Algoritmi e Strutture Dati

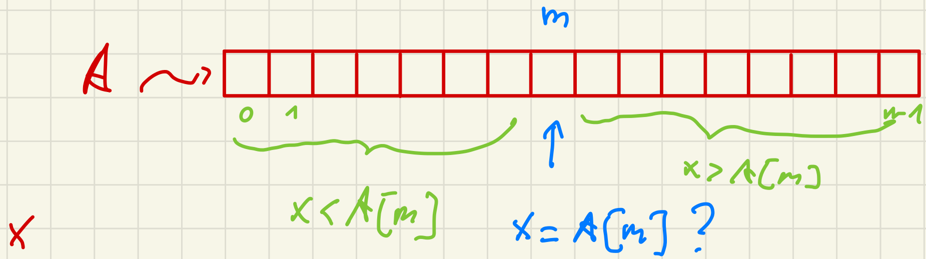
Lezione 7

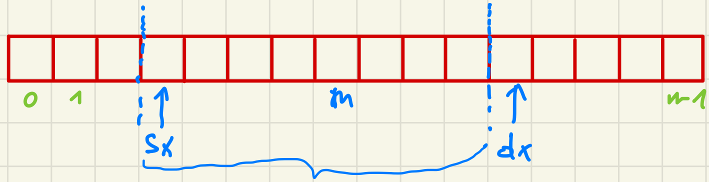
12 ottobre 2022

Ricerca in un array

input	Array A, elemento x
output	Indice i t.c. $A[i] = x$ -1, se A non contiene x

RICERCA IN ARRAY ORDINATO






```
FUNZIONE ricercaRic(Array A, indice sx, indice dx, elemento x)
                                                    → indice
   IF dx ≤ sx THEN RETURN -1
   ELSE
      m ← (sx+dx)/2        // div intera
      IF x = A[m] THEN RETURN m
      ELSE IF x < A[m] THEN
                 RETURN ricercaRic(A, sx, m, x)
      ELSE
                 RETURN ricercaRic(A, m+1, dx, x)


ALGORITMO ricercaBinaria (Array A[0..n-1],
                          elemento x) → indice
   RETURN ricercaRic(A, 0, n, x)
```

```
FUNZIONE ricercaRic(Array A, indice sx, indice dx, elemento x)
                                          → indice
  IF dx ≤ sx THEN RETURN -1
  ELSE
     m ← (sx+dx)/2          // div intera
     IF x = A[m] THEN RETURN m
     ELSE IF x < A[m] THEN
              RETURN ricercaRic(A, sx, m, x)
  ELSE
              RETURN ricercaRic(A, m+1, dx, x)
```

Record di attivazione

A
sx
dx
x
m

A →	1	5	7	12	16	18	20	22
	0	1	2	3	4	5	6	7

x 12

ricercaRic(A, 0, 8, 12)

3

sx	0
dx	8
m	4

3

ricercaRic(A, 0, 4, 12)

sx	0
dx	4
m	2

ricercaRic(A, 3, 4, 12)

3

Sx	3
dx	4
m	3

x = 11 ?

FUNZIONE ricercaRic(Array A, indice sx, indice dx, elemento x) → indice

```
IF dx ≤ sx THEN RETURN -1
ELSE
    m ← (sx+dx)/2   // div intera
    IF x = A[m] THEN RETURN m
    ELSE IF x < A[m] THEN
            RETURN ricercaRic(A, sx, m, x)
    ELSE
            RETURN ricercaRic(A, m+1, dx, x)
```

ALGORITMO ricercaBinaria(Array A[0..n-1], elemento x) → indice

```
RETURN ricercaRic(A, 0, n, x)
```

lunghezza spazio ricerca ($dx - sx$)

1^a chiamata $dx = n$ $sx = 0$ — n

2^a " — $\leq \frac{n}{2}$

3^a " — $\leq \frac{n}{2^2}$

,

i-esima chiamata — $\leq \frac{n}{2^{i-1}}$

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

$$i = 1 + \log_2 n$$

per arrivare a $dx \leq sx$ (caso base) una ulteriore chiamata

totale chiamate ricorsive è al più $2 + \log_2 n$

```
FUNZIONE ricercaRic(Array A, indice sx, indice dx, elemento x)
                                        → indice
  IF dx ≤ sx THEN RETURN -1
  ELSE
     m ← (sx+dx)/2        // div intera
     IF x = A[m] THEN RETURN m
     ELSE IF x < A[m] THEN
            RETURN ricercaRic(A, sx, m, x)
     ELSE
            RETURN ricercaRic(A, m+1, dx, x)
```

```
ALGORITMO ricercaBinaria(Array A[0..n-1],
                  elemento x) → indice
   RETURN ricercaRic(A, 0, n, x)
```

$\leq 2 + \log_2 n$ chiamate ricorsive in tutto

Ipotesi: costo confronti $\Theta(1)$

$\Rightarrow$ tempo $\Theta(\log n)$

Spazio: $\Theta(\log n)$ al più record di attivazione nello stack

spazio di un record $O(1)$

$\Longrightarrow$ Spazio $\Theta(\log n)$

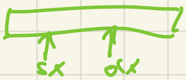
```
FUNZIONE ricercaRic(Array A, indice sx, indice dx, elemento x)
        → indice

  IF dx ≤ sx THEN RETURN -1
  ELSE
     m ← (sx+dx)/2        // div intera
     IF x = A[m] THEN RETURN m
     ELSE IF x < A[m] THEN
              RETURN ricercaRic(A, sx, m, x)
     ELSE
              RETURN ricercaRic(A, m+1, dx, x)
```

```
ALGORITMO ricercaBinaria(Array A[0..n-1], elemento x) → indice

  RETURN ricercaRic(A, 0, n, x)
```

```
ALGORITMO ricercaBinaria(Array A[0..n-1], elemento x)
                                         → indice
   sx ← 0
   dx ← n
   pos ← -1
   WHILE sx < dx AND pos = -1 DO
      m ← (sx+dx)/2
      IF x = A[m] THEN pos ← m
      ELSE IF x < A[m] THEN dx ← m
      ELSE sx ← m+1
   RETURN pos
```



Spazio $O(1)$

Tempo $O(\log n)$

A							
1	5	7	12	16	18	20	22
0	1	2	3	4	5	6	7

m sx dx

x = 11

pos -1

Algoritmi di ordinamento (sort)

Il problema dell'ordinamento

Tecniche differenti:

- Ordinamento interno
  Dati da ordinare in memoria centrale

  ⇒ Accesso diretto agli elementi → tipicamente ARRAY

- Ordinamento esterno
  Dati da ordinare in memoria di massa

  ⇒ Accesso a blocchi di dati
  Lentezza hardware periferiche

Ordinamento interno

Cosa ordiniamo?

- Vettori (array) di strutture complesse, come oggetti o record (`struct` in Go e in C)
- Un particolare *campo* è scelto come *chiave* per l'ordinamento
- *Per semplicità* negli esempi ordiniamo array di interi

libro
- Autori
- titolo
- editore
- anno
- ⋮

persona
- Nome
- Cognome
- Data di nascita
- Codice Fiscale
- ⋮

Stabilità

Un algoritmo di ordinamento è detto *stabile* se preserva l'ordine relativo tra record con la medesima chiave

chiave → 15 gatto 10 tigre 15 cane ...

$\Rightarrow$ 10 tigre ... 15 gatto 15 cane ...

Complessità degli algoritmi di ordinamento

- Studieremo principalmente algoritmi di ordinamento basati su *confronti tra chiavi*

Spazio: Considereremo la memoria utilizzata in aggiunta all'array da ordinare (inclusa la memoria sullo stack nel caso di algoritmi ricorsivi)

Tempo: Stimeremo la complessità in *tempo* di questi algoritmi, in funzione della lunghezza del vettore da ordinare, calcolando prima di tutto il *numero di confronti tra chiavi*

Complessità degli algoritmi di ordinamento

Perché il numero di confronti?

- Operazioni “più costose” utilizzate da questi algoritmi
- Se ciascun confronto viene effettuato in *tempo costante* il numero di confronti fornisce una stima del tempo di calcolo
- *In generale* il tempo di calcolo può essere stimato *moltiplicando* il numero di confronti per il tempo utilizzato da ciascun confronto

Alcune tecniche di ordinamento interno

basate su confronti

Algoritmi elementari

- per selezione `selectionSort`
- per inserimento `insertionSort`
- "a bolle" $\Theta(n^2)$ confronti `bubbleSort`

Alcune tecniche di ordinamento interno

basate su confronti

Algoritmi elementari

- per selezione `selectionSort`
- per inserimento `insertionSort`
- "a bolle" `bubbleSort`

Algoritmi avanzati $\Theta(n \lg n)$

- per fusione `mergeSort`
- veloce `quickSort` $\Theta(n^2)$
- basato su "heap" `heapSort`

Alcune tecniche di ordinamento interno

basate su confronti

Algoritmi elementari

- per selezione `selectionSort`
- per inserimento `insertionSort`
- "a bolle" `bubbleSort`

Algoritmi avanzati

- per fusione `mergeSort`
- veloce `quickSort`
- basato su "heap" `heapSort`

Sono tutti *in loco*, cioè non necessitano di strutture ausiliarie, eccetto `mergeSort` e, per certi aspetti, `quickSort`

Non tutti sono ~~sono~~ stabili

Selection Sort

Array $A[0..n-1]$

$0 \quad \dots \quad k-1 \;\vert\; k \quad k+1 \quad \dots \quad n-1$

ordinati / posizione definitiva

$A[0] \leq A[1] \leq \dots \leq A[k-1]$

da ordinare
più grandi di quelli della parte SX

$A[k-1] \leq A[j]$
per ogni $j \geq k$

PASSO K
$(k = 0 .. n-2)$

$0 \quad \dots \quad k-1 \quad k \;\vert\; k+1 \quad \dots \quad n-1$

ordinati / posizione definitiva

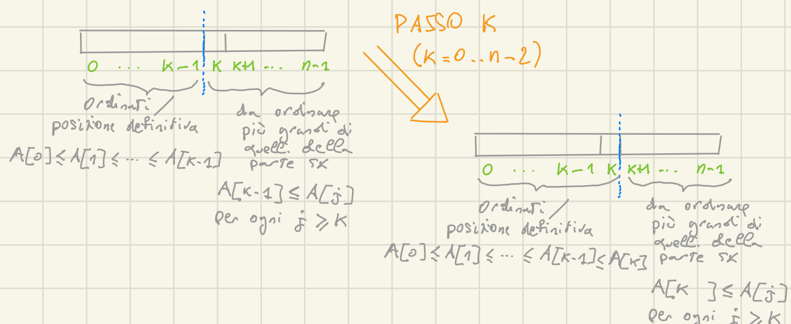
$A[0] \leq A[1] \leq \dots \leq A[k-1] \leq A[k]$

da ordinare
più grandi di quelli della parte SX

$A[k] \leq A[j]$
per ogni $j \geq k$

Selection Sort

```
ALGORITMO selectionSort (array A[0..n-1])
    FOR k ← 0 TO n-2 DO
        // ricerca la posizione del minimo in A[k..n-1]
        m ← k
        FOR j ← k+1 TO n-1 DO
            IF A[j] < A[m] THEN
                m ← j
        scambia A[m] con A[k]
```



SelectionSort: n° di confronti

```
ALGORITMO selectionSort (array A[0..n-1])
  FOR k ← 0 TO n-2 DO
  | // ricerca la posizione del minimo in A[k..n-1]
  | m ← k
  | FOR j ← k+1 TO n-1 DO
  | | IF A[j] < A[m] THEN
  | |_    m ← j
  |_scambia A[m] con A[k]
```

$n-k-1$ confronti

\# tot. confronti:

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

$n-1$
$n-2$
$\vdots$
1

spazio $O(1)$