

# Algoritmi e strutture dati

## Algoritmi di ordinamento

Alberto Montresor

Università di Trento

2020/04/20

This work is licensed under a Creative Commons  
Attribution-ShareAlike 4.0 International License.



## Sommario - Algoritmi di ordinamento

- SelectionSort -  $\Theta(n^2)$
- InsertionSort -  $\Omega(n)$ ,  $O(n^2)$
- ShellSort -  $\Omega(n)$ ,  $O(n^{3/2})$
- MergeSort -  $\Theta(n \log n)$
- HeapSort -  $\Theta(n \log n)$
- QuickSort -  $\Omega(n \log n)$ ,  $O(n^2)$

## Sommario - Problema dell'ordinamento

- **Tutti questi algoritmi sono basati su confronti**
  - Le decisioni sull'ordinamento vengono prese in base al confronto ( $<$ ,  $=$ ,  $>$ ) fra due valori
- **Algoritmi migliori:  $O(n \log n)$** 
  - InsertionSort e ShellSort sono più veloci solo in casi speciali

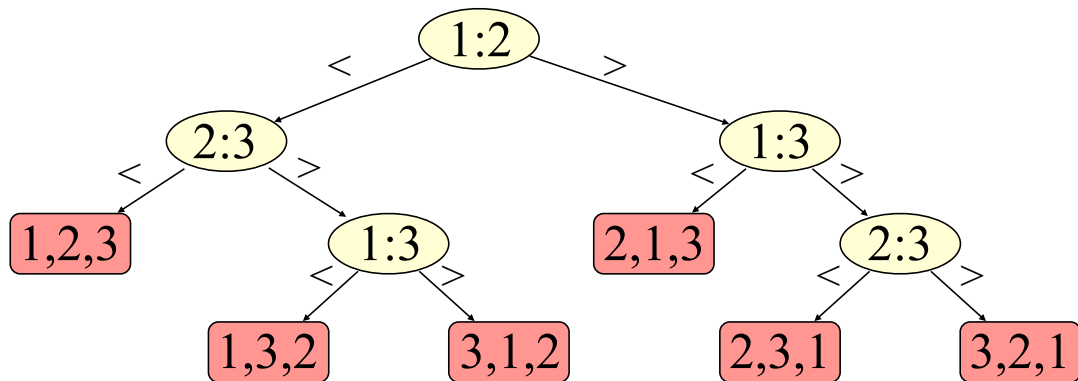
## Problema dell'ordinamento - Limite inferiore

E' possibile dimostrare che qualunque algoritmo di ordinamento **basato su confronti** ha una complessità  $\Omega(n \log n)$ .

## Assunzioni

- Consideriamo un qualunque algoritmo  $A$  basato su confronti
- Assumiamo che tutti i valori siano distinti (no perdita di generalità)
- L'algoritmo  $A$  può essere rappresentato tramite un **albero di decisione**, un albero binario che rappresenta i confronti fra gli elementi

## Albero di decisione



# Albero di decisione

## Proprietà

- **Cammino radice-foglia in un albero di decisione:** sequenza di confronti eseguiti dall'algoritmo corrispondente
- **Altezza dell'albero di decisione:** numero confronti eseguiti dall'algoritmo corrispondente nel caso pessimo

Si considerino tutti gli alberi di decisioni ottenibili da algoritmi di ordinamento basati su confronti

# Limite inferiore per l'ordinamento

## Lemma 1

Un albero di decisione per l'ordinamento di  $n$  elementi contiene almeno  $n!$  foglie

## Lemma 2

Sia  $T$  un albero binario in cui ogni nodo interno ha esattamente 2 figli e sia  $k$  il numero delle sue foglie. L'altezza dell'albero è almeno  $\log k$  - ovvero  $\Omega(\log k)$ .

## Teorema

Il numero di confronti necessari per ordinare  $n$  elementi nel caso peggiore è  $\Omega(n \log n)$

# Spaghetti Sort

## Algoritmo Spaghetti Sort – $O(n)$

- 1 Prendi  $n$  spaghetti
- 2 Taglia lo spaghetti  $i$ -esimo in modo proporzionale all' $i$ -esimo valore da ordinare
- 3 Con la mano, afferra gli  $n$  spaghetti e appoggiali verticalmente sul tavolo
- 4 Prendi il più lungo, misuralo e metti il valore corrispondente in fondo al vettore da ordinare
- 5 Ripeti (4) fino a quando non hai terminato gli spaghetti

# Counting Sort

## Assunzione

- I numeri da ordinare sono compresi in un intervallo  $[1 \dots k]$

## Come funziona

- Costruisce un array  $B[1 \dots k]$  che conta il numero di volte che un valore compreso in  $[1 \dots k]$  compare in  $A$
- Ricolloca i valori così ottenuti nel vettore da ordinare  $A$

## Miglioramenti

- L'intervallo non deve necessariamente iniziare in 1 e finire in  $k$ ; qualunque intervallo di cui conosciamo gli estremi può essere utilizzato nel Counting Sort.



# Counting Sort

---

```
countingSort(int[] A, int n, int k)
```

---

```
int[] B = new int[1...k]
```

```
for i = 1 to k do
```

```
    B[i] = 0
```

```
for j = 1 to n do
```

```
    B[A[j]] = B[A[j]] + 1
```

```
j = 1
```

```
for i = 1 to k do
```

```
    while B[i] > 0 do
```

```
        A[j] = i
```

```
        j = j + 1
```

```
        B[i] = B[i] - 1
```

```
    
```

---

# Counting Sort

## Complessità di Counting Sort

- $O(n + k)$
- Se  $k$  è  $O(n)$ , allora la complessità di Counting Sort è  $O(n)$

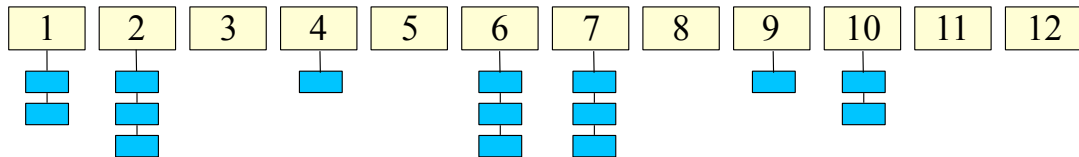
## Counting Sort e limiti inferiore per l'ordinamento

- Counting Sort non è basato su confronti
- Abbiamo cambiato le condizioni di base
- Se  $k$  è  $O(n^3)$ , questo algoritmo è peggiore di tutti quelli visti finora

# Pigeonhole Sort

## Casellario

- Cosa succede se i valori non sono numeri interi, ma record associati ad una chiave da ordinare?
- Non possiamo usare counting
- Ma possiamo usare liste concatenate!



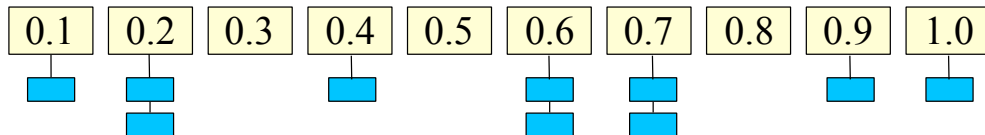
# Bucket Sort

## Ipotesi sull'input

- Valori reali uniformemente distribuiti nell'intervallo  $[0, 1)$
- Qualunque insieme di valori distribuiti uniformemente può essere normalizzato nell'intervallo  $[0, 1)$  in tempo lineare

## Idea

- Dividere l'intervallo in  $n$  sottointervalli di dimensione  $1/n$ , detti **bucket**, e poi distribuire gli  $n$  numeri nei bucket
- Per l'ipotesi di uniformità, il numero atteso di valori nei bucket è 1
- Possono essere ordinati con Insertion Sort



# Proprietà degli algoritmi di ordinamento

## Stabilità

Un algoritmo di ordinamento è **stabile** se preserva l'ordine iniziale tra due elementi con la stessa chiave

## Domande

- Quali dei seguenti algoritmi sono stabili? Insertion Sort, Merge Sort, Heap Sort, Quick Sort, Pigeonhole Sort
- Come si può rendere un qualunque algoritmo stabile?

# Proprietà degli algoritmi di ordinamento

## Stabilità

Un algoritmo di ordinamento è **stabile** se preserva l'ordine iniziale tra due elementi con la stessa chiave

## Domande

- Quali dei seguenti algoritmi sono stabili? Insertion Sort, Merge Sort, Heap Sort, Quick Sort, Pigeonhole Sort
- Come si può rendere un qualunque algoritmo stabile?

## Risposte

- Stabili: Insertion Sort, Merge Sort, Pigeonhole Sort
- Basta usare come chiave di ordinamento la coppia (chiave, posizione iniziale)

# Riassunto ordinamento

## Insertion Sort

$\Omega(n)$ ,  $O(n^2)$ , stabile, sul posto, iterativo. Adatto per piccoli valori, sequenze quasi ordinate.

## Merge Sort

$\Theta(n \log n)$ , stabile, richiede  $O(n)$  spazio aggiuntivo, ricorsivo (richiede  $O(\log n)$  spazio nello stack). Buona performance in cache, buona parallelizzazione.

## Heap Sort

$\Theta(n \log n)$ , non stabile, sul posto, iterativo. Cattiva performance in cache, cattiva parallelizzazione. Preferito in sistemi embedded.

# Riassunto ordinamento

## Quick Sort

$O(n \log n)$  in media,  $O(n^2)$  nel caso peggiore, non stabile, ricorsivo (richiede  $O(\log n)$  spazio nello stack). Buona performance in cache, buona parallelizzazione, buoni fattori moltiplicativi.

## Counting Sort

$\Theta(n + k)$ , richiede  $O(k)$  memoria aggiuntiva, iterativo. Molto veloce quando  $k = O(n)$

## Pigeonhole Sort

$\Theta(n + k)$ , stabile, richiede  $O(n + k)$  memoria aggiuntiva, iterativo. Molto veloce quando  $k = O(n)$



# Riassunto ordinamento

## Bucket Sort

$O(n)$  nel caso i valori siano distribuiti uniformemente, stabile, richiede  $O(n)$  spazio aggiuntivo

## Shell Sort

$O(n\sqrt{n})$ , stabile, adatto per piccoli valori, sequenze quasi ordinate.

## Algoritmi di ordinamento: un'esperienza psichedelica

<https://www.youtube.com/watch?v=kPRA0W1kECg>

# Reality check

## Tim Sort (<https://en.wikipedia.org/wiki/Timsort>)

- Algoritmo ibrido, basato su Merge Sort e Insertion Sort
- Cerca sequenze consecutive (run) già ordinate
- Complessità:
  - $\Omega(n)$  (sequenze già ordinate)
  - $O(n \log n)$  nel caso pessimo

## Utilizzazione

- Python
- Java 7
- Gnu Octave