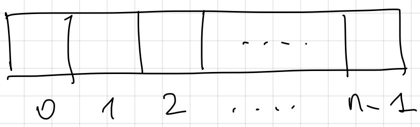
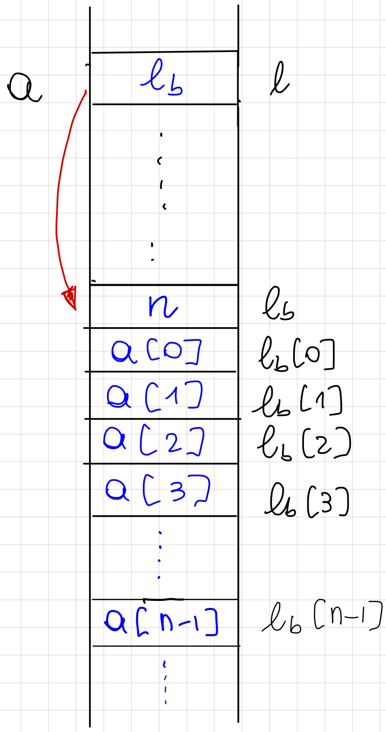
Array

MEMORIA

array a



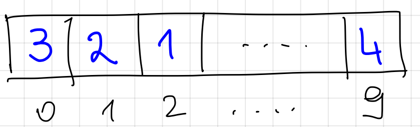


n = a.length

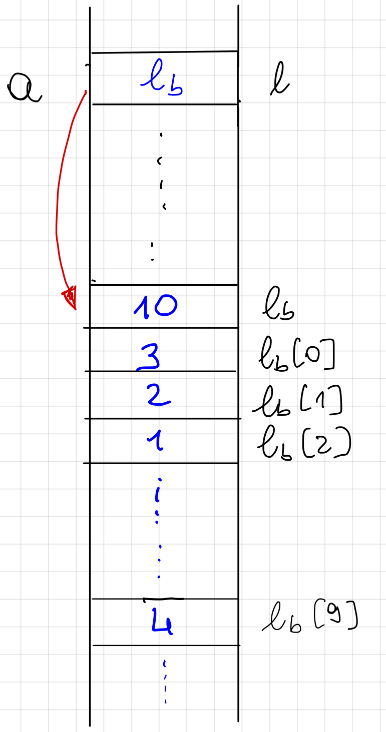
Array

MEMORIA

array a



a.length = 10

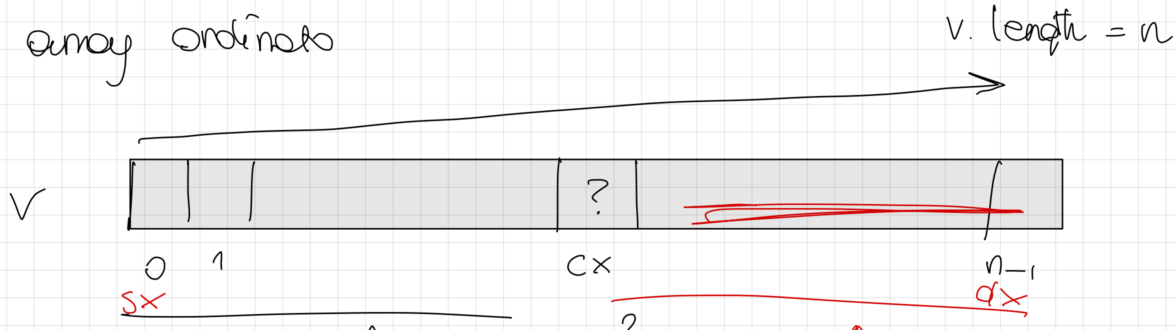


n = a.length

n = 10

RICERCA BINARIA

array ordinato



$k \in v$?

$v(cx) \stackrel{?}{==} k$

$v(cx) < k$

$v(cx) > k$

$sx = 0$

$dx = n-1$

Ricerca Binaria (v, k)

// input: array v, di n = v.length interi
intero k

$\dim I = |I| = \sim n$

```
n = v.length;
sx = 0;
dx = n-1;
pos = -1;
```
tempo costante

```
while ( sx ≤ dx && pos == -1) {
      cx = (sx + dx) / 2          // divisione intera
      if ( v(cx) == k )   pos = cx;
      else if ( v(cx) < k )   sx = cx + 1;
      else  dx = cx - 1;   // v(cx)] > k
}
print pos;
```

il while si esegue al più $\sim \log_2 n$ volte

Corpo while
tempo costante

V = 14 43 76 100 115 290 500 511

14	43	76	100	115	290	500	511
0	1	2	3	4	5	6	7

n = 8 k = 76

sx	dx	cx	confronto	pos
0	7	3	100 ÷ 76	−1
0	2	1	43 ÷ 76	
2	2	2	76 ÷ 76	2

k = 300

sx	dx	cx	confronto	pos
0	7	3	100 ÷ 300	−1
4	7	5	290 ÷ 300	
6	7	6	500 ÷ 300	
6	5			

300 non è presente

n elementi

↓ 1° confronto

$n/2$

↓ 2° confronto

$n/4$

⋮ i confronti

$n/2^i$

↳ ampiezza della porzione di array in cui si cerca la chiave

$$\frac{n}{2^i} = 1$$

$$n = 2^i \implies \log_2 n = \log_2 2^{i}$$

$$i = \log_2 n$$

\# Confronti al caso peggiore è $\sim \log_2 n$

↳ Ricerca senza successo

↳ $T_{peggiore}(n)$ cresce come $\log_2 n$

(a meno di costante)

$$T_{peggiore}(n) \leq cost \cdot \log_2 n$$

IL PROBLEMA DELL'ORDINAMENTO

Input : array a di n interi

Output array a ordinato in ordine non decrescente :

$$a[0] \leq a[1] \leq a[2] \leq \dots \leq a[n-1]$$

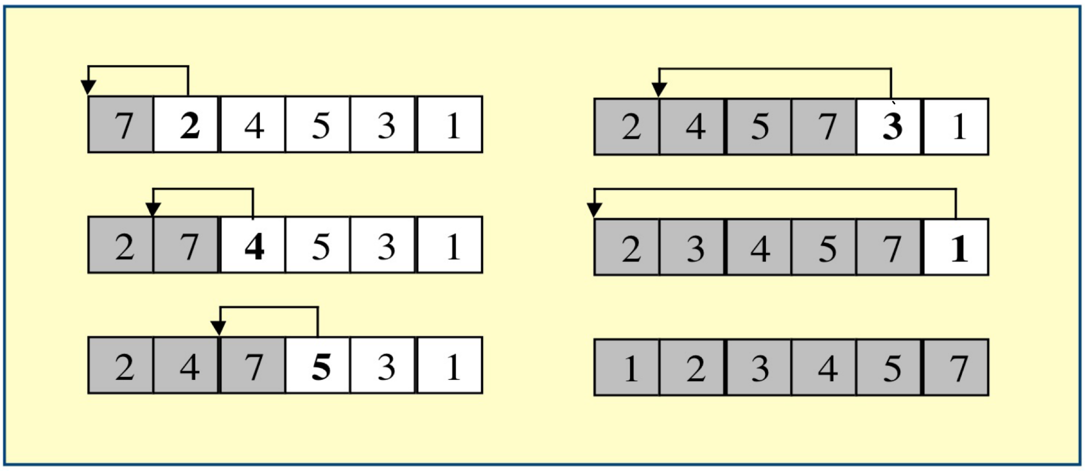
Istanza di input: a

dim istanza di input : n

$n = a.length$

Word - model

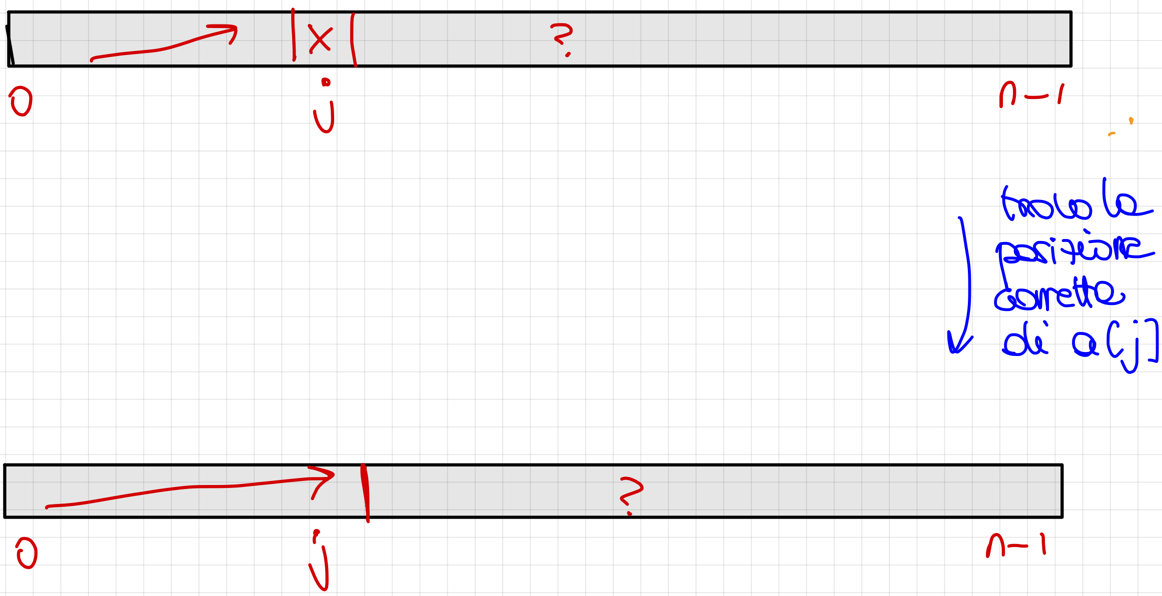
INSERTION SORT



Algoritmo INCREMENTALE: procede per fasi

$n-1$ fasi $\qquad j = 1, \dots, n-1$

Fase j: l'elemento di indice j di a viene inserito al posto corretto tra gli elementi che lo precedono



Insertion Sort (a)   // input: array a di n interi

```
for ( j = 1 ; j < n ; j++) {        → si ripete n-1 volte
    key = a[j] ;
    i = j - 1;
    while ( i ≥ 0 && a[i] > key ) {
        a[i+1] = a[i];
        i--;
    }
    a[i+1] = key;
}
```

inserire a[j] nella sequenza ordinata a[0...j-1]

spostamento degli elementi più grandi di key (= a[j]) di una posizione verso destra per far posto a key

- → si confronta key (= a[j]) con gli elementi precedenti, per trovare la posizione in cui inserirlo
- → procedendo da j-1 verso l'inizio dell'array si spostano gli elementi di una posizione verso destra per fare posto a quello da inserire.
- → si sfrutta il fatto che questi elementi sono ordinati

CORRETTEZZA con INVARIANTE di ciclo

proprietà mantenuta durante l'esecuzione dell'algoritmo:

INVARIANTE

all'inizio di ogni iterazione del ciclo for il sottoarray a[0...j-1] è ORDINATO e contiene gli stessi elementi che erano originariamente in a[0...j-1]

① Inizializzazione

vale prima della prima iterazione del for

$j = 1$ $a[0 \dots 1-1] = \underline{a[0]}$ ✓

② Conservazione

se vale prima di un'iterazione del ciclo, allora rimane valida prima della iterazione successiva

↳ ok, per ispezione diretta del codice: infatti

- $a[j]$ viene inserito nella posizione corretta in $a[0 \dots j]$
- $a[0 \dots j]$ risulta quindi ordinato e contiene i primi $j+1$ elementi dell'array a, quelli che erano originariamente in quella porzione di array.

③ Conclusione

valutazione dell'invariante a fine ciclo

↳ correttezza dell'algoritmo

quando termina il ciclo for $j = n$

↳ Invariante:

$a[0 \dots j-1] = a[0 \dots n-1]$ (intero array)

è ordinato e contiene gli elementi che si trovavano originariamente in a.

$\square$

Analisi (conteggio dei confronti)

$n-1$ iterazioni del for

il costo di ogni iterazione del for dipende dalle ripetizioni del while (della valutazione della guardia)

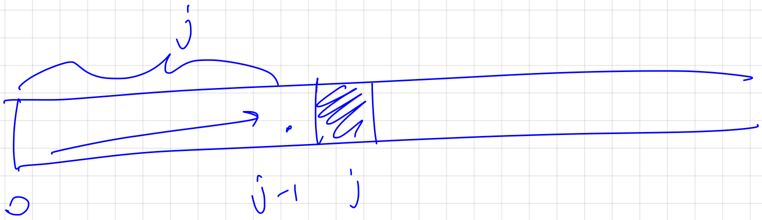
↑ confronto

$$1 \leq \#\text{ confronti} \leq j$$

guardia subito falsa (caso ottimo) — $a[i] \leq k$ — Array già ordinato

caso pessimo — guardia vera sempre — $a(i) > k$ — Array ordinato in senso inverso

1	2	3	4	5	
5	4	3	2	1	1 c.
4	5	3	2	1	2 c.
3	4	5	2	1	3



CASO OTTIMO

1 confronto $\forall$ iterazione del for
(guardia while sempre false)

$$C_{ottimo}(n) = n-1$$

$\Rightarrow$ costo in tempo
è proporzionale al # di confronti

$T_{ottimo}(n)$ cresce come n, cioè
come una funzione lineare
nella dimensione dell'array

CASO PESSIMO

$\forall j$, $1 \le j \le n-1$ j confronti (la guardia del while è sempre vera)

$$C_p(n) = \sum_{j=1}^{n-1} j = \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2} = \binom{n}{2}$$

$\rightarrow$ $C_p(n)$ cresce come una funzione quadratica di n

$\rightarrow$ $T_p(n)$ cresce come $C_p(n)$, a meno di costanti

$\Rightarrow$ costo in tempo quadratico al caso pessimo

$\binom{n}{2}$ = # coppie di 2 elementi presi da un insieme di n elementi

$\Rightarrow$ fa tutti i confronti possibili.
è quanto di peggio si possa fare

CASO MEDIO

possiamo aspettarci che al passo j, quando confrontiamo k con gli elementi che lo precedono, metà siano $\geq$ e metà $<$.

quindi: # iterazioni del while $\sim j/2$

confronti $\sim j/2$

$$C_{medio}(n) = \sum_{j=1}^{n-1} j/2 = \underbrace{\frac{1}{2}\binom{n}{2}}_{\text{cresce ancora come } n^2} = \frac{n(n-1)}{4}$$

$T_{medio}(n)$ cresce come n^2