SAPIENZA
UNIVERSITÀ DI ROMA
DIPARTIMENTO DI INFORMATICA

Corso di Introduzione agli algoritmi
Prof.ssa Tiziana Calamoneri

Il Problema dell'ordinamento: algoritmi naif

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

- 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 i cui indici vanno da 1 a *n*.
- 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.

Insertion sort (1)

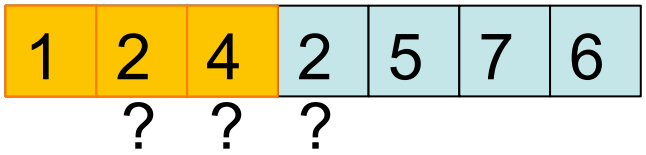
Per progettare un algoritmo di ordinamento, pensiamo dapprima a come ci comportiamo in situazioni reali…

Partendo da un mazzo vuoto, inseriamo una carta alla volta, cercando il punto giusto dove inserirla. Alla fine il mazzo di carte risulta ordinato…

Insertion sort (2)

Nel caso dell'algoritmo insertion sort:

- gli elementi da ordinare sono inizialmente contenuti nel vettore, quindi non si parte da un insieme vuoto ma si deve trovare la posizione giusta di ogni elemento (è come se si volessero ordinare le carte tenendole in mano, anziché appoggiandole e prendendole poi una per una).
- Ciò si effettua "estraendo" l'elemento, così da liberare la sua posizione corrente, spostando poi verso destra tutti gli elementi alla sua sinistra (già ordinati) che sono maggiori di esso ed, infine, inserendo l'elemento nella posizione che si è liberata.



e così via...

Insertion sort (3)

```
Funzione Insertion_Sort(A[1..n])
for j = 2 to n do                          n-1 volte
    x ← A[j]                               Θ(1)
    i ← j – 1                              Θ(1)
    while ((i > 0) and (A[i]) > x)         t_j volte
          A[i+1] ← A[i]                      Θ(1)
          i ← i – 1                          Θ(1)
    A[i+1] ← x                             Θ(1)
```

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

Caso migliore: $t_j=1$ per tutti i j: $T(n)=\Theta(n)$
Caso peggiore: $t_j=j$ per tutti i j: $T(n)=\Theta(n^2)$

Insertion sort (4)

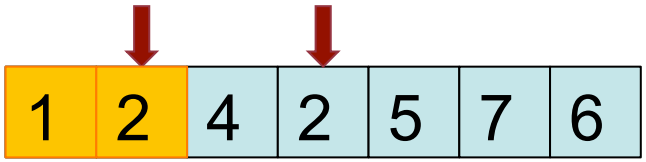
Per visualizzare l'insertion sort in modo inusuale, provate a guardare qui:

https://www.youtube.com/watch?v=EdIKIf9mHk0

Selection sort (1)

L'algoritmo selection sort:

- cerca il minimo dell'intero vettore e lo mette in prima posizione (con uno scambio);
- cerca il nuovo minimo nel vettore 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...



e così via...

Selection sort (2)

Funzione Selection_Sort(A[1..n])	
for i = 1 to n – 1 do	n-1 volte
m ← i	Θ(1)
for j = i + 1 to n do	n-i+1 volte
if (A[j] < A[m])	Θ(1)
m ← j	Θ(1)
Scambia A[m] e A[i]	Θ(1)

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

In tal caso non abbiamo casi migliore e peggiore:
$T(n)=\Theta(n^2)$

Selection sort (3)

Per visualizzare l'insertion sort in modo inusuale, provate a guardare qui:

https://www.youtube.com/watch?v=0-W8OEwLebQ

Bubble sort (1)

L'algoritmo bubble sort, ispeziona una dopo l'altra ogni coppia di elementi adiacenti e, se l'ordine dei due elementi non è quello giusto, essi vengono scambiati. L'algoritmo prosegue fino a che non vi sono più coppie di elementi adiacenti nell'ordine sbagliato.

Bubble sort (2)

```
Funzione Bubble_Sort(A[1..n])
for i = 1 to n do                          n volte
    for j = n downto i + 1 do              n-i+1 volte
        if (A[j] < A[j - 1])               Θ(1)
            Scambia A[j] e A[j - 1]        Θ(1)
```

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

Anche qui non dobbiamo distinguere i casi migliore e peggiore:
$T(n)=\Theta(n^2)$

Bubble sort (3)

Per visualizzare l'insertion sort in modo inusuale, provate a guardare qui:

https://www.youtube.com/watch?v=semGJAJ7i74

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 con costo minore

- Domanda 2: Se si, quanto meglio si può fare?
- Risposta: ??? chi ci assicura che non si possa fare ancora meglio?

Complessità dell'ordinamento (2)

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

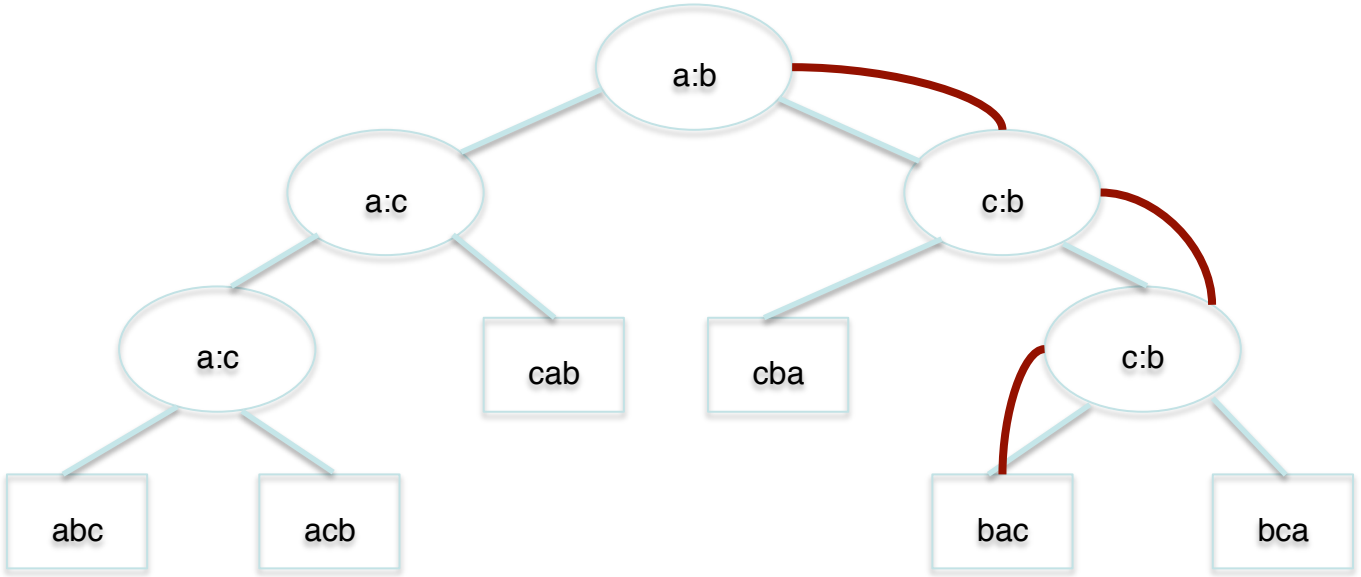
- Esiste uno strumento adatto allo scopo, l'***albero di decisione***, che permette di rappresentare tutte le strade che la computazione di uno specifico algoritmo può intraprendere, sulla base dei possibili esiti dei test previsti dall'algoritmo stesso.
- Nel caso degli algoritmi di ordinamento basati su confronti, ogni test effettuato ha due soli possibili esiti (ad es.: minore o uguale, oppure no).

Complessità dell'ordinamento (3)

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



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.

Complessità dell'ordinamento (5)

- 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.

.

Complessità dell'ordinamento (6)

Cerchiamo di determinare tale limitazione:

- Oss. 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*.
- Oss. 2. un albero binario di altezza *h* non può contenere più di 2^h foglie.
- Dalle precedenti oss. si ha che l'altezza *h* dell'albero di decisione di qualunque algoritmo di ordinamento basato su confronti deve essere tale per cui:

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

- Oss. 3. Approssimazione di Stirling: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n = \Theta(\sqrt{n} \cdot n^n)$

Segue che:

$$h \geq \log n! = \Theta(\log \sqrt{n} \cdot n^n) = \Theta(\log^{n+1/2}) = \Theta((n+1/2)\log n) = \Theta(n \log n)$$

Quanto detto si riassume nel seguente:

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

Esercizi (1)

- Si consideri l'algortimo Bubble sort e si sostituisca il ciclo interno for (j=n downto i+1) con for (j=n downto 1). 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 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 il vettore di input ha gli elementi sono tutti uguali? e quando è già ordinato?

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 calcolo costo computazionale dell'algoritmo?

- Scrivere una funzione che, dato un vettore di *n* elementi ed un indice *j,* trovi il minimo del sottovettore *A[j..n].* Riscrivere lo pseudocodice del Selection Sort sfruttando questa funzione.

- Dato un vettore di *n* elementi, si progetti un algoritmo che verifichi se ci sono occorrenze ripetute di uno stesso valore (e, ad esempio, restituisca 1 se vi sono e 0 altrimenti).