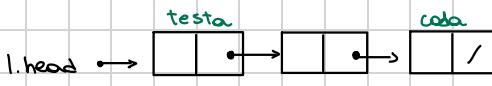
occlime circolare (l dopo m)

All'inizio: $Q.\text{head} = Q.\text{tail} = 1$. Se $Q.\text{head} = Q.\text{tail} + 1$, la coda è piena.

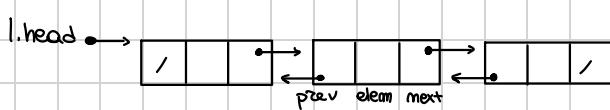
Liste Concatenate

(linked lists)

modi non contigui in memoria



lista semplicemente concatenata

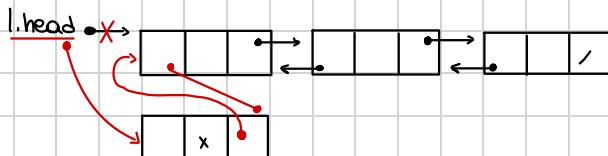


lista doppialmente concatenata

prev, next: puntatori

l.head: puntatore al primo elemento della lista (testa)

Inserire / Cancellare un elemento in testa? O(1)



X Accedere all'iesimo elemento $\rightarrow \Theta(i)$

Heap

- Struttura dati composta da un array che possiamo considerare come un albero binario quasi completo.
- Ogni nodo dell'albero corrisponde a un elemento dell'array.
- Tutti i livelli sono completamente riempiti tranne eventualmente l'ultimo che può essere riempito da sinistra fino ad un certo punto.
- Un array A(heap) ha due attributi:
 - A.length → numero di elementi nell'array
 - A.heap-size → numero degli elementi dell'heap che sono registrati nell'array A

↓

Anche se ci possono essere degli elementi memorizzati in tutto l'array $A[1 \dots A.length]$, soltanto i numeri in $A[1 \dots A.heap-size]$, dove $0 \leq A.heap-size \leq A.length$, sono elementi validi dell'heap

- root $A[1]$

- se i è l'indice di un nodo, gli indici di suo padre $PARENT(i)$ }
 del figlio destro $RIGHT(i)$ } sono facili da calcolare
 del figlio sinistro $LEFT(i)$ }

$PARENT(idx)$	$LEFT(idx)$	$RIGHT(idx)$
$return idx/2$	$return 2 * idx$	$return (idx * 2) + 1$

- Ci sono due tipi di heap binari: - MAX-HEAP

- MIN-HEAP

In entrambi i modi devono soddisfare una proprietà dell'heap:

- per MAX-HEAP: ogni nodo i diverso dalla radice $\Rightarrow A[PARENT(i)] > A[i]$, ovvero il valore di un nodo è al massimo il valore di suo padre. Infatti l'elemento più grande di un MAX-HEAP è nella radice.
- per MIN-HEAP: ogni nodo i diverso dalla radice $\Rightarrow A[PARENT(i)] < A[i]$. Infatti il più piccolo elemento in un MIN-HEAP è nella radice. (è spesso utilizzato nelle code di priorità)

- Un heap in forma di albero:

- altezza di un nodo: il cammino più lungo che va dal nodo fino ad una foglia
- altezza di un heap: l'altezza della sua radice

Poiché un heap è basato su un albero binario completo, la sua altezza è $\Theta(\log n)$; inoltre le operazioni fondamentali sugli heap vengono eseguite in un tempo che è al massimo proporzionale all'altezza:

- MAX-HEAPIFY, procedura eseguita in $O(\log n)$, serve per conservare la proprietà del MAX-HEAP

- **BUILD-MAX-HEAP**, eseguita in tempo lineare, genera un MAX-HEAP da un array di input non ordinato
- **HEAP-SORT**, eseguita in $O(n \log n)$, ordina sul posto un array
- **MAX-HEAP-INSERT**, **HEAP-EXTRACT-MAX**, **HEAP-INCREASE-KEY**, **HEAP-MAXIMUM**, eseguite in $O(\log n)$ permettono ad un heap di essere utilizzato come una coda di priorità.

HEAPSORT (A)

```

1. BUILD-MAX-HEAP(A) // in questo modo l'elemento maggiore si trova nella radice
2. for i = A.length down to 2 do // itera partendo dalla fine dell'array
3.   swap A[1] with A[heap-size] // scambia il primo elemento (il maggiore) con l'ultimo
4.   A.heap-size = A.heap-size - 1 // diminuisce la dimensione dell'heap in quanto l'elemento maggiore è stato
                                spostato in fondo
5.   MAX-HEAPIFY(A, 1) // per garantire che la proprietà del MAX-HEAP sia mantenuta e che il massimo elemento sia la
                        radice

```

MAX-HEAPIFY: -input: A (array), indice i:

- quando viene chiamata assume che gli alberi binari con radice in $\text{LEFT}(i)$ e $\text{RIGHT}(i)$ siano max-heap, ma che $A[i]$ possa essere più piccolo dei suoi figli, violando la proprietà del MAX-HEAP. Quindi, fa scendere il valore $A[i]$ in modo che il sottoalbero con radice di indice i diventi un max-heap. Ad ogni passo viene determinato il più grande tra $A[i]$, $A[\text{LEFT}(i)]$ e $A[\text{RIGHT}(i)]$, l'indice del maggiore viene memorizzato:
- se $A[i]$ è il più grande, allora il sottoalbero con radice nel modo i è un max-heap e la procedura termina
- se uno dei due figli è l'elemento maggiore, $A[i]$ viene scambiato con $A[\text{max}]$ così da rispettare la proprietà: la funzione viene chiamata ricorsivamente

MAX-HEAPIFY (A, i)

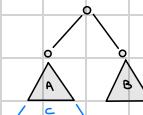
```

1. l = LEFT(i)
2. r = RIGHT(i)
3. if l <= A.heap-size and A[l] > A[i]
4.   massimo = l
5. else massimo = i
6. if r <= A.heap-size and A[r] > A[i]
7.   massimo = r
8. if massimo != i
9.   swap A[i] with A[massimo]
10.  MAX-HEAPIFY(A, massimo)

```

worst case: A e B sono uguali, ma l'albero è sbilanciato sulla

$$\text{sinistra. } A+B+C=N \quad A+C = \frac{2}{3}n$$



Il tempo di esecuzione in un sottoalbero di dimensione m con radice in un modo i è $\Theta(1)$ per sistemare le relazioni fra gli elementi $A[i], A[\text{LEFT}(i)], A[\text{RIGHT}(i)]$ più il tempo per eseguire MAX-HEAPIFY in un sottoalbero con radice in uno dei figli del modo i (chiamata ricorsiva). I sottoalberi dei figli hanno ciascuno una dimensione che non supera $2m/3$.

+ caso peggiore: l'ultima ziga dell'albero è piena esattamente a metà e la ricorsione diventa:

$$T(m) \leq T(2m/3) + \Theta(1) \Rightarrow T(m) = O(\log m)$$

Possiamo usare MAX-HEAPIFY dal basso verso l'alto per convertire un array $[1..m]$ con $m = A.length$, in un MAX-HEAP. Gli elementi nel Sottoarray $A[(\lfloor m/2 \rfloor + 1) ... m]$ sono foglie dell'albero dunque ciascuno di essi è un heap di un solo elemento che possiamo usare come punto di partenza. Quindi partiamo dalle foglie e applichiamo MAX-HEAPIFY in modo rimontante dell'albero.

BUILD-MAX-HEAP (A)

```

1. A.heap-size = A.length
2. for i = A.length down to 1     $\Rightarrow O(n \log n)$ 
3.   MAX-HEAPIFY(A, i)

```

BUILD-MIN-HEAP (P)

```

1. A.heap-size = A.length
2. for i = (A.length/2) down to 1
3.   MIN-HEAPIFY(A, i)

```

MIN-HEAPIFY (A, i)

```

1. l = LEFT(i)
2. r = RIGHT(i)
3. if l <= A.heap-size and A[l] < A[i]
4.   min = l
5. else
6.   min = i
7. if r <= A.heap-size and A[r] < A[i]
8.   min = r
9. if min != i
10.  swap A[i] with A[min]
11.  MIN-HEAPIFY(A, min)

```