Corso di laurea in Informatica
Introduzione agli Algoritmi
A.A. 2024/25

Il problema dell'ordinamento: Ordinamenti naif

Tiziana Calamoneri

SAPIENZA
UNIVERSITÀ DI ROMA

Slides realizzate sulla base di quelle preparate da
T. Calamoneri e G. Bongiovanni per il corso di
Informatica Generale tenuto a distanza nell'A.A. 2019/20

Il problema dell'ordinamento

Il problema dell'ordinamento degli elementi di un insieme è un problema molto ricorrente in informatica poiché ha un'importanza fondamentale per le applicazioni: lo si ritrova molto frequentemente come sottoproblema nell'ambito dei problemi reali.

Si stima che una parte rilevante del tempo di calcolo complessivo consumato nel mondo sia relativa all'esecuzione di algoritmi di ordinamento.

Algoritmi di ordinamento (1)

Un algoritmo di ordinamento è un algoritmo capace di ordinare gli elementi di un insieme sulla base di una certa relazione d'ordine, definita sull'insieme stesso.

Per semplicità di trattazione, supponiamo che gli n elementi da ordinare siano numeri interi e siano contenuti in un array i cui indici vanno da 0 ad n-1.

Algoritmi di ordinamento (2)

Tuttavia, nei problemi reali, i dati da ordinare sono ben più complessi: in generale essi sono strutturati in record, cioè in gruppi di informazioni non sempre omogenee relative allo stesso soggetto, e si vuole ordinarli rispetto ad una di tali informazioni (ad esempio, il Codice Fiscale).

Algoritmi di ordinamento (3)

Esistono diversi algoritmi di ordinamento.
Partiamo da quelli più semplici ma, volendo migliorare l'efficienza, dobbiamo mettere in campo idee più elaborate.

Gli algoritmi semplici che illustreremo sono:

Insertion sort Selection sort
Bubblesort

Gli algoritmi più evoluti che vedremo sono:

Mergesort Quicksort Heapsort

Insertion sort (1)

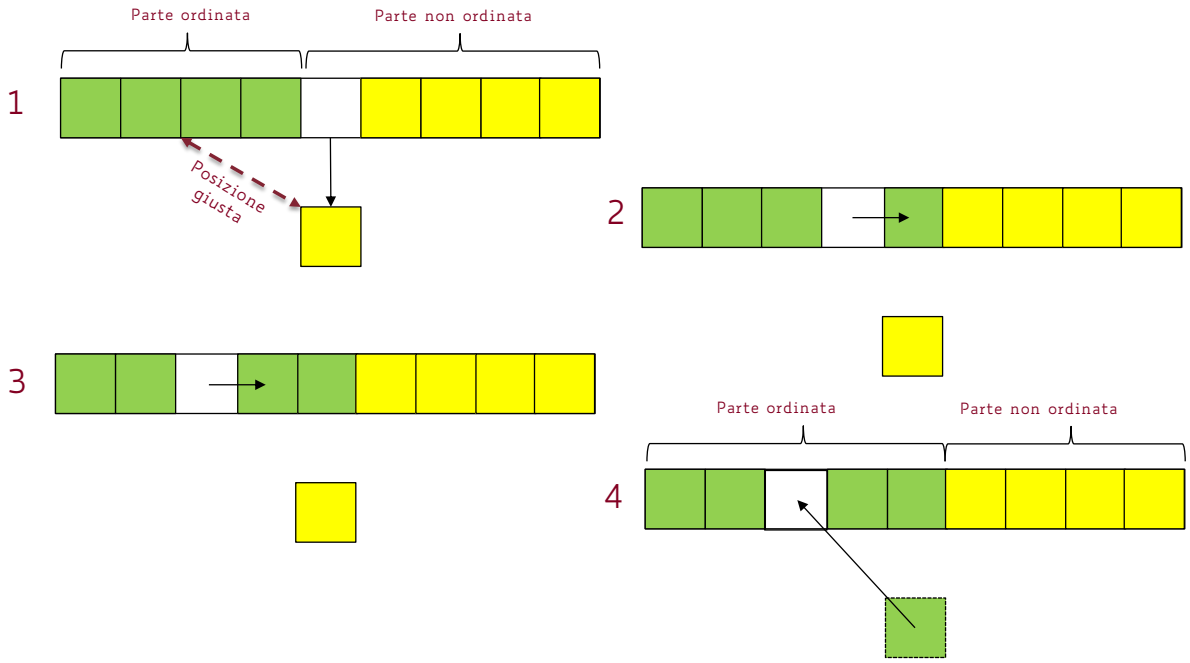
- Gli elementi da ordinare sono inizialmente contenuti in un array.
- Per ogni i=0, …, n-1:
  Si estrae l'elemento della posizione i, così da liberare la sua posizione corrente
- Si spostano di una posizione verso destra tutti gli elementi alla sua sinistra (già ordinati) che sono maggiori di esso
- Si inserisce l'elemento nella posizione che si è liberata.

Insertion sort (2)

INVARIANTE: ad ogni iterazione j, gli elementi con indice <j (a sinistra) sono già ordinati, mentre quelli con indice >j (a destra) sono ancora da processare.

L'algoritmo procede per iterazioni, aumentando ad ogni iterazione la dimensione della porzione di sinistra dell'array, che è ordinata.

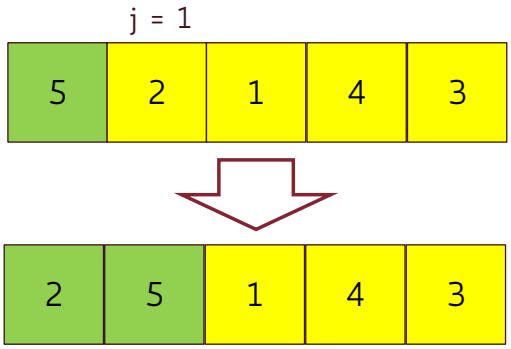
Insertion sort (3)



Insertion sort (4)

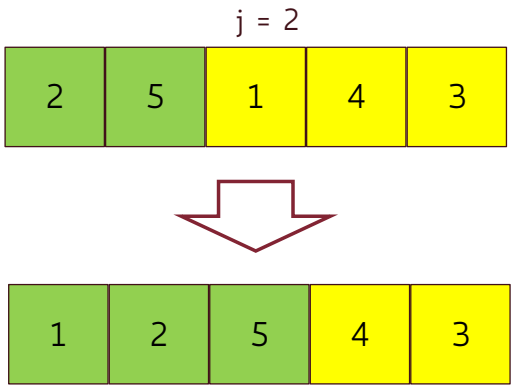
esempio.

Un array di un solo elemento è già ordinato, quindi si estrae per primo il secondo elemento. L'indice usato nel codice per l'elemento da sistemare è j



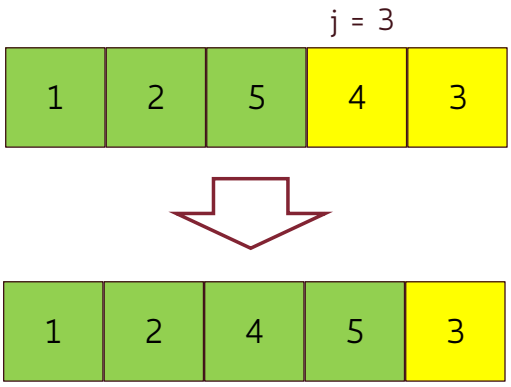
Insertion sort (5)

Nella successiva iterazione j = 2



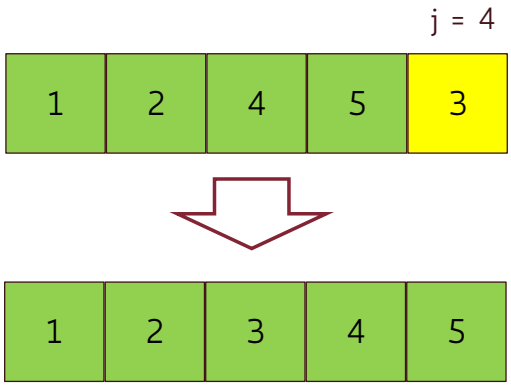
Insertion sort (6)

Nella successiva iterazione j = 3



Insertion sort (7)

Nell'ultima iterazione, per j = 4:



Dopodiché l'array risulta ordinato.

Insertion sort (8)

Lo pseudocodice dell'algoritmo è il seguente.

```
def Insertion_Sort(A)
for j in range(1,len(A)):          (n-1) Θ(1) + Θ(1)
    x = A[j]                          Θ(1)
    i = j - 1                         Θ(1)
    while ((i>=0)and(A[i])>x)         t_j Θ(1)+ Θ(1)
      A[i+1] = A[i]                     Θ(1)
      i = i - 1                         Θ(1)
    A[i+1] = x                       Θ(1)
```

Costo computazionale:

$$T(n) = \sum_{j=1}^{n-1}\left(\Theta(1) + t_j\Theta(1) + \Theta(1)\right) + \Theta(1)$$

Insertion sort (9)

$$T(n) = \sum_{j=1}^{n-1}\left(\Theta(1) + t_j\Theta(1) + \Theta(1)\right) + \Theta(1)$$

Il numero di iterazioni del while interno, t_j, può andare:

- da un minimo di 1 per ogni j (se ogni x > A[j-1])
- a un massimo di j per ogni j (se ogni x < A[1])

Quindi caso migliore e caso peggiore differiscono:

Caso migliore: $T(n) = (n-1)\Theta(1) = \Theta(n)$

Caso peggiore: $T(n) = \sum_{j=1}^{n-1}\left(\Theta(1) + \Theta(j)\right) =$

$$= \Theta(n) + \Theta(n^2) = \Theta(n^2)$$

Selection sort (1)

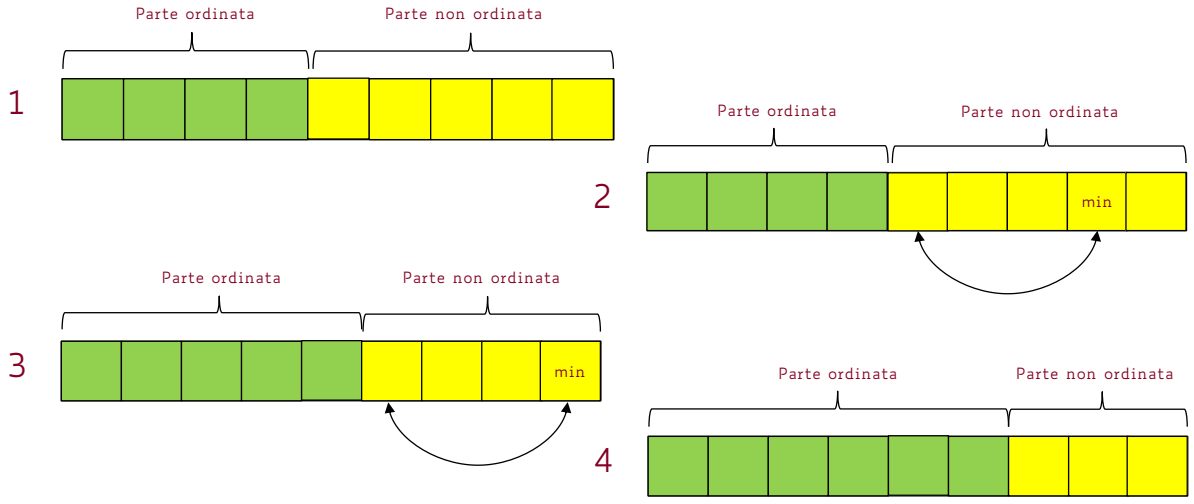
- Cerca il minimo dell'intero array e lo mette in prima posizione (con uno scambio);
- Cerca il nuovo minimo nell'array restante, cioè nelle posizioni dalla seconda all'ultima incluse, e lo mette in seconda posizione (con uno scambio);
- Cerca il minimo nelle posizioni dalla terza all'ultima incluse e lo mette in terza posizione;
- e così via…

Selection sort (2)

INVARIANTE: ad ogni iterazione i, gli elementi con indice <i (a sinistra) sono già ordinati, mentre quelli con indice >i (a destra) sono ancora da processare.

L'algoritmo procede per iterazioni, aumentando ad ogni iterazione la dimensione della porzione di sinistra dell'array, che è ordinata.

Selection sort (3)

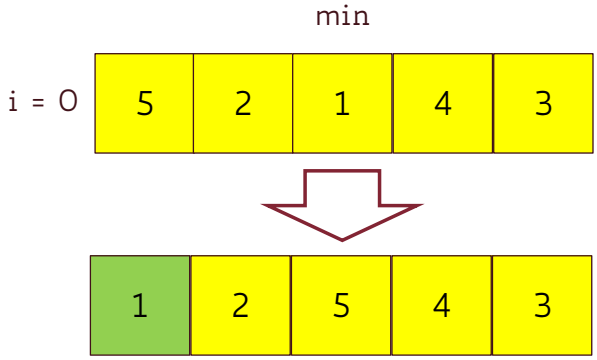


ecc.

Selection sort (4)

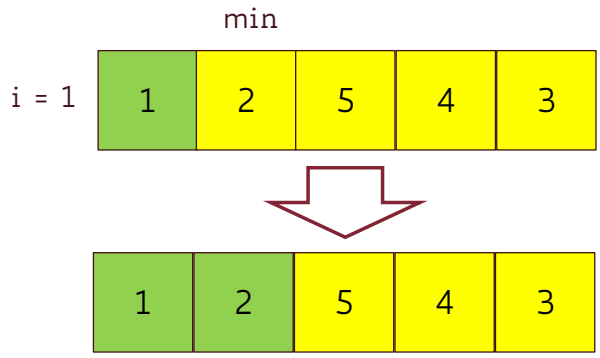
esempio.

L'indice usato nel codice per l'elemento da sistemare è i



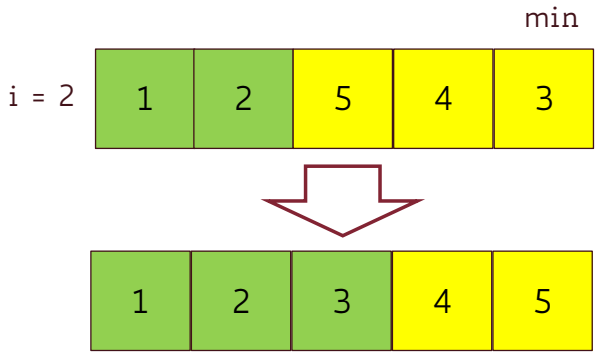
Selection sort (5)

Per i = 1 il minimo, in questo esempio, viene rimesso nella sua posizione:



Selection sort (5)

Successiva iterazione, per i = 2:



E così via. In questo esempio le ultime due iterazioni rimettono ciascuno dei due elementi al proprio posto

Selection sort (6)

Lo pseudocodice dell'algoritmo è il seguente.

```
def Selection_Sort(A)
for i in range(len(A)-1):          (n - 1) Θ(1) + Θ(1)
  m = i                              Θ(1)
  for j in range(i+1,len(A)):      (n - i) Θ(1) + Θ(1)
      if (A[j] < A[m])               Θ(1)
        m = j                        Θ(1)
    A[m],A[i]=A[i],A[m]            Θ(1)
```

Costo computazionale:

$T(n)=\sum_{i=0}^{n-2}\big(\Theta(1)+(n-i)\Theta(1)+\Theta(1)\big)+\Theta(1)=\sum_{i=0}^{n-2}\big(i\Theta(1)+\Theta(1)\big)=\Theta(n^2)$

In questo algoritmo non c'è differenza fra i costi di caso migliore e peggiore.

Bubble sort (1)

Ad ogni iterazione:

- ispeziona una dopo l'altra, da destra a sinistra, ogni coppia di elementi adiacenti e, se l'ordine dei due elementi non è quello giusto, essi vengono scambiati.

La prima iterazione si ferma alla prima posizione e sistema il minimo nella posizione corretta.

La seconda iterazione si ferma alla seconda posizione e sistema il secondo minimo.

E così via per le iterazioni successive.

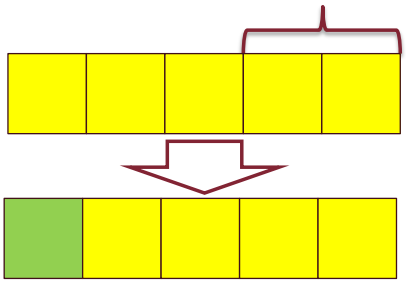
Bubble sort (2)

INVARIANTE: ad ogni iterazione i, gli elementi con indice <i (a sinistra) sono già ordinati, mentre quelli con indice >i (a destra) sono ancora da processare.

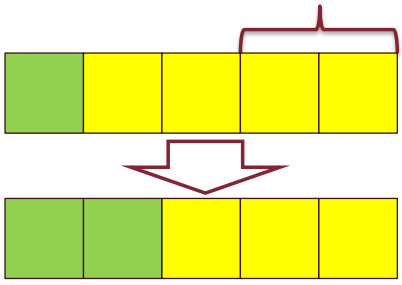
L'algoritmo procede per iterazioni, aumentando ad ogni iterazione la dimensione della porzione di sinistra dell'array, che è ordinata.

Bubble sort (3)

Prima iterazione:



Seconda iterazione:



Bubble sort (3)

Lo pseudocodice dell'algoritmo è il seguente.

```
def Bubble_Sort(A)
   for i in range(len(A)):    n Θ(1) + Θ(1)
      for j in range(len(A)-1,i,-1):    (n – i) Θ(1) + Θ(1)
         if (A[j] < A[j - 1])              Θ(1)
            A[j], A[j - 1]=A[j-1],A[j]     Θ(1)
```

Costo computazionale:

$T(n)=\sum_{i=0}^{n-1}\left(\Theta(1)+(n-i)\Theta(1)+\Theta(1)\right)+\Theta(1)=\Theta(n^2)$

Anche in questo algoritmo non c'è differenza fra i costi di caso migliore e peggiore.

Animazioni degli algoritmi di ordinamento

In rete sono disponibili diverse animazioni di vari algoritmi di ordinamento. Queste sono particolari.

- Insertion sort:
  - https://www.youtube.com/watch?v=EdIKIf9mHkO
- Selection sort:
  - https://www.youtube.com/watch?v=O-W8OEwLebQ
- Bubble sort:
  - https://www.youtube.com/watch?v=semGJAJ7i74

La complessità dell'ordinamento (1)

I tre algoritmi di ordinamento appena visti hanno tutti un costo computazionale asintotico che cresce come il quadrato del numero di elementi da ordinare.

Domanda 1: si può fare di meglio?
Risposta: si, se troviamo un algoritmo X con costo minore

Domanda 2: Se si, quanto meglio si può fare?
Risposta: Non sappiamo: chi ci assicura che non si possa fare ancora meglio dell'ipotetico algoritmo X?

La complessità dell'ordinamento (2)

Per stabilire un limite al di sotto del quale il costo computazionale di nessun algoritmo di ordinamento basato su confronti possa andare, uno strumento adatto è l'albero di decisione.
Esso permette di rappresentare tutte le strade che la computazione di uno specifico algoritmo può intraprendere, sulla base dei possibili esiti dei test previsti dall'algoritmo stesso.

La complessità dell'ordinamento (3)

Nel caso degli algoritmi di ordinamento basati su confronti, ogni test effettuato ha due soli possibili esiti (ad es.: minore o uguale, oppure no).
L'albero di decisione relativo a un qualunque algoritmo di ordinamento basato su confronti ha queste proprietà:

La complessità dell'ordinamento (4)

- è un albero binario che rappresenta tutti i possibili confronti che vengono effettuati dall'algoritmo; ogni nodo interno (ossia un nodo che non è una foglia) ha esattamente due figli;
- …

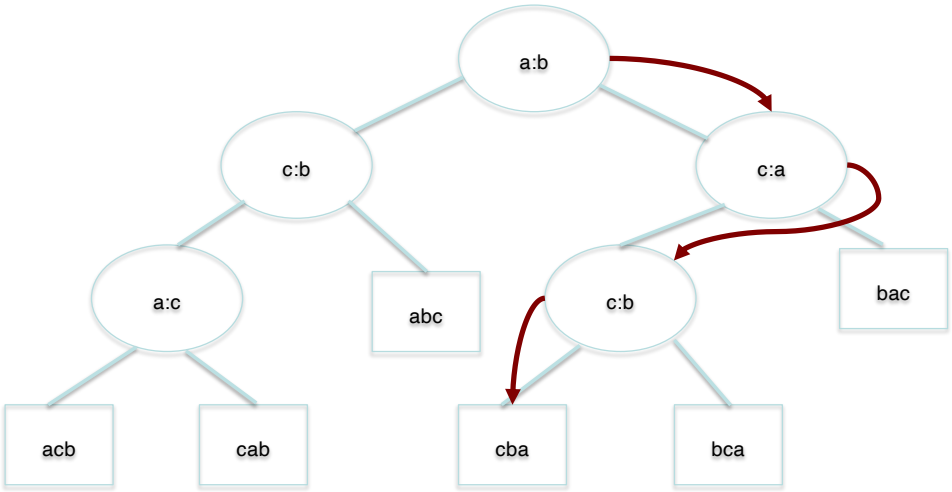
La complessità dell'ordinamento (5)

- …
- ogni nodo interno rappresenta un singolo confronto, ed i due figli del nodo sono relativi ai due possibili esiti di tale confronto;

- ogni foglia rappresenta una possibile soluzione del problema, la quale è una specifica permutazione della sequenza in ingresso.

La complessità dell'ordinamento (6)

Esempio: albero di decisione dell'insertion sort su 3 elementi a, b, c



La complessità dell'ordinamento (7)

Eseguire l'algoritmo corrisponde a scendere dalla radice dell'albero alla foglia che contiene la permutazione che costituisce la soluzione; la discesa è governata dagli esiti dei confronti che vengono via via effettuati.

La complessità dell'ordinamento (8)

La lunghezza (= numero degli archi) di tale cammino rappresenta il numero di confronti necessari per trovare la soluzione.
La lunghezza del percorso più lungo dalla radice ad una foglia (altezza dell'albero binario) rappresenta il numero di confronti che l'algoritmo deve effettuare nel caso peggiore.

La complessità dell'ordinamento (9)

Determinare una limitazione inferiore all'altezza dell'albero di decisione relativo a qualunque algoritmo di ordinamento basato su confronti equivale a trovare una limitazione inferiore al tempo di esecuzione nel caso peggiore di qualunque algoritmo di ordinamento basato su confronti.

La complessità dell'ordinamento (10)

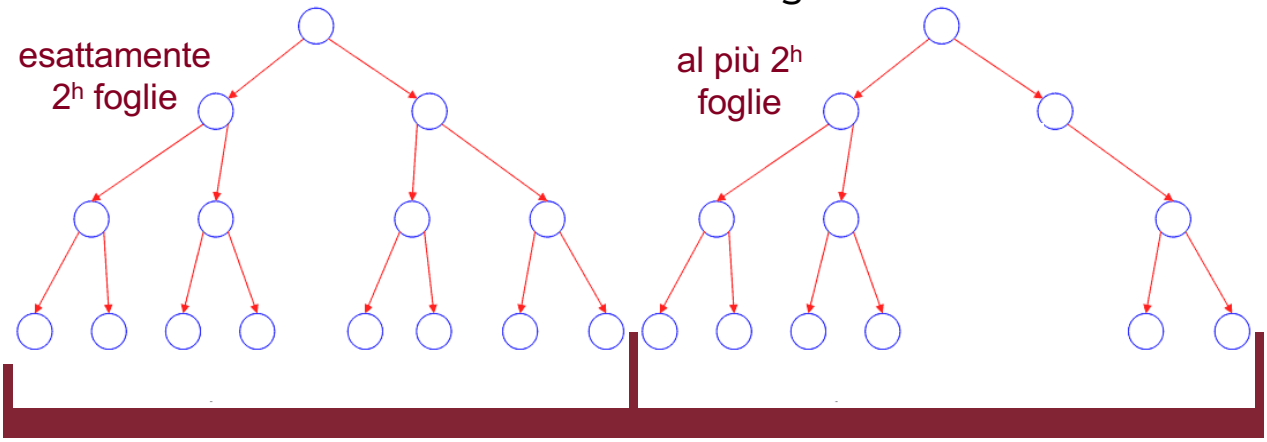
Cerchiamo di determinare tale limitazione:

Osservazione 1: dato che la soluzione può essere una qualunque permutazione della sequenza di ingresso, l'albero di decisione deve contenere nelle foglie tutte le permutazioni della sequenza in ingresso, che sono n! per un problema di dimensione n.

La complessità dell'ordinamento (11)

Osservazione 2: un albero binario di altezza h non può contenere più di 2^h foglie.
Segue che l'altezza h dell'albero di decisione di qualunque algoritmo di ordinamento basato su confronti deve essere tale per cui:

$$2^h \geq n! \text{ ossia } h \geq \log n!$$



La complessità dell'ordinamento (12)

Valutiamo log n! (per semplicità assumiamo che n sia pari).

$$\log n! = \log(n*(n-1)*(n-2)*(n-3)\ldots*2*1) =$$

$$= \sum_{i=1}^{n} \log i = \sum_{i=1}^{n/2} \log i + \sum_{i=\frac{n}{2}+1}^{n} \log i \geq$$

$$\geq \quad 0 \quad + \sum_{i=\frac{n}{2}+1}^{n} \log \frac{n}{2} =$$

$$= \frac{n}{2} \log n - \frac{n}{2} = \Theta(n \log n)$$

Siccome abbiamo visto che $h \geq \log n!$ e che
$\log n! \geq \Theta(n \log n)$
ciò significa che $h = \Omega(n \log n)$

La complessità dell'ordinamento (13)

Quanto detto si riassume nel seguente:

Teorema.
Il costo computazionale di qualunque algoritmo di ordinamento basato su confronti è $\Omega(n \log n)$.

Esercizio svolto (1)

Nell'algoritmo di InsertionSort è possibile ricercare la posizione in cui inserire l'elemento *i*-esimo tramite la ricerca binaria. Come cambia il calcolo del costo computazionale dell'algoritmo?

Soluzione. Per ogni *j* da 1 ad *n-1*, l'algoritmo di InsertionSort in versione standard:

- scorre la parte ordinata di array confrontando ciascun elemento con *x*
- trova la posizione corretta di *x*
- sposta tutti gli elementi che sono a destra per fare posto ad *x*

Esercizio svolto (2)

Consideriamo lo pseudocodice modificato dell'Insertion Sort (sfrutta `RicBinMod(A,j,x)`, che trova l'indice corrispondente alla posizione che deve assumere x nell'array A[0,j-1])

```
def Bin_Insertion_Sort(A)
  for j in range(1,len(A)):          (n-1) Θ(1) + Θ(1)
    x = A[j]                         Θ(1)
    k = RicBinMod(A,j,x)             O(log j)
    i = j - 1                        Θ(1)
    while ((i>=0)and(i>=k))          t_j Θ(1)+ Θ(1)
      A[i+1] = A[i]                    Θ(1)
      i = i - 1                        Θ(1)
    A[k] = x                         Θ(1)
```


Esercizio svolto (3)

- Il contributo della ricerca binaria non si sostituisce a t_j, ma si aggiunge ad esso.
- Anche se troviamo la posizione di x più velocemente, dobbiamo comunque spostare gli elementi a destra…
- pertanto il costo computazionale non migliora!

Corso di laurea in Informatica
Introduzione agli Algoritmi
A.A. 2024/25

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

Esercizi (1)

- Si consideri l'algortimo BubbleSort in modo che nelle sue varie fasi la parte ordinata che va via via crescendo sia la parte destra. Funziona ancora correttamente? Il costo computazionale dell'algoritmo cambia?
- Un algoritmo di ordinamento si dice stabile se, in caso in cui ci siano più occorrenze dello stesso valore nell'array, esso le lascia nell'ordine in cui si trovano originariamente. Qualcuno tra i tre algoritmi visti in questa lezione è stabile?

Esercizi (2)

- Qual è il costo computazionale dei tre algoritmi visti in questa lezione quando l'array di input ha gli elementi che sono tutti uguali? e quando è già ordinato?
- Scrivere una funzione che, dato un array di n elementi ed un indice j, trovi il minimo del sottoarray A[j..n]. Riscrivere lo pseudocodice del SelectionSort sfruttando questa funzione.

Esercizi (3)

- Dato un array di n elementi, sulla base di quanto appreso fin ora, si progetti un algoritmo che verifichi se ci sono occorrenze ripetute di uno stesso valore; restituisca 1 se ve ne sono e 0 altrimenti. Si calcoli il suo costo computazionale.