

Lo sapere?

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

$$\sum_{n=3}^m n = \sum_{n=1}^m n(-1-2)$$

$$\sum_{n=1}^{m+1} n = \sum_{n=1}^m n + m + 1$$

# ALGORITMI DI RICERCA

**RICERCA SEQUENZIALE:** scarto tutto l'array con un while, se sono troppo (cerco  $K$ ) ritorno -1 altrimenti  $i$ .

**CASO MIGLIORE:**  $K$  è in prima posizione

il tempo è  $T_m(m) = \mathcal{O}(1)$

**CASO PEGGIORE:**  $K$  non è nell'array

il tempo è  $T_p(m) = \mathcal{O}(m)$

**RICERCA BINARIA (DICOTOMICA):** dato un array ordinato, trovo la metà e la confronto con  $K$ :

## TEMPI

1) se è maggiore allora confronto

la metà di sinistra



Caso migliore:  $K$  è in  $V[m]$

$T(m) = \mathcal{O}(1)$

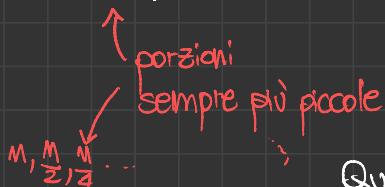
2) Se è minore allora confronto

Caso peggiore:  $K$  non è nel vettore

la metà di destra

$T(m) = \mathcal{O}(\log m)$

porzioni  
sempre più piccole



3) Se è uguale ritorna  $m$

QUESTE OPERAZIONI SI RIPETONO CON UN WHILE



# ALGORITMI di ordinamento

**SELECTION SORT:** si trova il primo minimo e si mette in posizione 1, si scatta e si trova il nuovo minimo

STABILE: NO, non mantiene l'ordine

IN LOCO: SÌ, non usa → due for immediati

APPORTEI

- ① Scatto fino a fine array-1 (i)
- | ② Scatto da i+1 a fine array (j)
  - ↳ cerca il minimo
  - ↳ scambio A[i] col minimo

$$T(m) = 5(m-1) + \sum_{i=1}^{m-1} i + t_{if} = 5(m-1) + 3 \frac{(m-1)m}{2} + t_{if}$$
$$\approx 5m + \frac{3}{2}m^2 + t_{if}$$

Caso migliore: il vettore è già ordinato

$$T_m(m) = 5m + \frac{3}{2}m^2 + \theta \sim \mathcal{O}(m^2)$$

Caso peggiore: il vettore è ordinato in modo decrescente

$$t_{if} \sim \frac{n^2}{2} \longrightarrow T_p(m) \sim \mathcal{O}(m^2)$$

$$\text{Quindi } T(m) = \Theta(m^2)$$

**INSERTION SORT** → Si parte dal secondo elemento, si controlla

STABILE: sì, mantiene  
l'ordine

IN-PLACE: sì, non ha  
struttura d'appoggio

se il valore precedente è più piccolo e in caso  
positivo si scambiano

↳ quindi avviene finché quello  
prima di lui è minore o  
uguale

### Codice

```
void INSERTION-SORT (A [])
```

```
for i := 2 to A.length
```

(m)

```
J := i - 1
```

(m)

```
key := A[i]
```

(m)

```
while J >= 1 AND A[J] > key
```

$$\sum_{i=2}^m t_{wi} *$$

```
A[J+1] := A[J] *
```

```
A[J] := key *
```

```
J := J - 1 *
```

$$T(m) = 3m + 4 \sum_{i=2}^m t_{wi}$$

Caso migliore: il vettore  
è già ordinato

$$t_{wi} = 0 \quad T(m) = \underline{m} (m)$$

Caso peggiore: il vettore  
è decrescente

$$t_{wi} = J = i - 1$$

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

$$t_p(m) = 3m + \sum_{i=2}^m i - 1 = 3m + \sum_{i=2}^{m-1} i$$

$$= 3m + \frac{(m-1)m}{2} \sim 3m + 2m^2$$

DIVIDI et IMPERA

→ si divide il problema in più sotto problemi DIVIDE

→ si risolve ciascun sotto problema ricorsivamente IMPERA

→ si uniscono le soluzioni per la soluzione completa CONQUISTA

$$T(n) = D(n) + I(n) + C(n)$$

TEOREMA DELL'ESPERTO (da usare nel DIVIDI ET IMPERA)

per  $T(n)$  t.c.

quante ricorrenze?

quante sottodivisioni?

parte iterativa

per  $n^{log_b a}$

1) se  $n^{log_b a}$  è assintoticamente superiore a  $f(n)$ , allora

$$f(n) = O(n^{\log_b a - \varepsilon}) \text{ per } \varepsilon > 0$$

$$\text{quindi } T(n) = \Theta(n^{\log_b a})$$

2) se  $n^{\log_b a}$  è asintoticamente uguale a  $f(n)$

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

3) se  $n^{\log_b a}$  è asintoticamente inferiore allora

1)  $f(n) = \Omega(n^{\log_b a + \varepsilon}) \quad \exists \varepsilon > 0$

2)  $\exists c > 0 \text{ s.t. } a \cdot f\left(\frac{n}{b}\right) < c \cdot f(n) \quad (c < 1)$

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

ALTRÒ MODO PER CALCOLARE I TEMPI DI RICORSIONE

→ ITERATIVO: itero fino ad arrivare ad una funzione in  $n$

$$\begin{cases} b \sim \Theta(1) & \text{caso base } n=1 \\ 8 + T(n-1) \end{cases}$$

$$8 + T(n-1) = 8 + [8 + T(n-2)] = 8n - 8 + T(1)$$

$$= 8 + 8 + [8 + T(n-3)] =$$

$$= \dots 8 \cdot k + T(n-k)$$

$$\approx k = n-1$$

$$8n - 8 + \Theta(1) \sim \Theta(n)$$

MERGE SORT → divide il vettore in due metà (divide), che a loro volta vengono divisi ricorsivamente (impara) fino a 1, ordina e poi COMBINA il risultato con una HERBE

↳ avanza una metà ordinandola, e la riportante fa semplicemente da riempimento

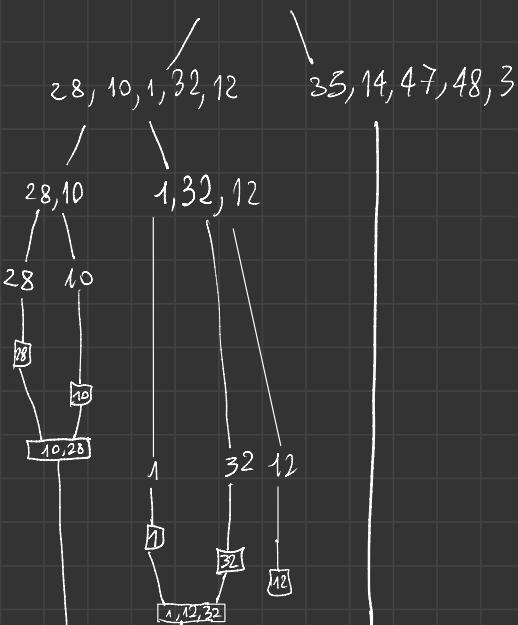
## TEMPI HERBESORT

$$T(m) = \begin{cases} \Theta(1) & m=1 \\ \Theta(1) + 2 T\left(\frac{m}{2}\right) + \Theta(m) & \text{ricorsione divisioni} \end{cases}$$

il tempo della HERBE

## SIMULAZIONE

28, 10, 1, 32, 12, 35, 14, 47, 48, 3

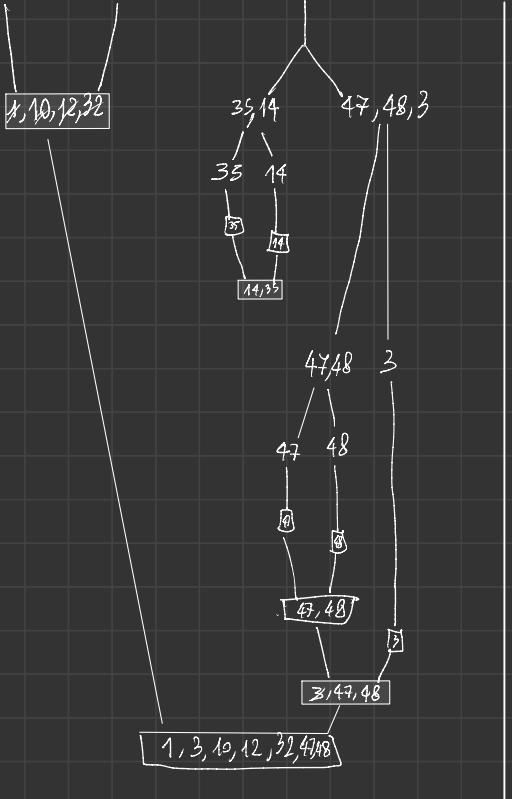


TEOREMA dell'esperto

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

$$m^{\log_b a} = m^{\log_2 2} = m^1 = m$$

CONDIZIONE 2 →  $T(m) = \Theta(m \log m)$



## QUICKSORT →

STABILE: NO

IN PLACE: SÌ

(consuma poco spazio)

## PARTITION

→ Scatto  $sx$  finché  $A[sx] > pivot$

→ decremento  $dx$  finché  $A[dx] \leq pivot$

} se si fermano e hanno i valori  $sx < dx$  (non ai posti corrispondenti) allora scambia  $A[sx]$  con  $A[dx]$

di solito il primo elemento

trovo un pivot per l'array e da quello

divido l'array in 2 parti (non per forma uguali)

↳ logiche

i numeri minori a destra, gli altri a sx

ordino le due metà ricorsivamente



## SIMULAZIONE PARZIALE

pivot=13

13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21

$\sum_{i=1}^k$

1)  $sx = 1$

$\sum_{i=1}^k$

2)  $dx = 2$

3)  $dx = 2$ ,  $A[1] \leftrightarrow A[11]$  scambio

4)  $dx = 3$

5)  $sx = 2$

6)  $dx = 3$ ,  $A[2] \leftrightarrow A[10]$

⋮

QUICKSORT con gli array ordinati funziona male



Soluzione: scegli un pivot random  
(random e poi lo scambio in prima posizione)

$$T_M = \Theta(m \log m)$$

T Empi

Partition:  $T(m) = \Theta(m)$

QUICKSORT:

$$T_{QS}(m) = \begin{cases} \Theta(1) & m=1 \\ \Theta(m) + T(Q) + T(M-Q) & \text{altra metà} \end{cases}$$

Due SITUAZIONI

1) Se divide in due metà precise (pivot a metà)

$$T(m) = \Theta(m) + T\left(\frac{M}{2}\right) + T\left(m - \frac{M}{2}\right) = \Theta(m) + T(m) + T\left(\frac{m}{2}\right) = \Theta(m) + 2T\left(\frac{m}{2}\right)$$

$$f(n) = \Theta(n)$$

$$n^{\log_b a} = n^{\frac{\log_a n}{\log_a b}} = n^{\frac{1}{\log_a b}}$$

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

2) Se una parte ha un solo elemento

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

Tramite albero di ricorsione  $T(n) = \Theta(n^2)$

COUNTING SORT → senza confronti/controlli



INPLACE: NO  
STABILE: sì

COME FUNZIONA

① AZZERO  $C[]$   $\Theta(k)$

② conta quante volte compare ogni elemento di  $A$ , memorizzata in  $C[]$   
 $\Theta(n)$   $\hookrightarrow$  alla posizione corrispondente  
al numero elementi

③ somma gli elementi da sinistra a destra in  $C$   
 $(da 2 a n) C[3]=C[2]+C[3] \dots$

$\Theta(k)$

④ quelle sono le posizioni dove ogni elemento

$T(m, k) = \Theta(m, k)$  e con  $k = \Theta(m)$  finirà in  $B$ . scorro  $A$  dalla fine all'inizio  
allora  $T(m) = \Theta(m)$  (per tenere stabile) e inserisco il valore in  $B$ ,  
decremento il valore di  $C[A[j]]$ .  
 $\downarrow$   
ORDINAMENTO CONCLUSO  $\Theta(m)$

## STRUTTURE DATI

Operazioni chiave:

- search → Maximum / Minimum
- Insert → Predecessor / Successor
- Delete

Array

Array ordinato

S	$O(n)$	$O(\log n)$
I	$O(1)$	$O(n)$
D	$O(1)$	$O(n)$
M/M	$O(m)$	$O(1)$
P/S	$O(m)$	$O(1)$

→ LISTE di due tipi:

① LISTE SEMPLICI: ogni nodo è  , next indica il prossimo nodo della lista

② LISTE doppie: ogni nodo è  , prev punta al precedente

In una lista il primo elemento è head [L], talvolta può esserci anche la tail [L]  
tail [L]   
↑ ultimo elemento

ESISTE UNA LISTA DETTA CIRCOLARE

→ lista che si divide



head[L].prev = tail e tail[L].next = head [L] → liste doppie circolari

## OPERAZIONI

→ search

pointer ListSearch (L, K)  
x = head(L)

$$T(n) = O(n)$$

while x.next ≠ null and x.key ≠ K

x = x.next

return x

→ INSERT ↗ inserimento in testa  
LISTINSERT(L, K)

K.next = head[L]  
head[L] = K

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

→ DELETE (doppia)

LISTDELETE(L, x)  $T(n) = \Theta(1)$

if  $x.\text{prev} \neq \text{null}$

$(x.\text{prev}).\text{next} = x.\text{next}$

else

$\text{head}[L] = x.\text{next}$

a) Se x non è la head, la prev del modo prima punta ad  $x.\text{next}$

b) x è la head, la nuova head

diventà  $x.\text{next}$

→ DELETE(SINGOLA)

LISTDELETE(L, x)

if  $\text{head}[L] = x$

$\text{head}[L] = x.\text{next}$

$$T_m(n) = \Theta(1)$$

$$T_p(n) = \Theta(n)$$

else

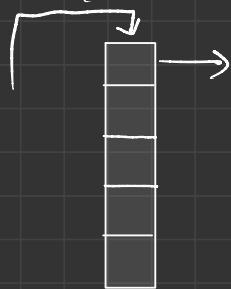
$y = \text{head}[L]$

while  $y.\text{next} \neq x$

$y = y.\text{next}$

$y.\text{next} = x.\text{next}$

STACK / PILE → politica LIFO (LAST IN, FIRST OUT)



→ Push ( $S, x$ ) → mette elemento in cima

→ Pop ( $S$ ) → estrae elemento dalla cima

→ stackEmpty ( $S$ ) → dice se lo stack è vuoto

$$T(m) = \Theta(1)$$

→ Top ( $S$ ) → restituisce la cima senza rimuoverla

# QUEUE / CODA $\rightarrow$ FIFO (FIRST IN, FIRST OUT)



$\rightarrow$  Enqueue ( $Q, x$ )  $\rightarrow$  aggiunge in posizione head

$\rightarrow$  Dequeue ( $Q$ )  $\rightarrow$  rimuove da posizione tail

$\rightarrow$  QueueEmpty ( $Q$ )  $\rightarrow$  dice se una coda è vuota

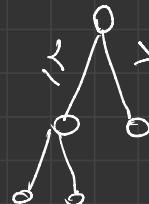
$\rightarrow$  head  $\rightarrow$  equivalente della Top

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

## ALBERO BINARIO DI RICERCA

Proprietà alberi:

① Root - radice in cima



② grado ( $x$ ) = quanti figli ha  $x$

③ profondità ( $x$ ) = livello di dove si trova il nodo

④ altezza = massima profondità possibile

parent	left child	key	right child

Root [ $T$ ] ha parent = null

Foglia ha l ed r = null

## Visite all'albero

- 1) Pre-order: stampo la radice, poi sottoalbero sx e poi dx
- 2) In-order: stampo il sottoalbero sx, radice e poi dx
- 3) Post-order: stampo il sottoalbero sx, poi dx e poi radice

Pre-order-visit( $x$ )

if  $x \neq \text{null}$

- ① → print  $x.\text{Key}$
- ② → PreorderVisit( $x.\text{sx}$ )
- ③ → PreOrderVisit( $x.\text{dx}$ )

## RICERCA nell'ALBERO

SBT\_Search( $x, K$ )

if  $x.\text{Key} == K$   
return  $x$   
else

if  $x.\text{Key} > K$

return SBT( $x.\text{right}, K$ )

*albero albero*

$$T(n) = O(h)$$

$h = O(n)$  oppure  $h = O(\log n)$   
*è ben bilanciato*

else

return SBT( x.left, K )

## MIN/MAX

MIN-SBT(T)

← va fino alla foglia più a sinistra

x = Root(T)

while x.left ≠ null

T(n) = O(h)

x = x.left

return x

MAX è la stessa cosa, ma con dx

SUCCESSOR → cerca giù in dx, se è null allora su

SBT-SUCCESSOR(x)

if right(x) ≠ null

return SBT-MIN(x.right)

else

y = x.parent

T(n) = O(h)

while y ≠ null AND x = y.dx

$x = y$   
 $y = y.\text{parent}$

return y

INSERT

→ scende fino alla prima foglia

SBT\_INSERT ( $T, z$ )

compatibile. Se il nodo prima

è null allora l'albero è vuoto  
e metto  $z$  in  $\text{root}$ , altrimenti

controlla se metterlo de  
foglia o sinistra

while  $x \neq \text{null}$

$y = x$

if  $z.\text{key} < x.\text{key}$

$x = x.\text{left}$   
else

$x = x.\text{right}$

$z.\text{parent} = y$

$T(n) = O(h)$

if  $y = \text{null}$

$h = n \rightarrow h = \log n$

$T.\text{root} = z$

else

if  $z.\text{key} < y.\text{key}$

$y.\text{left} = z$

else

$y.\text{right} = z$

$\rightarrow SBT \rightarrow \text{DELETE } (x, T) \longrightarrow 3 \text{ CASI}$

if  $x.\text{right} == x.\text{left} == \text{null}$

if  $x.\text{parent} == \text{null}$

Root [T] = null

else

if  $x == x.\text{parent}.sx$

$x.\text{parent}.left == \text{null}$

else

$x.\text{parent}.right == \text{null}$

else

if  $x.\text{left} == \text{null}$

combinazione  $(T, x, x.\text{right})$

else

if  $x.\text{right} == \text{null}$

combinazione  $(T, x, x.\text{left})$

else

1) eliminiamo una foglia

2) eliminiamo un padre con un solo figlio: cerco il successore nel ramo del figlio

3) eliminiamo un padre con due modi figli:

$$T(n) = O(h)$$

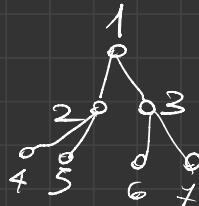
$y = \text{SBT\_SUCCESSOR}(x.\text{right})$

scambia ( $x, y$ )

$\text{SBT\_Delete}(T, y)$

HEAP  $\rightarrow$  array visto come albero

numero di foglie:  $\lfloor \frac{m}{2} \rfloor$



$$\text{FiglioLeft} = 2^*i$$

$$\text{FiglioRight} = (2^*i) + 1$$

$$\text{parent}(i) = \lfloor \frac{i}{2} \rfloor$$

Heapify  $\rightarrow$  Scambia figlio più grande del padre col padre e poi lancia la heapify su quel nodo scambiato

Buildheap  $\rightarrow$  costruisce lo heap dall'array

(passa heapinse e poi ciclo di heapify da  $\frac{n}{2}$ )

Heapsort  $\rightarrow$  buildheap + estrae l'ultima foglia, scambia con  
radice, lancia heapify su radice e stacca  
la foglia (ripete fino ad ordinare)

## GRAFI