TECNICO SUPERIORE
WEB DEVELOPER
FULL STACK

#6 - Tipologie di input e algoritmi di ordinamento

Valutare gli algoritmi in base all'input

In alcuni casi, gli algoritmi si comportano diversamente a seconda delle caratteristiche dell'input (come è organizzato l'input).

Conoscere in anticipo tali caratteristiche permette di scegliere il miglior algoritmo per una particolare situazione.

Tipologia di analisi

Analisi del caso pessimo (la più importante)

- Il tempo di esecuzione nel caso peggiore è un limite superiore al tempo di esecuzione per qualsiasi input
- Per alcuni algoritmi, il caso peggiore si verifica molto spesso (es.: ricerca di dati non presenti in un database)

Analisi del caso ottimo

- Può avere senso se si hanno informazioni particolari sull'input

Analisi del caso medio

- Difficile in alcuni casi: cosa si intende per "medio"?
- Algoritmi probabilistici (che non ci interessano)

Definizioni di base

Un algoritmo di ordinamento è un algoritmo che:

prende in input una sequenza $A = a_1, a_2, \ldots, a_n$ di n valori

produce in output una sequenza $B = b_1, b_2, \ldots, b_n$ che sia una permutazione di A e tale per cui $b_1 \leq b_2 \leq \ldots \leq b_n$.

Ce ne sono diversi: semplici, complicati, ricorsivi, iterativi, basati su confronti o meno, che usano strutture di appoggio o senza altri supporti oltre all'array (in-place).

E poi ci sono quelli brutti.

Name	Best	Average	Worst	Memory	Stable
Quicksort	n	$n \log n$	n^2	$\log n$	No
Merge sort	$n \log n$	$n \log n$	$n \log n$	n	Yes
In-place merge sort	—	—	$n \log^2 n$	1	Yes
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No
Insertion sort	n	n^2	n^2	1	Yes
Block sort	n	$n \log n$	$n \log n$	1	Yes
Quadsort	n	$n \log n$	$n \log n$	n	Yes
Timsort	n	$n \log n$	$n \log n$	n	Yes
Selection sort	n^2	n^2	n^2	1	No
Cubesort	n	$n \log n$	$n \log n$	n	Yes
Shell sort	$n \log n$	$n^{4/3}$	$n^{4/3}$	1	No
Bubble sort	n	n^2	n^2	1	Yes
Binary tree sort	$n \log n$	$n \log n$	$n \log n$(balanced)	n	Yes
Cycle sort	n^2	n^2	n^2	1	No
Library sort	n	$n \log n$	n^2	n	Yes
Patience sorting	n	—	$n \log n$	n	No
Smoothsort	n	$n \log n$	$n \log n$	1	No
Strand sort	n	n^2	n^2	n	Yes
Tournament sort	$n \log n$	$n \log n$	$n \log n$	n[12]	No
Cocktail sort	n	n^2	n^2	1	Yes
Comb sort	$n \log n$	n^2	n^2	1	No
Gnome sort	n	n^2	n^2	1	Yes
UnShuffle Sort[13]	n	kn	kn	n	No
Franceschini's method[14]	—	$n \log n$	$n \log n$	1	Yes
Odd–even sort	n	n^2	n^2	1	Yes

Dobbiamo saperli tutti?

No.

Però è importante conoscere gli approcci più noti e meglio performanti; inoltre, in caso di necessità, meglio saperne almeno uno di quelli più intuitivi.

Proprietà degli algoritmi di ordinamento

- **Stabilità**: un algoritmo di ordinamento è *stabile* se preserva l'ordine iniziale tra due elementi con la stessa chiave
  - Ad esempio: ordinamento per nome e per cognome
- **Ordinamento sul posto** (o *in place*): si tratta di un algoritmo che non crea copie dell'input per generare la sequenza ordinata
- **Adattatività**: un algoritmo di ordinamento è adattativo se trae vantaggio dagli elementi già ordinati

Partiamo con un gioco

Immaginate che ad ognuno di voi siano state date 5 carte.

L’obiettivo è rimetterle in ordine dopo averle mescolate, senza però muovere più di una carta alla volta.

Lo sforzo che vi chiedo di compiere è quello di analizzare il vostro ragionamento, cercando di capire con quale criterio avete riordinato le carte.


A LITTLE LONGER
THAN A FEW
MINUTES LATER

Gli algoritmi di ordinamento più intuitivi

Tra i tanti, gli algoritmi che andremo ad analizzare sono:

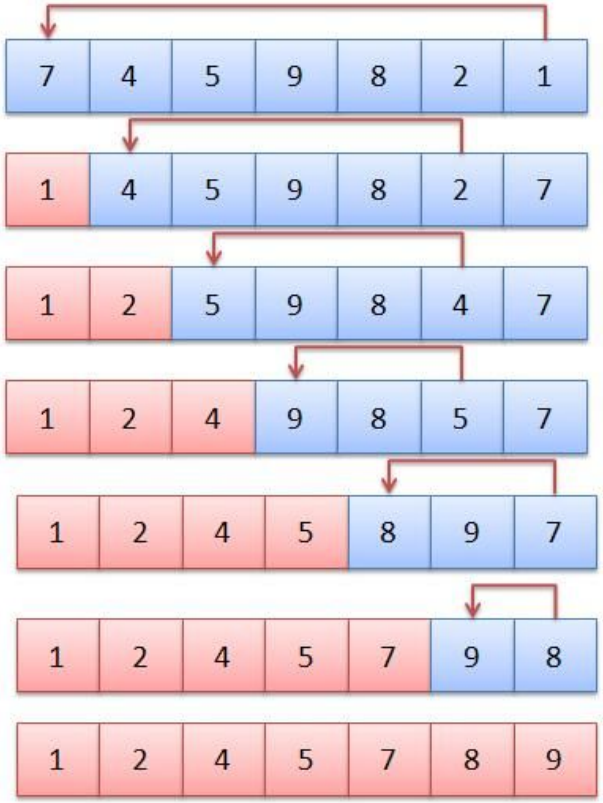
- Selection Sort
- Insertion Sort
- MergeSort
- QuickSort

Selection Sort

L'algoritmo seleziona di volta in volta il numero minore nella sequenza ancora da ordinare e lo sposta nella sequenza ordinata.

La sequenza viene quindi divisa in due parti: la sottosequenza ordinata, che occupa le prime posizioni dell'array, e la sottosequenza da ordinare, che costituisce la parte restante dell'array su cui ri-applicare l’algoritmo.

Selection Sort in un'immagine



Ma prima...

Dobbiamo velocemente aggiustare la funzione $\min()$

```
int min(int[] S, int n)
int min = S[1]
for i = 2 to n do
    if S[i] < min then
        min = S[i]
return min
```

```
int min(ITEM[] A, int i, int n)
% Posizione del minimo parziale
int min = i
for j = i + 1 to n do
    if A[j] < A[min] then
        % Nuovo minimo parziale
        min = j
return min
```

Selection Sort, lo pseudocodice

```
SelectionSort(ITEM[] A, int n)
for i = 1 to n − 1 do
    int min = min(A, i, n)
    A[i] ↔ A[min]
```

```
int min(ITEM[] A, int i, int n)
% Posizione del minimo parziale
int min = i
for j = i + 1 to n do
    if A[j] < A[min] then
        % Nuovo minimo parziale
        min = j
return min
```

Qual è la complessità:

- Nel caso pessimo?
- Nel caso medio?
- Nel caso ottimo?

Selection Sort, lo pseudocodice

```
SelectionSort(ITEM[] A, int n)
for i = 1 to n − 1 do
    int min = min(A, i, n)
    A[i] ↔ A[min]
```

```
int min(ITEM[] A, int i, int n)
% Posizione del minimo parziale
int min = i
for j = i + 1 to n do
    if A[j] < A[min] then
        % Nuovo minimo parziale
        min = j
return min
```

Qual è la complessità:

Alla prima chiamata eseguo `n` confronti, alla seconda `n-1`, alla terza `n-2`, ecc.

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

$$= \frac{n(n+1)}{2} - 1 = O(n^2)$$

Analisi dei casi

In questo caso specifico non esiste un caso “ottimo”, “pessimo” o “medio”, perché in qualunque caso l’algoritmo scorre sempre tutta la sequenza $O(n^2)$ volte.

Provate a testare l’algoritmo con questi input:

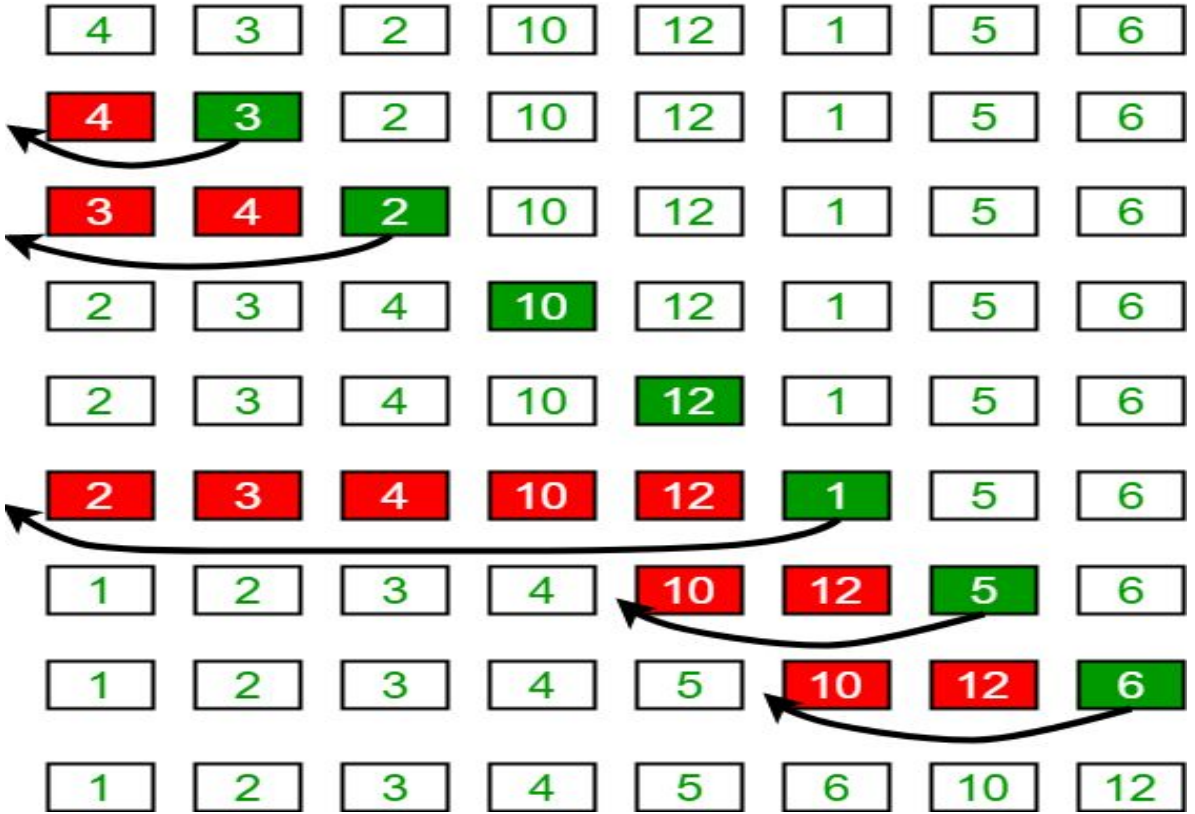
- `1,2,3,4,5,6,7,8`
- `8,7,6,5,4,3,2,1`
- `5,4,7,2,1,8,3,6`

Il comportamento è identico.

Insertion Sort

E’ un algoritmo efficiente per ordinare piccoli insiemi di elementi e si basa sul principio di ordinamento di una "mano" di carte da gioco: ogni carta viene presa dalla posizione in cui si trova per essere spostata dove sarà maggiore di tutte le carte precedenti (o all’inizio).

Insertion Sort in un'immagine



Insertion Sort, lo pseudocodice

```
insertionSort(ITEM[] A, int n)
for i = 2 to n do
    ITEM temp = A[i]
    int j = i
    while j > 1 and A[j − 1] > temp do
        A[j] = A[j − 1]
        j = j − 1
    A[j] = temp
```

L'idea è che, per ogni elemento, memorizzo il valore A[i] e, se necessario, "sposto" tutti valori maggiori di A[i] verso destra, creando uno spazio per inserire A[i] nella corretta posizione.

Insertion Sort, lo pseudocodice

```
insertionSort(ITEM[] A, int n)
for i = 2 to n do
    ITEM temp = A[i]
    int j = i
    while j > 1 and A[j − 1] > temp do
        A[j] = A[j − 1]
        j = j − 1
    A[j] = temp
```

Qual è la complessità:

- Nel caso ottimo?
- Nel caso pessimo?
- Nel caso medio?

Insertion Sort, lo pseudocodice

```
insertionSort(ITEM[] A, int n)
for i = 2 to n do
    ITEM temp = A[i]
    int j = i
    while j > 1 and A[j − 1] > temp do
        A[j] = A[j − 1]
        j = j − 1
    A[j] = temp
```

Qual è la complessità:

- Nel caso ottimo?

Nel caso ottimo (quando l'array è già ordinato) il ciclo while non viene mai eseguito (`A[j-1]>temp` è sempre falsa), quindi devo solo scorrere gli elementi.

$$T(n) = 3c * n = O(n)$$

Insertion Sort, lo pseudocodice

```
insertionSort(ITEM[] A, int n)
for i = 2 to n do
    ITEM temp = A[i]
    int j = i
    while j > 1 and A[j − 1] > temp do
        A[j] = A[j − 1]
        j = j − 1
    A[j] = temp
```

Qual è la complessità:

- Nel caso pessimo?

Nel caso pessimo (quando l'array è ordinato in ordine inverso) il ciclo while viene eseguito prima (n-1) volte, poi (n-2), ecc.

$$T(n) = O(n^2)$$

Insertion Sort, lo pseudocodice

```
insertionSort(ITEM[] A, int n)
for i = 2 to n do
    ITEM temp = A[i]
    int j = i
    while j > 1 and A[j − 1] > temp do
        A[j] = A[j − 1]
        j = j − 1
    A[j] = temp
```

Qual è la complessità:

- Nel caso medio?

Nel caso medio, senza entrare nei dettagli della dimostrazione, ci saranno elementi che devo spostare da un estremo all'altro del vettore, quindi "in media".

$$T(n) = O(n^2)$$

Insertion VS Selection

Insertion sort, nel caso ottimo, si comporta meglio di Selection sort, che è $O(n^2)$ sempre.

Questo è vero anche quando l'input è "parzialmente ordinato", ad es. in una sequenza come [1,2,5, 4,6,7,8].

L'unico scambio da effettuare è quello tra 4 e 5, quindi Insertion sort sarà più rapido di Selection.

Nel caso pessimo, si comportano in modo uguale (dal punto di vista della complessità).

Esercizi

- Per il puro spirito di esercizio, implementateli!