

Ordinamento, algoritmi quadratici

Algoritmi e strutture dati

Ugo de'Liguoro, Andras Horvath

1

Sommario

- Obiettivi
 - sviluppo di algoritmi di ordinamento di complessità quadratica, verifica della correttezza con invariante e introduzione dell'analisi di complessità
- Argomenti
 - problema dell'ordinamento (sorting)
 - insertion-sort
 - selection-sort

2

Ricerca in vettore non ordinato

- come si cerca un elemento in un vettore non ordinato?
- dobbiamo esaminare il vettore elemento per elemento
- quanti confronti servono per cercare un elemento in un vettore di n elementi nel caso peggiore e nel caso migliore?
- n nel caso peggiore (elemento non c'è)
- 1 nel caso migliore (il primo elemento esaminato è quello cercato)
- come si procede nel caso in cui il vettore è ordinato?

3

Ricerca binaria (dicotomica)

- algoritmo di ricerca per cercare elementi in vettori ordinati
- l'idea: confrontiamo l'elemento centrale e quello ricercato
 - se sono uguali, allora l'elemento è presente
 - se l'elemento ricercato è più grande, bisogna cercare nella prima metà del vettore
 - se l'elemento ricercato è più piccolo, bisogna cercare nella seconda metà del vettore
- iterando l'idea, in ogni giro o si trova l'elemento o si dimezza la dimensione del problema
- quando la porzione ancora "valida" del vettore contiene un elemento solo, è facile decidere se l'elemento c'è o meno

4

Ricerca binaria (dicotomica)

```

BINSEARCH-RIC( $x, A, i, j$ )
  ▷ Pre:  $A[i..j]$  ordinato
  ▷ Post:  $true$  se  $x \in A[i..j]$ 
  if  $i > j$  then
    ▷  $A[i..j] = \emptyset$ 
    return false
  else
     $m \leftarrow \lfloor (i + j) / 2 \rfloor$ 
    if  $x = A[m]$  then
      return true
    else
      if  $x < A[m]$  then
        return BINSEARCH-RIC( $x, A, i, m - 1$ )
      else
        ▷  $A[m] < x$ 
        return BINSEARCH-RIC( $x, A, m + 1, j$ )
      end if
    end if
  end if
end if

```

5

Ricerca in vettore ordinato

- quanti confronti servono con la ricerca binaria nel caso peggiore e nel caso migliore?
- 1 nel caso migliore (il primo elemento esaminato è quello cercato)
- nel caso peggiore (elemento ricercato non c'è) si dimezza il problema in ogni giro: numero di confronti è all'incirca $\log_2 n$ (possiamo dimezzare il vettore $\log_2 n$ volte senza svuotarlo)

Conviene ordinare se
bisogna fare tante
ricerche.

6

Problema dell'ordinamento

Ordinamento come problema computazionale:

Input: una sequenza di n numeri a_1, a_2, \dots, a_n

Output: una permutazione $a_{i_1}, a_{i_2}, \dots, a_{i_n}$ della sequenza in ingresso tale che $a_{i_1} \leq a_{i_2} \leq \dots \leq a_{i_n}$

7

Forza bruta

```

SORTED(A)
for i ← 2 to length(A) do
  if A[i-1] > A[i] then
    return false
  end if
end for
return true

```

```

TRIVIAL-SORT(A)
for all A' permutazione di A do
  if SORTED(A') then
    return A'
  end if
end for

```

Il numero di permutazioni di un vettore di n elementi distinti sono $n!$

8

Crescita di 2^n ed $n!$

n	2^n	$n!$
0	1	1
1	2	1
2	4	2
3	8	6
4	16	24
5	32	120
6	64	720
7	128	5040
8	256	40320
9	512	362880
...
18	262144	6402373705728000
19	524288	121645100408832000
...

9

Ordinamento per inserimento

- l'idea per ordinare il vettore $A[1..n]$:
 - quando la parte $A[1..i-1]$ è già ordinato
 - si può **inserire** l'elemento $A[i]$ nella parte ordinata tramite scambi:
 - se $A[i] \geq A[i-1]$ allora $A[1..i]$ è ordinato e ci si ferma, altrimenti si scambia $A[i]$ con $A[i-1]$
 - se $A[i-1] \geq A[i-2]$ allora $A[1..i]$ è ordinato e ci si ferma, altrimenti si scambia $A[i-1]$ con $A[i-2]$
 - se $A[i-2] \geq A[i-3]$ allora $A[1..i]$ è ordinato e ci si ferma, altrimenti si scambia $A[i-2]$ con $A[i-3]$
 - ...
 - dopo gli scambi $A[1..i]$ è ordinato
 - per partire abbiamo che $A[1..1]$ è ordinato
 - si inserisce nella parte ordinata prima $A[2]$, poi $A[3]$, ... e infine $A[n]$

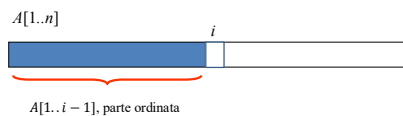
10

Simulazione

- simuliamo l'idea con $A = (5, 4, 7, 3, 6, 6)$
 - (4, 5, 7, 3, 6, 6)
 - (4, 5, 3, 7, 6, 6)
 - (4, 3, 5, 7, 6, 6)
 - (3, 4, 5, 7, 6, 6)
 - (3, 4, 5, 6, 7, 6)
 - (3, 4, 5, 6, 6, 7)

11

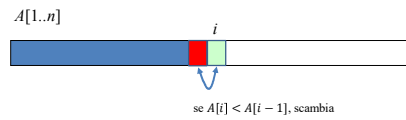
Ordinamento per inserimento



12

Ordinamento per inserimento

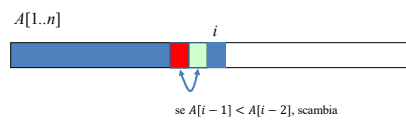
$A[i] < A[i-1] ?$



13

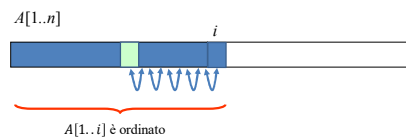
Ordinamento per inserimento

$A[i-1] < A[i-2] ?$



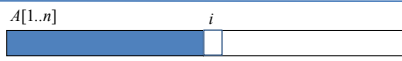
14

Ordinamento per inserimento



15

Ordinamento per inserimento



$A[1..i-1]$, parte ordinata

```

INSERTION-SORT(A)
  for  $i \leftarrow 2$  to  $\text{length}(A)$  do
    ▷ inserisce  $A[i]$  in  $A[1..i-1]$ 
     $j \leftarrow i$ 
    while  $j > 1$  and  $A[j-1] > A[j]$  do
      scambia  $A[j-1]$  con  $A[j]$ 
       $j \leftarrow j - 1$ 
    end while
  end for
  return  $A$ 

```

16

Ordinamento per inserimento

```

INSERTION-SORT(A)
  for  $i \leftarrow 2$  to  $\text{length}(A)$  do
    ▷ inserisce  $A[i]$  in  $A[1..i-1]$ 
     $j \leftarrow i$ 
    while  $j > 1$  and  $A[j-1] > A[j]$  do
      scambia  $A[j-1]$  con  $A[j]$ 
       $j \leftarrow j - 1$ 
    end while
  end for
  return  $A$ 

```

- Insertion-Sort(A) termina per ogni A ?
- La sequenza che restituisce è ordinata?
- Quanto tempo impiega in funzione di $n = \text{length}(A)$?

La **terminazione** è assicurata dal fatto che sia il **for** che il **while** sono cicli limitati.

17

Correttezza dell'algorithm

```

INSERTION-SORT(A)
  for  $i \leftarrow 2$  to  $\text{length}(A)$  do
    ▷ inserisce  $A[i]$  in  $A[1..i-1]$ 
     $j \leftarrow i$ 
    while  $j > 1$  and  $A[j-1] > A[j]$  do
      scambia  $A[j-1]$  con  $A[j]$ 
       $j \leftarrow j - 1$ 
    end while
  end for
  return  $A$ 

```



$A[1..i-1]$ è ordinato

Insertion-Sort è iterativo con due cicli, usiamo invarianti per la sua verifica.

L'invariante del ciclo esterno?

$A[1..i-1]$ è ordinato.

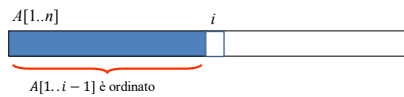
18

Correttezza dell'algoritmo

Dimostrazione dell'invariante esterno: $A[1..i-1]$ è ordinato:

- **inizializzazione:**

- prima di eseguire il ciclo per la prima volta $i = 2$
- con $i = 2$ l'invariante diventa $A[1..1]$ è ordinato e questo è vero



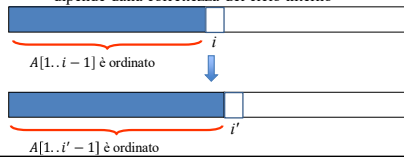
19

Correttezza dell'algoritmo

Dimostrazione dell'invariante esterno: $A[1..i-1]$ è ordinato:

- **mantenimento:**

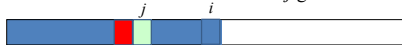
- dobbiamo dimostrare che " $A[1..i-1]$ ordinato $\Rightarrow A[1..i'-1]$ ordinato" dove $i' = i + 1$
- se $A[i]$ viene inserito correttamente in $A[1..i-1]$, allora l'invariante viene mantenuto
- dipende dalla correttezza del ciclo interno



20

Correttezza dell'algoritmo

- Invariante del ciclo interno?
- osserviamo la situazione con un j generico

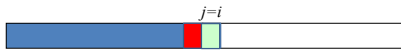


- verde: elemento che era nella posizione i prima di eseguire il ciclo interno
- abbiamo già eseguiti degli scambi
- rosso: elemento da confrontare con verde
- invariante:
 - $A[1..j-1]$ e $A[j..i]$ sono ordinati
 - ciascun elemento in $A[1..j-1]$ è minor uguale di tutti gli elementi di $A[j+1..i]$

21

Correttezza dell'algoritmo

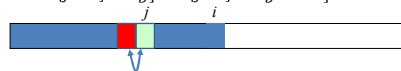
- **dimostriamo l'invariante:** $A[1..j-1]$ e $A[j..i]$ sono ordinati e ciascun elemento in $A[1..j-1]$ è minor uguale di tutti gli elementi di $A[j+1..i]$ (abbrev.: $A[1..j-1] \leq A[j+1..i]$)
- **inizializzazione:** con $j = i$ l'invariante diventa:
 - $A[1..i-1]$ e $A[i..i]$ (vettore di singolo elemento) sono ordinati
 - ciascun elemento in $A[1..i-1]$ è minor uguale di tutti gli elementi di $A[i+1..i] = \emptyset$
- quindi $j = i$ l'invariante si riduce a: $A[1..i-1]$ è ordinato
- e questo è garantito dal invariante esterno



22

Correttezza dell'algoritmo

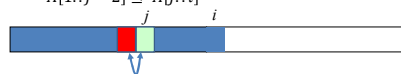
- **dimostriamo l'invariante:** $A[1..j-1]$ e $A[j..i]$ sono ordinati e $A[1..j-1] \leq A[j+1..i]$
- **mantenimento:** bisogna dimostrare che "se l'invariante vale prima allora vale anche dopo l'esecuzione del ciclo"
- il ciclo si esegue solo se $j > 1 \wedge A[j-1] > A[j]$
- se il ciclo si esegue allora si scambiano $A[j-1]$ e $A[j]$ e j viene decrementato ($j' = j - 1$), implicazioni:
 - $A[1..j-1]$ è ordinato $\Rightarrow A[1..j'-1] = A[1..j-2]$ è ordinato
 - $A[j..i]$ è ordinato $\wedge A[1..j-1] \leq A[j+1..i] \wedge A[j-1] > A[j] \Rightarrow A[j'..i] = A[j-1..i]$ è ordinato



23

Correttezza dell'algoritmo

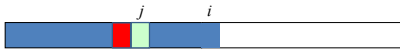
- **dimostriamo l'invariante:** $A[1..j-1]$ e $A[j..i]$ sono ordinati e $A[1..j-1] \leq A[j+1..i]$
- **mantenimento:** bisogna dimostrare che "se l'invariante vale prima allora vale anche dopo l'esecuzione del ciclo"
- il ciclo si esegue solo se $j > 1 \wedge A[j-1] > A[j]$
- se il ciclo si esegue allora si scambiano $A[j-1]$ e $A[j]$ e j viene decrementato ($j' = j - 1$), implicazioni:
 - $A[1..j-1] \leq A[j+1..i] \wedge A[j-1] > A[j] \Rightarrow A[1..j'-1] \leq A[j'+1..i]$ ovvero $A[1..j-1] \leq A[j+1..i] \wedge A[j-1] > A[j] \Rightarrow A[1..j-2] \leq A[j..i]$



24

Correttezza dell'algoritmo

- **invariante interno:** $A[1..j-1]$ e $A[j..i]$ sono ordinati e $A[1..j-1] \leq A[j+1..i]$
- all'uscita dal ciclo interno abbiamo $j = 1 \vee A[j-1] \leq A[j]$
- in ogni caso all'uscita l'invariante implica che $A[1..i]$ è ordinato
- quindi abbiamo dimostrato che
 se prima di eseguire il ciclo interno $A[1..i-1]$ è ordinato
 allora dopo l'esecuzione del ciclo interno $A[1..i]$ è ordinato



25

Correttezza dell'algoritmo

- **invariante esterno:** $A[1..i-1]$ è ordinato
- all'uscita dal ciclo interno abbiamo $i = n + 1$
- dunque all'uscita l'invariante implica che $A[1..n]$ è ordinato
- quindi abbiamo dimostrato che
 l'algoritmo è corretto

26

Il tempo di calcolo di Insert-Sort

Quanto tempo impiega?
 Dipende dalla dimensione dall'ingresso, $n = \text{length}(A)$.

27

Il tempo di calcolo di Insert-Sort

	costo	num. volte
1. for $i \leftarrow 2$ to $\text{length}(A)$	c_1	n
2. $j \leftarrow i$	c_2	$n - 1$
3. while $j > 1$ and $A[j-1] > A[j]$	c_3	$\sum_{i=2}^n t_i$
4. scambia $A[j-1]$ con $A[j]$	c_4	$\sum_{i=2}^n (t_i - 1)$
5. $j \leftarrow j - 1$	c_5	$\sum_{i=2}^n (t_i - 1)$

Riga 1: l'esecuzione prevede assegnare un valore alla variabile i (2 la prima volta e $i+1$ successivamente) e controllare se $i \leq \text{length}(A)$; il controllo viene eseguito con $i=2,3,\dots, \text{length}(A)+1$, quindi n volte

$t_i = n$. esecuzioni del test del **while** = $\begin{cases} 1 & \text{nel caso migliore} \\ i & \text{nel caso peggiore} \end{cases}$

28

Il tempo di calcolo di Insert-Sort

Con $t_i = i$, caso peggiore:

$$T_{\text{ins}}(n) = c_1 n + c_2 (n-1) + c_3 \sum_{i=2}^n i + c_4 \sum_{i=2}^n (i-1) + c_5 \sum_{i=2}^n (i-1) \\ = (c_1 + c_2)n - c_2 + c_3 \sum_{i=2}^n i + (c_4 + c_5) \sum_{i=2}^n (i-1)$$

$$\sum_{i=2}^n i = 2 + 3 + \dots + n = \frac{n+2}{2}(n-1) = \frac{n^2 + n - 2}{2}$$

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

$$T_{\text{ins}}(n) = \frac{c_3 + c_4 + c_5}{2} n^2 + \left(c_1 + c_2 + \frac{c_3 - c_4 - c_5}{2} \right) n - (c_2 + c_3) \\ = an^2 + bn + c$$

Nel caso peggiore Insert-Sort ha
complessità temporale quadratica.

29

Il tempo di calcolo di Insert-Sort

Con $t_i = 1$, caso migliore:

$$T_{\text{ins}}(n) = c_1 n + c_2 (n-1) + c_3 \sum_{i=2}^n 1 + c_4 \sum_{i=2}^n (1-1) + c_5 \sum_{i=2}^n (1-1) \\ = (c_1 + c_2)n - c_2 + c_3 \sum_{i=2}^n 1$$

$$\sum_{i=2}^n 1 = 1 + 1 + \dots + 1 = n - 1$$

$$T_{\text{ins}}(n) = (c_1 + c_2 + c_3)n - (c_2 + c_3) = dn + e$$

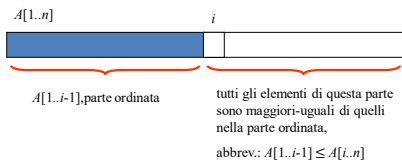
Nel caso migliore Insert-Sort ha
complessità temporale lineare.

30

Ordinamento per selezione

L'idea dell'algoritmo:

- assumiamo che la parte sinistra del vettore sia ordinata e quella a destra contenga elementi maggiori-uguali
- graficamente:



- cerchiamo l'elemento minimo in $A[i..n]$ e lo scambiamo con $A[i]$
- così la parte ordinata si allarga (la disordinata diminuisce)

31

Ordinamento per selezione

```

SELECT-SORT(A)
for  $i \leftarrow 1$  to  $\text{length}(A) - 1$  do    ▷  $n = \text{length}(A)$ 

     $k \leftarrow i$ 
    for  $j \leftarrow i + 1$  to  $\text{length}(A)$  do
        if  $A[k] > A[j]$  then
             $k \leftarrow j$ 
        end if
    end for
    scambia  $A[i]$  con  $A[k]$ 
end for
return A

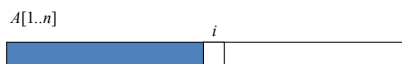
```

32

Ordinamento per selezione

Invariante del ciclo esterno:

- $A[1 \dots i-1]$ è ordinato
- se x è in $A[i \dots n]$ ed y è in $A[1 \dots i-1]$ allora $x \geq y$



Inizializzazione:

- con $i=1$ la porzione $A[1 \dots i-1]$ è vuota
- dunque la proposizione espressa dall'invariante non può che valere

33

Ordinamento per selezione

Mantenimento:

- dobbiamo dimostrare che "se vale prima allora vale anche dopo"
- come ipotesi possiamo assumere che prima di eseguire il corpo del ciclo interno
 - $A[1..i-1]$ è ordinato e
 - se x è in $A[i..n]$ ed y è in $A[1..i-1]$ allora $x \geq y$
- assumiamo che la ricerca del minimo in $A[i..n]$ sia eseguita correttamente e di conseguenza $A[k]$ sia il minimo valore in $A[i..n]$ (dimostreremo dopo che è corretto)
- $A[k]$ è minimo in $A[i..n]$ ma è maggior-uguale di qualunque elemento in $A[1..i-1] \Rightarrow$ scambiando $A[i]$ con $A[k]$ e incrementando i di 1 l'invariante si mantiene



34

Ordinamento per selezione

Invariante del ciclo interno:

- $A[k]$ è minimo in $A[i..j-1]$



Inizializzazione:

- con $k=i$ e $j=i+1$ l'invariante si riduce a " $A[i]$ è minimo in $A[i..i]$ " e questo è evidente che sia vero

35

Ordinamento per selezione

Invariante del ciclo interno:

- $A[k]$ è minimo in $A[i..j]$



Mantenimento:

- come ipotesi induttiva assumiamo che l'invariante vale prima di eseguire il ciclo
- il corpo del ciclo aggiorna la posizione del massimo se $A[k] > A[j]$ e, in ogni caso, incrementa j
- dunque l'invariante viene mantenuto

36

Ordinamento per selezione

Invariante del ciclo interno:

- $A[k]$ è minimo in $A[i..j-1]$
- quando si esce dal ciclo $j=n+1$ quindi $A[k]$ è minimo in $A[i..n]$ dunque il minimo si trova correttamente

Invariante del ciclo esterno:

- $A[1..i-1]$ è ordinato
- se x è in $A[i..n]$ ed y è in $A[1..i-1]$ allora $x \geq y$
- quando si esce dal ciclo $i=n$ quindi $A[i..n]$ è ordinato e $A[n]$ è maggiore uguale di qualunque elemento di $A[1..n-1]$, dunque il vettore è ordinato e l'algoritmo è corretto

$A[1..n]$



37

Complessità di Select-Sort

- come nel caso si Insert-Sort possiamo contare per ogni riga quante volte viene eseguito
- **caso migliore:** il minimo si trova sempre all'inizio della parte non ordinata (k non viene mai aggiornato)
- **caso peggiore:** la parte non ordinata in realtà è ordinata decrescente e quindi k viene aggiornato dopo ogni confronto
- in tutti e due i casi la funzione $T_{\text{sel}}(n)$ (il costo di eseguire Select-Sort) è un polinomio di secondo grado
- questo succede perché j in ogni caso deve arrivare in fondo delle parte non ordinata



Sia nel caso migliore sia nel caso peggiore Select-Sort ha **complessità temporale quadratica**.

38

Insertion-Sort vs Select-Sort

```

INSERTION-SORT(A)
for i ← 2 to length(A) do
  > inserisce A[i] in A[1..i-1]
  j ← i
  while j > 1 and A[j-1] > A[j] do
    scambia A[j-1] con A[j]
    j ← j - 1
  end while
end for
return A

```

$C^{\min}(n)$ = n. confronti nel caso migliore

$C^{\max}(n)$ = n. confronti nel caso peggiore

$S^{\min}(n)$ = n. spostamenti nel caso migliore

$S^{\max}(n)$ = n. spostamenti nel caso peggiore

```

SELECT-SORT(A)
for i ← 1 to length(A) - 1 do
  k ← i
  for j ← i + 1 to length(A) do
    if A[k] > A[j] then
      k ← j
    end if
  end for
  scambia A[i] con A[k]
end for
return A

```

39

Insertion-Sort vs Select-Sort

```

INSERTION-SORT(A)
for i ← 2 to length(A) do
  ▷ insere A[i] in A[1..i-1]
  j ← i
  while j > 1 and A[j-1] > A[j] do
    scambia A[j-1] con A[j]
    j ← j - 1
  end while
end for
return A

```

$$C_{Ins}^{min}(n) = ???$$

$$C_{Ins}^{max}(n) = ???$$

$$S_{Ins}^{min}(n) = ???$$

$$S_{Ins}^{max}(n) = ???$$

```

SELECT-SORT(A)
for i ← 1 to length(A) - 1 do
  k ← i
  for j ← i + 1 to length(A) do
    if A[k] > A[j] then
      k ← j
    end if
  end for
  scambia A[i] con A[k]
end for
return A

```

$$C_{Sel}^{min}(n) = ???$$

$$C_{Sel}^{max}(n) = ???$$

$$S_{Sel}^{min}(n) = ???$$

$$S_{Sel}^{max}(n) = ???$$

40

Alberi di decisione

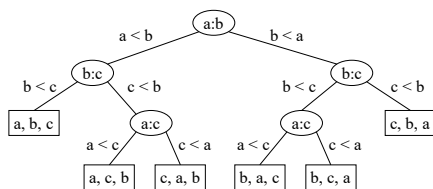
Un albero rappresenta le esecuzioni di un algoritmo:

- i **nodi interni** rappresentano **decisioni da prendere**
- le **foglie** rappresentano possibili **uscite (output)**
- i **rami** rappresentano particolari **esecuzioni (secondo il risultato decisione)**

L'albero di decisione che minimizza l'altezza fornisce un confine inferiore al numero di decisioni necessarie nel caso peggiore.

41

L'albero per l'ordinamento di 3 el.



42

Il problema dell'ordinamento

Nel caso dell'ordinamento:

- $n!$ foglie (un ordinamento è una permutazione)
- i nodi interni rappresentano confronti

In un albero binario per avere k foglie ci vogliono almeno $\log_2 k$ livelli.

Nel caso dell'ordinamento (*sorting*) il numero dei confronti deve essere dunque maggiore di (usando la formula di Stirling per approssimare $n!$):

$$\log_2 n! \approx \log_2 \left(\sqrt{2\pi n} (n/e)^n \right) = \log_2 \sqrt{2\pi n} + n \log_2 (n/e) \approx n \log_2 n$$
