Corso di laurea in Informatica
Introduzione agli Algoritmi

Il problema dell'ordinamento: algoritmi naïf

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Il problema dell'ordinamento

I 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.

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 **vettore**.

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).

Esistono diversi algoritmi di ordinamento.
Dapprima illustreremo gli algoritmi più semplici ma ci renderemo conto che, volendo migliorare l'efficienza, sarà necessario mettere in campo idee più elaborate.

Gli algoritmi semplici che illustreremo sono:

- Selection sort
- Insertion sort
- Bubble sort

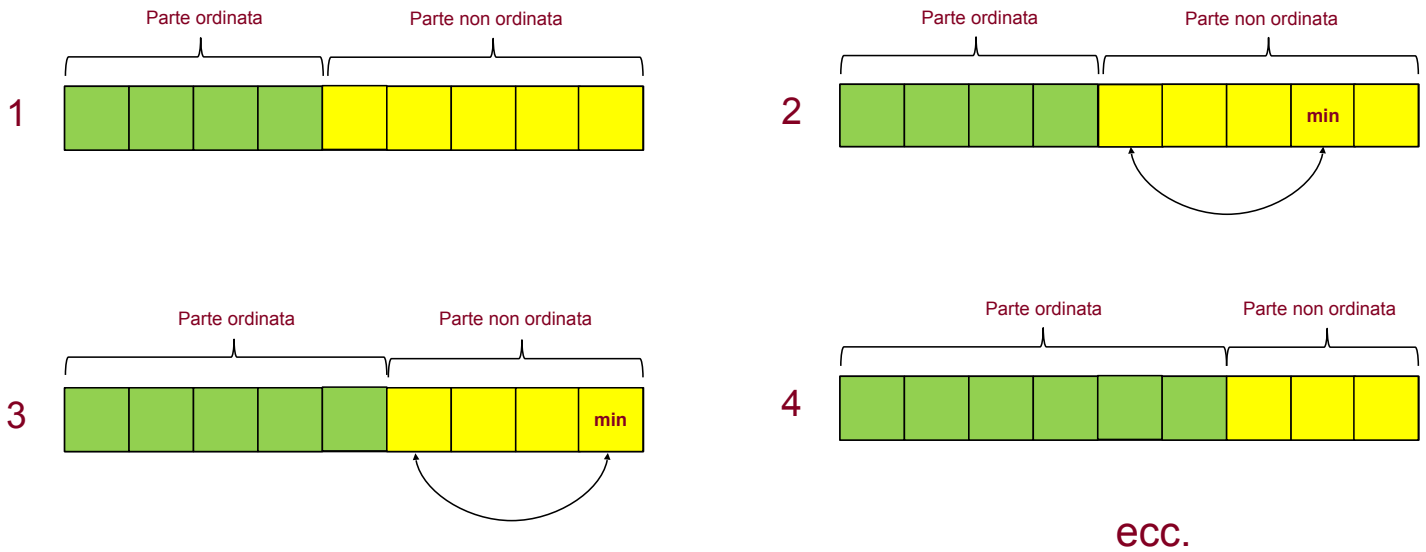
Gli algoritmi più evoluti che vedremo sono:

- Mergesort
- Quicksort
- Heapsort

Selection Sort

Selection sort è un algoritmo di ordinamento che seleziona iterativamente l'elemento più piccolo dall'array non ancora ordinato e lo sposta nella posizione corretta nell'array ordinato. L'algoritmo continua fino a quando tutti gli elementi non sono stati inseriti nell'array ordinato.

1. Inizialmente, l'array è considerato come diviso in due parti: l'array ordinato e l'array non ordinato. L'array ordinato è vuoto, mentre l'array non ordinato contiene tutti gli elementi dell'array originale.
2. In ogni passo dell'algoritmo, si cerca l'elemento più piccolo nell'array non ordinato e lo si scambia con il primo elemento dell'array non ordinato. In questo modo, l'elemento più piccolo viene spostato nell'inizio dell'array non ordinato e nell'array ordinato viene aggiunto un nuovo elemento.
3. L'array ordinato viene quindi esteso per includere l'elemento appena aggiunto.
4. L'algoritmo continua ad eseguire questi passaggi fino a quando tutti gli elementi non sono stati inseriti nell'array ordinato.



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

$n-1$ **iterazioni**

$n-i-1$ **iterazioni**

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

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

$$= \Theta(1) + \Theta(n^2)$$

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

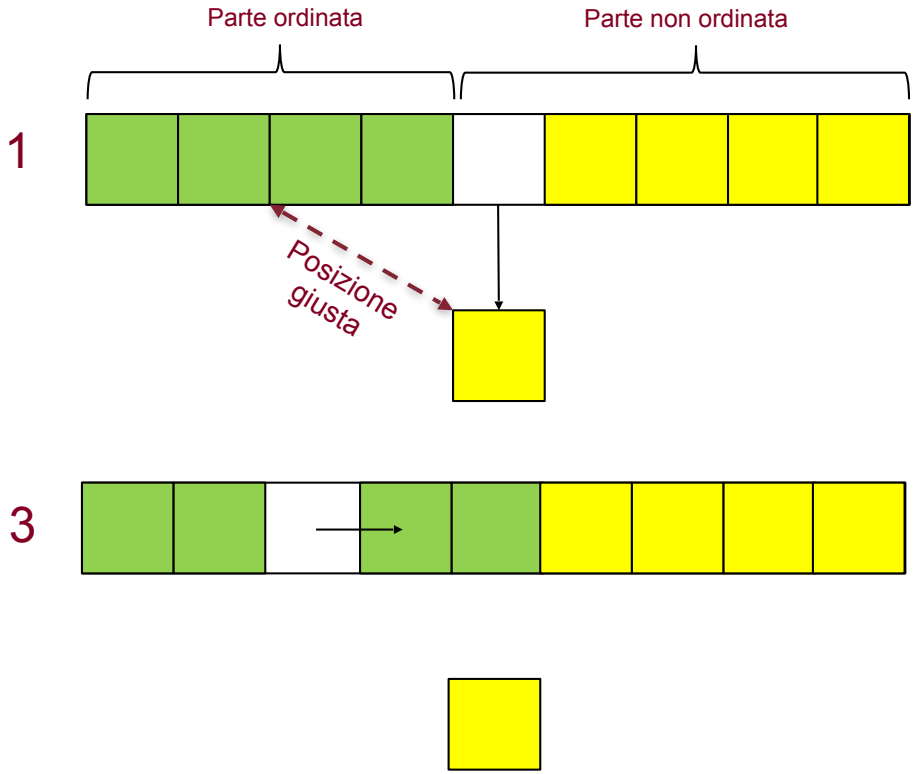
ricorda che $\sum_{j=1}^{x} j = \Theta(x^2)$

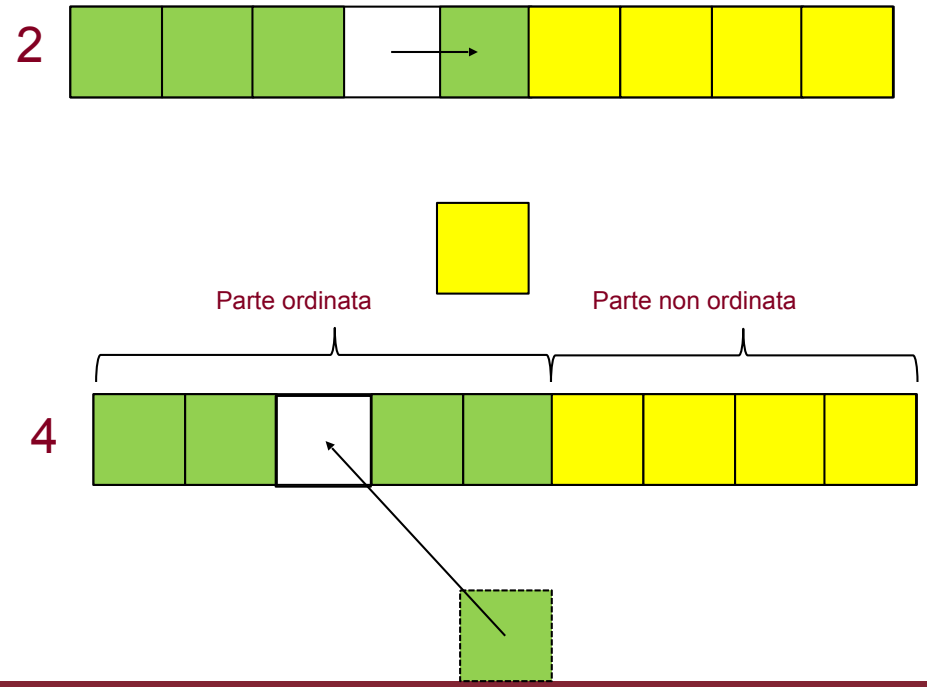
In questo algoritmo non c'è distinzione tra i costi del caso migliore ed il caso peggiore.

Insertion sort

Il funzionamento dell'algoritmo di Insertion Sort è il seguente:

1. L'array è diviso in due parti: la parte ordinata e la parte non ordinata. Inizialmente, la parte ordinata contiene solo il primo elemento dell'array e la parte non ordinata contiene tutti gli altri elementi.
2. L'algoritmo prende il primo elemento della parte non ordinata e lo confronta con gli elementi della parte ordinata. Se l'elemento è minore di un elemento della parte ordinata, l'elemento viene spostato a destra fino a trovare la sua posizione corretta.
3. L'array ordinato viene esteso per includere l'elemento appena aggiunto.
4. L'algoritmo continua ad eseguire questi passaggi fino a quando tutti gli elementi dell'array non sono stati ordinati.





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

$n-1$ **iterazioni**

$O(i)$ **iterazioni**

$$T(n) = \sum_{i=1}^{n} O(i)$$

Caso migliore: si verifica quando ad ogni iterazione si ha $O(i) = \Theta(1)$.
Questo accade quando la lista è già ordinata:

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

Caso peggiore: si verifica quando ad ogni iterazione si ha $O(i) = \Theta(i)$.
Questo accade quando la lista è in ordine inverso (ad ogni passo l'elemento selezionato va inserito al primo posto e faccio $i-1$ passi)

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

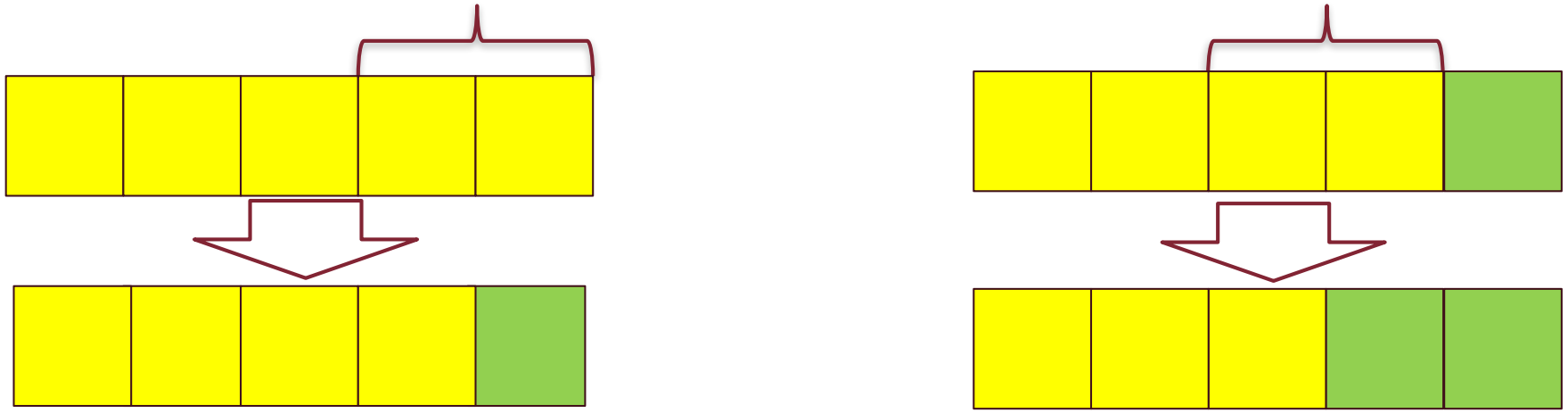
dove ho usato che $\sum_{i=1}^{x} i = \Theta(x^2)$

La complessità dell'algoritmo è $O(n^2)$

Bubble sort

Anche l'algoritmo bubble sort funziona per fasi. Ogni fase ispeziona una dopo l'altra, da sinistra a destra, ogni coppia di elementi adiacenti e, se l'ordine dei due elementi non è quello giusto, essi vengono scambiati.

La prima «passata» sale fino all'ultima posizione e sistema il massimo nella posizione corretta.
La seconda passata si ferma alla penultima posizione e sistema il secondo massimo.
E così via per le passate (iterazioni) successive.



Bubble sort

Lo pseudocodice dell'algoritmo è il seguente.

```
def BubbleSort(A):
    n=len(A)
    for i in range(n-1):
        for j in range(n-i-1):
            if A[j]>A[j+1]:
                A[j],A[j+1]=A[j+1],A[j]
```

$n-1$ **iterazioni**

$n-i-1$ **iterazioni**

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

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

Come per il selection sort anche in questo algoritmo non c'è distinzione tra i costi del caso migliore e quelli del caso peggiore.

La complessità dell'ordinamento

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 sapremmo, chi ci assicura che non si possa fare ancora meglio del nuovo algoritmo X?

La complessità dell'ordinamento

Come si fa a stabilire un limite di costo computazionale al caso pessimo ***al di sotto del quale nessun algoritmo di ordinamento basato su confronti fra coppie di elementi possa andare***?

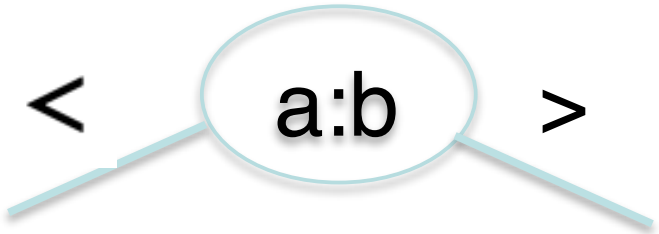
• Esiste uno strumento adatto allo scopo, l'***albero di decisione***, che permette di rappresentare tutte le strade che la computazione di uno specifico algoritmo può intraprendere, sulla base dei possibili esiti dei test previsti dall'algoritmo stesso.

• Nel caso degli algoritmi di ordinamento basati su confronti, ogni test effettuato ha due soli possibili esiti (ad es.: minore, oppure maggiore), in quanto assumiamo che i numeri da ordinare siano tutti distinti.

L’albero di decisione relativo a un qualunque algoritmo di ordinamento basato su confronti ha queste proprietà:

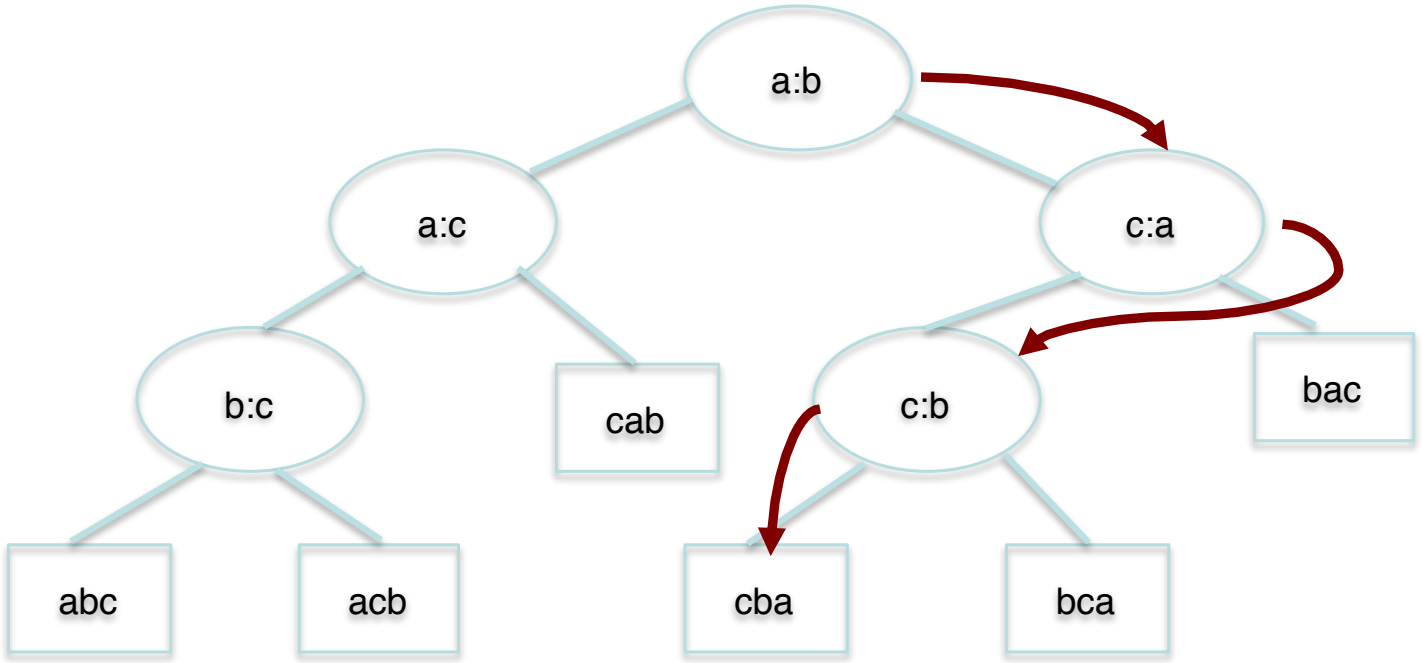
• è un albero binario che rappresenta tutti i possibili confronti che vengono effettuati dall’algoritmo; ogni nodo interno (ossia un nodo che non è una foglia) ha esattamente due figli;

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



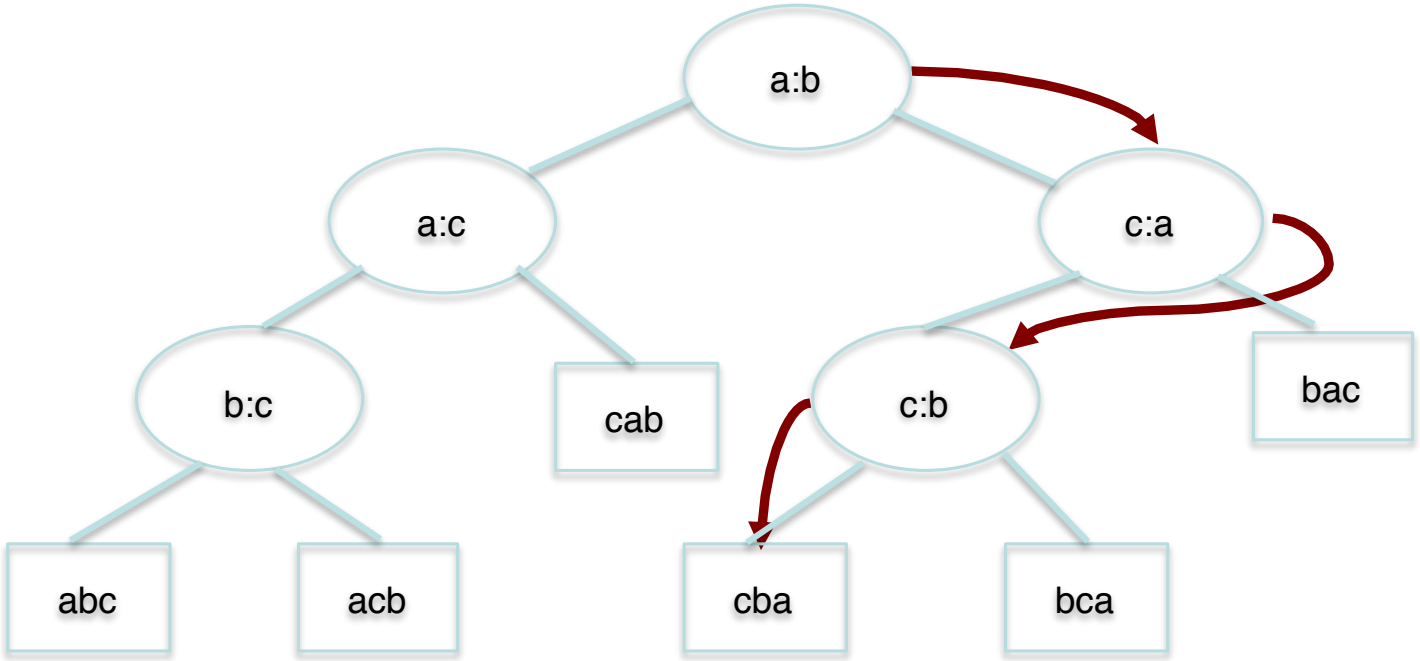
• ogni foglia rappresenta una possibile soluzione del problema, la quale è una specifica permutazione della sequenza in ingresso.

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



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 lunghezza (ossia il 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.

- 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.

Cerchiamo di determinare tale delimitazione:

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

•Osservazione 2: un albero binario di altezza h non può contenere più di 2^h foglie (**da provare per esercizio notando che l'albero di altezza h con più foglie le ha tutte a livello h**)

Dalle precedenti due osservazioni si ha che l'altezza *h* dell'albero di decisione di ***qualunque*** algoritmo di ordinamento basato su confronti deve essere tale per cui:

$$2^h \geq n!$$

vale a dire: $h \geq log_2(n!)$

valutiamo ora $\log(n!)$ (per semplicità assumiamo n pari)

$$\log_2(n!) = \log_2 n*(n-1)*(n-2)*(n-3)\ldots*2*1$$

$$\geq \log_2 \underbrace{n*(n-1)*(n-2)*(n-3)\ldots*\frac{n}{2}}_{\frac{n}{2} \text{ prodotti}}$$

$$\geq \log_2 \underbrace{\frac{n}{2}*\frac{n}{2}*\frac{n}{2}*\frac{n}{2}\ldots*\frac{n}{2}}_{\frac{n}{2} \text{ prodotti}}$$

$$\geq \log_2 \left(\frac{n}{2}\right)^{\frac{n}{2}}$$

$$= \frac{n}{2}\log_2\frac{n}{2} = \frac{n}{2}\log_2 n - \frac{n}{2} \quad = \Omega(n \log n)$$

Siccome abbiamo visto che $h \geq \log_2(n!)$ e che $\log_2(n!) = \Omega(n \log n)$ abbiamo:

$$h = \Omega(n \log n)$$

Possiamo riassumere quanto appena dimostrato col seguente teorema:

Teorema

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

Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZI 1

•Modificare l'algoritmo di Bubble sort in modo che nelle sue varie fasi la parte ordinata che va via via crescendo sia la parte sinistra. La complessità asintotica dell'algoritmo deve restare $\Theta(n^2)$.

•Un algoritmo di ordinamento si dice stabile se, in caso in cui ci siano più occorrenze dello stesso valore nel vettore, esso le lascia nell'ordine in cui si trovano originariamente. Qualcuno tra i tre algoritmi visti in questa lezione è stabile?

•Qual è il costo computazionale dei tre algoritmi visti in questa lezione quando la lista da ordinare ha gli elementi che sono tutti uguali? e quando è già ordinata? e quando e' ordinata in modo decrescente?

ESERCIZI 2

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

•Scrivere una funzione che, data una lista A di *n* elementi ed un indice *j,* trovi il minimo tra gli ultimi $n - j$ elementi della lista. Riscrivere lo pseudocodice del Selection sort sfruttando questa funzione.

•Data una lista di *n* elementi, si progetti un algoritmo che verifichi se ci sono occorrenze ripetute di uno stesso valore (e, ad esempio, restituisca 1 se ve ne sono e 0 altrimenti). Calcolare la complessità asintotica dell'algoritmo proposto.