Corso di laurea in Informatica
Introduzione agli Algoritmi

Il problema dell'ordinamento: algoritmi naïf

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Il problema dell'ordinamento

L'ordinamento è un problema fondamentale in informatica che consiste nel riarrangiare gli elementi di un insieme in un ordine specifico, tipicamente crescente o decrescente.

Gli algoritmi di ordinamento sono essenziali per molte applicazioni, si stima che una parte rilevante del tempo di calcolo complessivo consumato nel mondo sia relativa all'esecuzione di algoritmi di ordinamento.

Per semplicità di trattazione, nel seguito supporremo sempre che gli 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 tantissimi algoritmi di ordinamento.
Dapprima illustreremo alcuni algoritmi più semplici, in seguito, 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**

Ordinamento per selezione (Selection Sort)

L'algoritmo di Selection Sort ordina un vettore trovando ripetutamente il minimo elemento nell'array non ordinato e spostandolo nella sua posizione finale.

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.

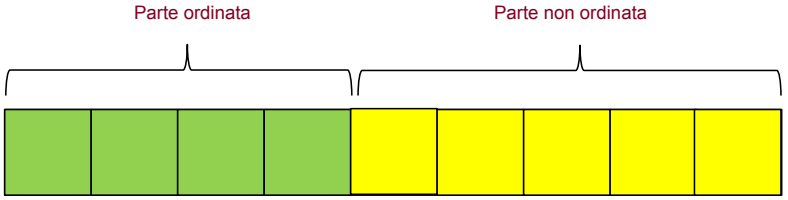
2. In ogni passo dell'algoritmo, si cerca l'elemento più piccolo nell'array non ordinato e lo si scambia con il primo elemento dell'array non ordinato. In questo modo, l'elemento più piccolo viene spostato nell'inizio dell'array non ordinato e nell'array ordinato viene aggiunto un nuovo elemento.

3. L'array ordinato viene quindi esteso per includere l'elemento appena aggiunto.

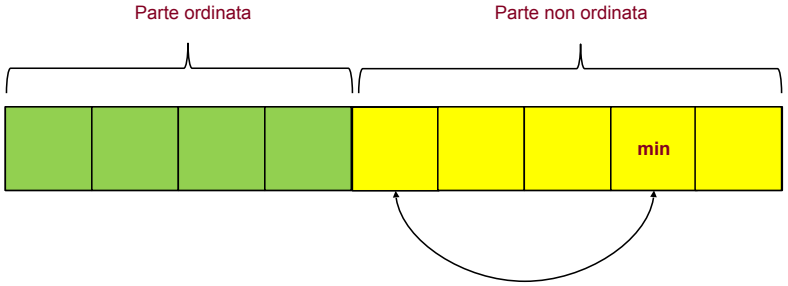
4. L'algoritmo continua ad eseguire questi passaggi fino a quando tutti gli elementi non sono stati inseriti nell'array ordinato.

Selection Sort

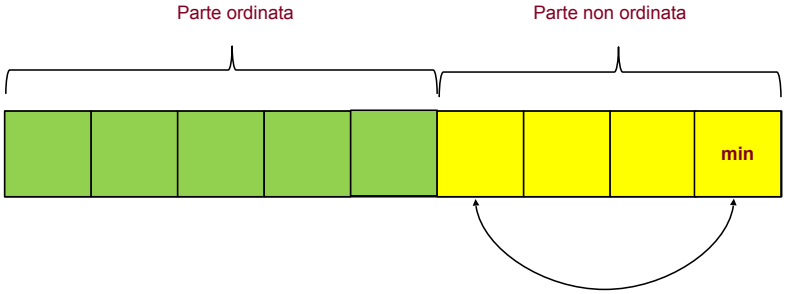
1



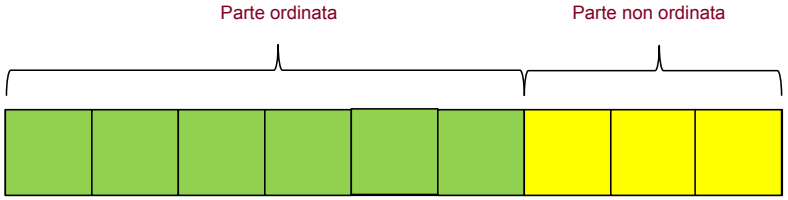
2



3



4



ecc.

```python
def SelectionSort(A):
    n = len(A)
    for i in range(n-1):
        # Trova il minimo nella parte non ordinata
        indice_min = i
        for j in range(i+1, n):
            if A[j] < A[indice_min]:
                indice_min = j
        # Scambia il minimo con il primo elemento non ordinato
        A[i], A[indice_min] = A[indice_min], A[i]
```

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

Selection Sort ha una complessità computazionale che dipende dai due cicli *for* annidati. Il ciclo esterno esegue $n-1$ iterazioni, mentre il ciclo interno esegue $n-i-1$ confronti per ogni iterazione del ciclo esterno, dove i è l'indice della iterazione corrente del ciclo esterno. Pertanto, la complessità totale dell'algoritmo è:

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

- In quest'algoritmo non c'è distinzione tra i costi del caso migliore e del caso peggiore.
- L'algoritmo ha il pregio di essere in-place , non richiede cioè spazio extra oltre a quello necessario a contenere gli elementi da ordinare.

Ordinamento per inserzione (Insertion Sort)

L'algoritmo funziona costruendo gradualmente un array ordinato. L'idea di base è quella di iterare attraverso gli elementi non ordinati del vettore, "inserendoli" nella loro posizione corretta nella parte ordinata del vettore. Ogni elemento viene confrontato con gli elementi precedenti e inserito nella posizione corretta spostando gli elementi maggiori di esso di una posizione verso destra.

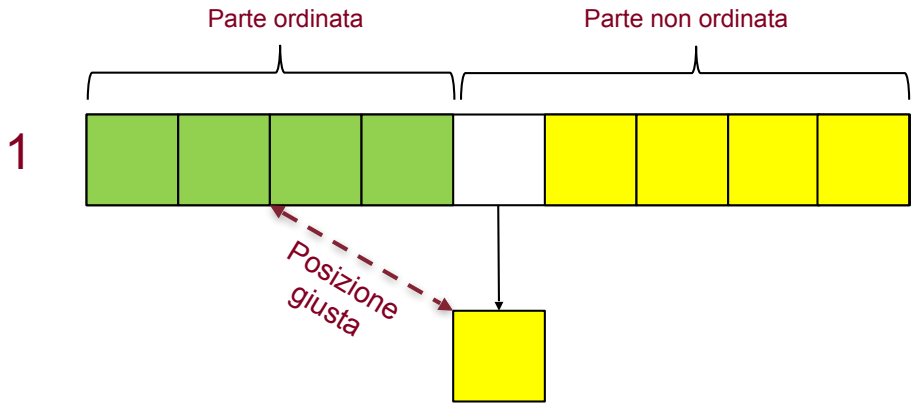
1. L'array è diviso in due parti: la parte ordinata e la parte non ordinata. Inizialmente, la parte ordinata contiene solo il primo elemento dell'array e la parte non ordinata contiene tutti gli altri elementi.

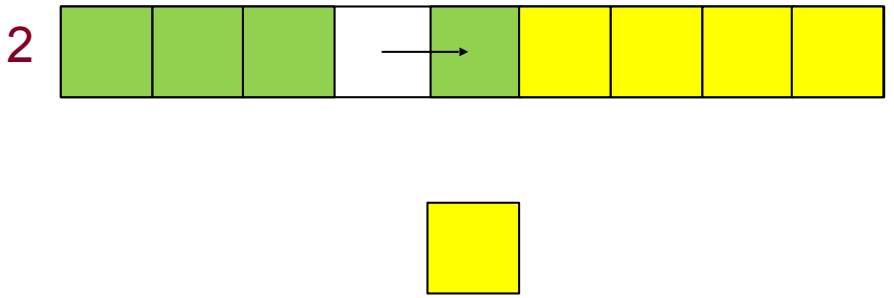
2. L'algoritmo prende il primo elemento della parte non ordinata e lo confronta con gli elementi della parte ordinata. Se l'elemento è minore di un elemento della parte ordinata, l'elemento viene spostato a destra fino a trovare la sua posizione corretta.

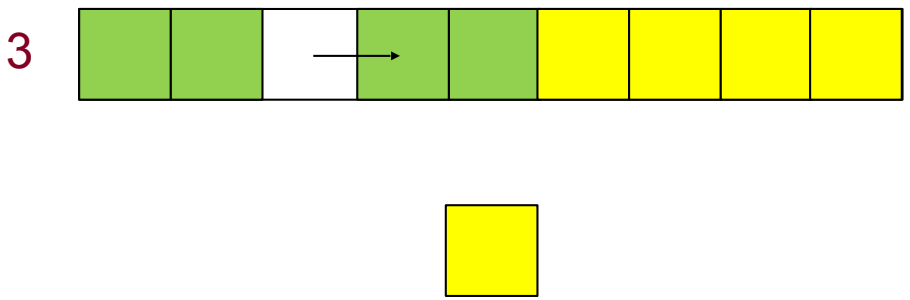
3. L'array ordinato viene esteso per includere l'elemento appena aggiunto.

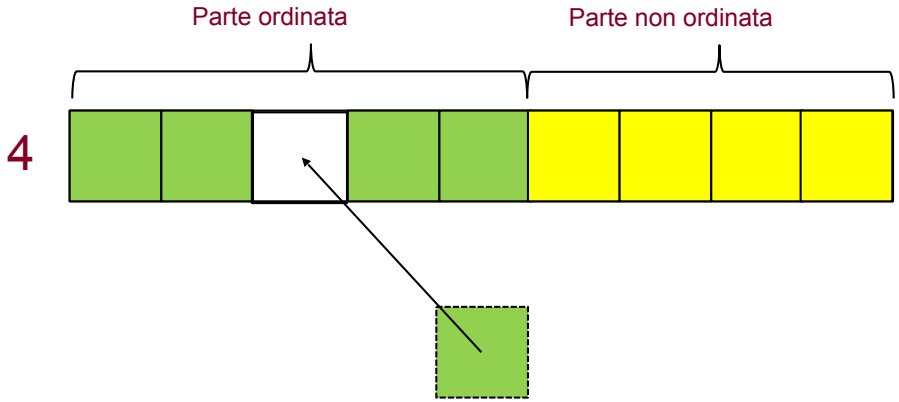
4. L'algoritmo continua ad eseguire questi passaggi fino a quando tutti gli elementi dell'array non sono stati ordinati.

Insertion Sort









```python
def InsertionSort(A):
    n = len(A)
    for i in range(1, n):
        x = A[i]
        # x è l'elemento da inserire nella parte ordinata
        j = i - 1
        # Sposta a destra gli elementi di A[:i] che
        # sono maggiori di x
        while j >= 0 and A[j] > x:
            A[j + 1] = A[j]
            j -= 1
        # Inserisci x nella posizione corretta
        A[j + 1] = x
```

```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
```

La complessità dell'algoritmo dipende dal numero di iterazioni eseguite dal *for* e dal *while* in esso annidato. Il *for* viene iterato esattamente n volte mentre all'i esima iterazione del *for* il *while* sarà iterato $O(i)$ volte.

- **Caso migliore:** si ha quando il il numero di iterazioni del *while* è $\Theta(1)$. Questo accade quando il vettore è già ordinato In questo caso, ogni elemento da esaminare è già al suo posto, quindi l'algoritmo non deve effettuare alcun spostamento. Alla prima iterazione, l'elemento 'A[1]' verrà confrontato con 'A[0]' ma non sarà necessario alcun spostamento, poiché 'A[1] $\geq$A[0]'. Lo stesso avviene per tutte le successive iterazioni: ogni elemento sarà maggiore o uguale al precedente, quindi non sarà necessario fare spostamenti. Il numero totale di iterazioni sarà in questo caso

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

```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
```

La complessità dell'algoritmo dipende dal numero di iterazioni eseguite dal *for* e dal *while* in esso annidato. Il *for* viene iterato esattamente n volte mentre all'i esima iterazione del *for* il *while* sarà iterato $O(i)$ volte.

- **Caso peggiore:** Il caso peggiore si verifica quando l'array è ordinato in ordine decrescente. In questo caso, ogni elemento dovrà essere spostato attraverso l'intero array per essere inserito nella sua posizione corretta. E quindi il *while* all'i-esima iterazione richiederà $\Theta(i)$. Ad esempio, alla prima iterazione, 'A[1]' dovrà essere spostato fino alla posizione di 'A[0]'. Alla seconda iterazione, 'A[2]' dovrà essere spostato fino alla posizione di 'A[1]', e così via. E quindi il *while* all'i-esima iterazione richiederà $\Theta(i)$. e per la complessità si avrà

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

La complessità del caso ottimo e quella del caso pessimo sono asintoticamente differenti, diventa quindi interessante valutare la complessià dell'algoritmo al caso medio:
In media, un elemento dovrà essere spostato a metà della sua distanza, possiamo dire che il numero medio di confronti per ciascun elemento sarà circa $\frac{i}{2}$. Il numero atteso di spostamenti sarà quindi $E = \sum_{i=1}^{n-1} \frac{i}{2} = \frac{n(n-1)}{4}$. Pertanto, nel caso medio, l'algoritmo ha una complessità $\Theta(n^2)$.

- Riassumendo: quest'algoritmo ha complessità $\Theta(n)$ nel caso migliore e complessità $\Theta(n^2)$ al caso medio e al caso pessimo.
- L'algoritmo ha il pregio di essere in-place e non usa quindi memoria aggiuntiva significativa.

Ordinamento a bolle (Bubble Sort)

L'algoritmo ordina un vettore confrontando coppie di elementi adiacenti e scambiandoli se sono nell'ordine sbagliato. Questo processo viene ripetuto fino a quando l'intero vettore è ordinato. L'algoritmo è composto da due cicli annidati:

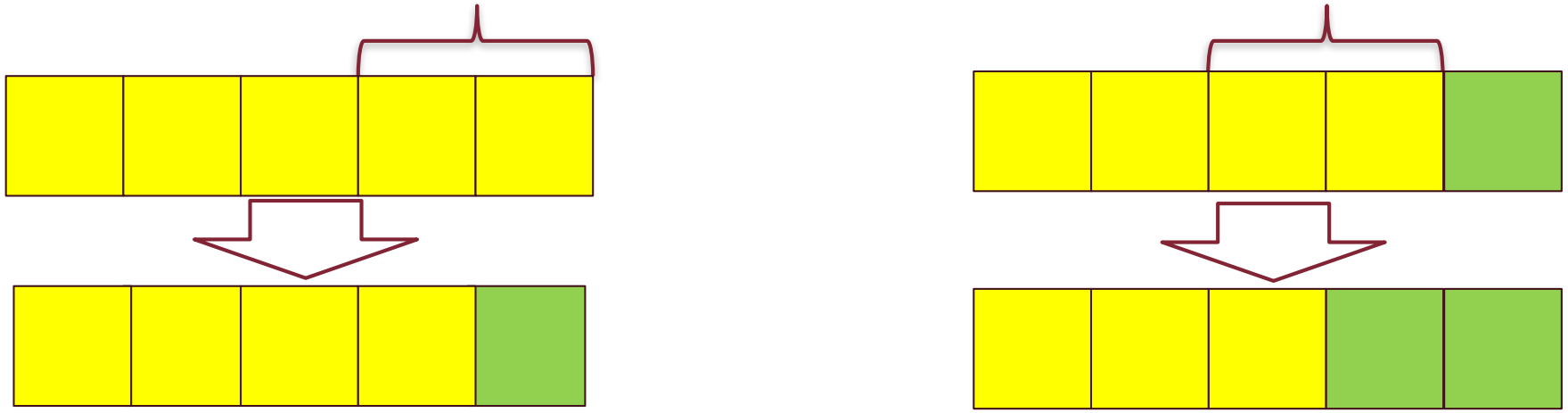
- Il ciclo esterno garantisce che all'i-esima iterazione l'elemento i esimo finisca nella corretta posizione (il ciclo viene iterato $n-1$ volte in quanto al termine l'elemento in prima posizione risulta automaticamente nella posizione corretta
- Il ciclo interno esegue i confronti tra coppie di elementi adiacenti e li scambia se sono nell'ordine sbagliato.

Ad ogni passaggio del ciclo interno, il valore maggiore tra gli elementi comparati "sale" verso la fine dell'array, mentre gli altri valori più piccoli si spostano verso l'inizio. Di conseguenza, dopo il primo passaggio, il massimo elemento sarà collocato nell'ultima posizione, dopo il secondo passaggio il secondo massimo sarà collocato nella penultima posizione, e così via. Dopo i iterazioni del for *esterno* gli ultimi i elementi dell'array sono collocati nella loro corretta posizione. Con ogni passaggio, il numero di confronti necessari diminuisce progressivamente, finché l'array non è completamente ordinato.

Bubble Sort

L'algoritmo funziona per fasi. Ogni fase ispeziona una dopo l'altra, da sinistra a destra, le coppie 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.



```
def BubbleSort(A):
    n = len(A)
    for i in range(n-1):
        # gli elementi adiacenti in A[:n-i-1] vengono
        # confrontati a coppie e se non sono nell'ardine
        # giusto vengono scambiati di posto
        for j in range(n-i-1):
            if A[j] > A[j+1]:
                A[j], A[j+1] = A[j+1], A[j]
```

```python
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]
```

La complessità dipende essenzialmente dai due cicli for annidati, il primo effettua $n-1$ iterazioni e, all'i-esima iterazione, il ciclo interno effettua $n-i-1$ iterazioni, ciascuna di costo $\Theta(1)$. Abbiamo dunque:

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

- In quest'algoritmo non c'è distinzione tra i costi del caso migliore e del caso peggiore. E l'algoritmo lavora in-place.

Nota che, se durante l'esecuzione del ciclo esterno, il ciclo interno non effettua alcuno scambio, significa che il vettore è già ordinato. In questo caso, possiamo ridurre il numero di confronti necessari. Per fare ciò, utilizziamo un flag scambi per segnalare se durante l'esecuzione del ciclo interno sono stati effettuati scambi. Ecco la versione migliorata dell'algoritmo:

```
def BubbleSort(A):
    n = len(A)
    for i in range(n - 1):
        scambi = False
        # scambi è un Flag per determinare se
        # sono stati effettuati scambi
        for j in range(n - i - 1):
            if A[j] > A[j + 1]:
                # Scambio degli elementi
                A[j], A[j + 1] = A[j + 1], A[j]
                scambi = True
        # Se non ci sono stati scambi, l'array è già ordinato
        if not scambi:
            return
```

```python
def BubbleSort(A):
    n = len(A)
    for i in range(n - 1):
        scambi = False
        for j in range(n - i - 1):
            if A[j] > A[j + 1]:
                A[j], A[j + 1] = A[j + 1], A[j]
                scambi = True
        if not scambi:
            return
```

Questa versione si comporta meglio della precedente nel caso migliore:

- **Caso migliore:** Quando il vettore è già ordinato. Durante la prima iterazione del ciclo più interno, vengono eseguiti $n-1$ confronti e 0 scambi, il che causa la terminazione dell'algoritmo. La complessità è in questo caso $O(n)$.
- **Caso peggiore:** Quando il vettore è ordinato in ordine decrescente. In questo caso l'algoritmo dovrà effettuare $n-1$ passate. Ogni iterazione del ciclo più interno esegue almeno uno scambio e quindi, il numero totale di iterazioni è (esattamente come nell'altro caso):

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

```python
def BubbleSort(A):
    n = len(A)
    for i in range(n - 1):
        scambi = False
        for j in range(n - i - 1):
            if A[j] > A[j + 1]:
                A[j], A[j + 1] = A[j + 1], A[j]
                scambi = True
        if not scambi:
            return
```

Riassumendo:

- **Caso migliore:** $\Theta(n)$ (quando l'array è già ordinato). In questo caso, dopo la prima iterazione del ciclo esterno, l'algoritmo termina.
- **Caso peggiore:** $\Theta(n^2)$ (quando l'array è ordinato in ordine decrescente). In questo caso tutte le $n-1$ iterazioni del ciclo esterno vengono eseguite.

Poiché caso ottimo e caso pessimo non coincidono, vale la pena di esaminare il caso medio. L'analisi di quest'ultimo è piuttosto complessa, ma alla fine si conclude che anche al caso medio la complessità è $\Theta(n^2)$.

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 per semplicità assumiamo che i numeri da ordinare siano tutti distinti.

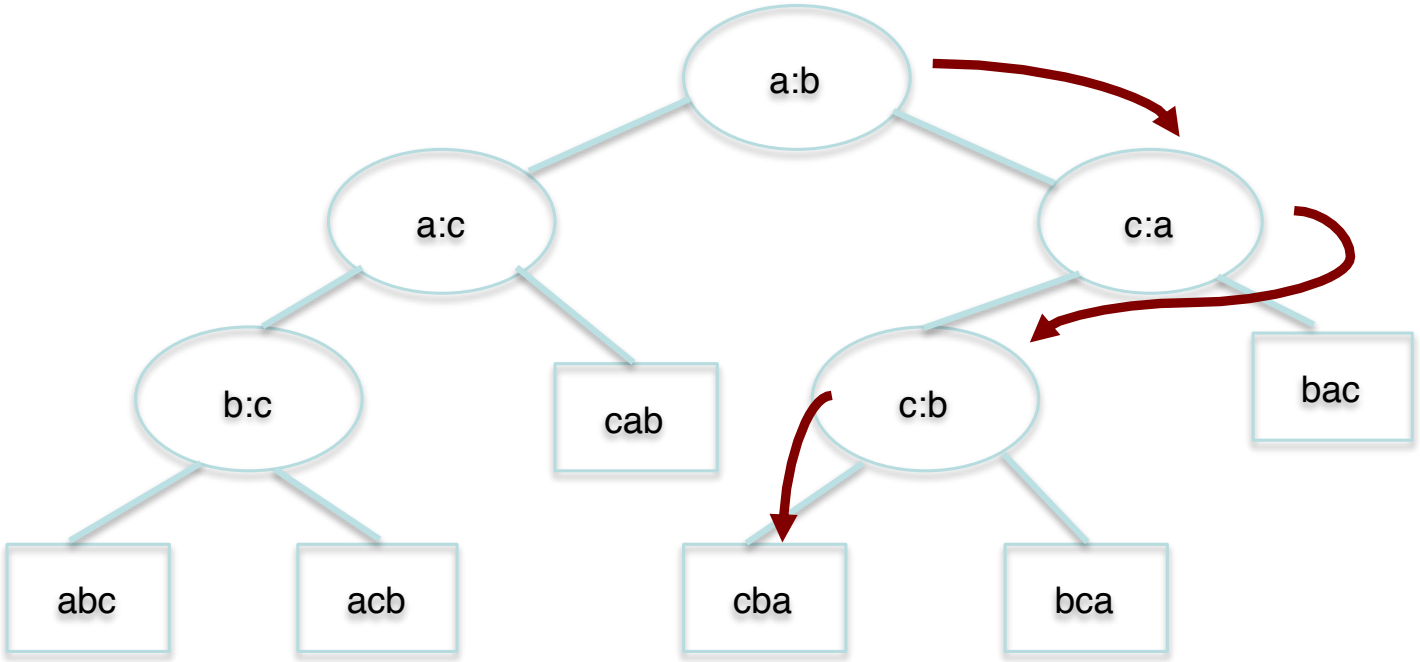
L'albero di decisione relativo a un qualunque algoritmo di ordinamento basato su confronti ha queste proprietà:

· 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;

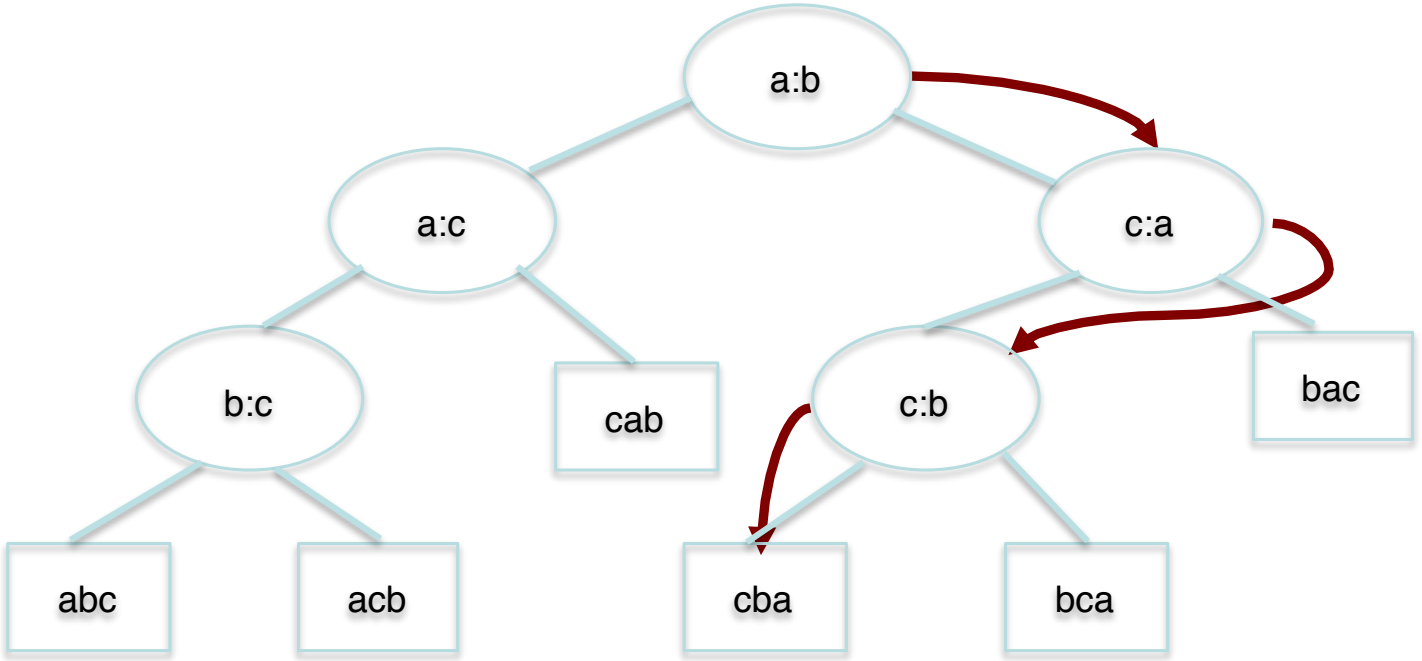
· ogni foglia rappresenta una possibile soluzione del problema, la quale è una specifica permutazione della sequenza in ingresso.

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)

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

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

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

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

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

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 cresce progressivamente sia quella a sinistra. La complessità asintotica dell'algoritmo deve rimanere $O(n^2)$.

2. Un algoritmo di ordinamento si dice **stabile** se mantiene l'ordine relativo degli elementi con valori uguali nell'input. In altre parole, se due elementi a e b sono uguali secondo il criterio di ordinamento, allora a apparirà prima di b nel risultato finale se nell'input a compariva prima di b.

   La stabilità è particolarmente utile in contesti dove si desidera mantenere informazioni aggiuntive che sono collegate agli oggetti da ordinare. Ad esempio, se si ordina una lista di persone per età, ma si desidera che le persone con la stessa età restino ordinate secondo un altro criterio (ad esempio, per nome), un algoritmo stabile manterrà l'ordine originale per quelle persone che hanno lo stesso valore di età.

   Indicare, tra i seguenti algoritmi di ordinamento, quale di essi è stabile e spiegare il motivo della risposta:

   - Bubble Sort
   - Selection Sort
   - Insertion Sort

3. 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 è ordinata in modo decrescente?

4. 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?

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

6. Data una lista di n elementi, progettare 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.

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 $\Theta(n^2)$.

·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?

·3 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 è ordinata in modo decrescente?

·4 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?

·5 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.

·6 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.