

# Algoritmi e Strutture Dati

## Divide-et-impera

Alberto Montresor and Davide Rossi

Università di Bologna

12 dicembre 2024

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



# Sommario

- 1 Approcci algoritmici
- 2 Divide-et-impera
- 3 Quicksort

# Risoluzione problemi

## Dato un problema

- Non ci sono "ricette generali" per risolverlo in modo efficiente
- Tuttavia, è possibile evidenziare quattro fasi
  - Classificazione del problema
  - Caratterizzazione della soluzione
  - Tecnica di progetto
  - Utilizzo di strutture dati
- Queste fasi non sono necessariamente sequenziali

# Classificazione dei problemi

## Problemi decisionali

- Il dato di ingresso soddisfa una certa proprietà?
- Soluzione: risposta sì/no
- Esempio: Stabilire se un grafo è connesso

## Problemi di ricerca

- Spazio di ricerca: insieme di "soluzioni" possibili
- Soluzione ammissibile: soluzione che rispetta certi vincoli
- Esempio: posizione di una sottostringa in una stringa

# Classificazione dei problemi

## Problemi di ottimizzazione

- Ogni soluzione è associata ad una funzione di costo
- Vogliamo trovare la soluzione di costo minimo
- Esempio: cammino più breve fra due nodi

# Definizione matematica del problema

È fondamentale definire bene il problema in modo formale

- Spesso la formulazione è banale...
- ... ma può suggerire una prima idea di soluzione
- Esempio: Data una sequenza di  $n$  elementi, una permutazione ordinata è data dal minimo seguito da una permutazione ordinata dei restanti  $n - 1$  elementi (Selection Sort)

La definizione matematica può suggerire una possibile tecnica

- Sottostruttura ottima  $\rightarrow$  Programmazione dinamica
- Proprietà greedy  $\rightarrow$  Tecnica greedy

# Tecniche di soluzione problemi

## Divide-et-impera

- Un problema viene suddiviso in sotto-problemi indipendenti, che vengono risolti ricorsivamente (top-down)
- Ambito: problemi di decisione, ricerca

## Programmazione dinamica

- La soluzione viene costruita (bottom-up) a partire da un insieme di sotto-problemi potenzialmente ripetuti
- Ambito: problemi di ottimizzazione

## Memoization (o annotazione)

- Versione top-down della programmazione dinamica

# Tecniche di soluzione problemi

## Tecnica greedy

- Approccio "ingordo": si fa sempre la scelta localmente ottima

## Backtrack

- Procediamo per "tentativi", tornando ogni tanto sui nostri passi

## Ricerca locale

- La soluzione ottima viene trovata "migliorando" via via soluzioni esistenti

## Algoritmi probabilistici

- Meglio scegliere con giudizio (ma in maniera costosa) o scegliere a caso ("gratuitamente")



# Sommario

- 1 Approcci algoritmici
- 2 Divide-et-impera**
- 3 Quicksort

# Divide-et-impera

## Tre fasi

- **Divide:** Dividi il problema in sotto-problemi più piccoli e indipendenti
- **Impera:** Risolvi i sotto-problemi ricorsivamente
- **Combina:** "unisci" le soluzioni dei sottoproblemi

Non esiste una ricetta "unica" per divide-et-impera

- Merge Sort: "divide" banale, "combina" complesso
- Quicksort: "divide" complesso, niente fase di "combina"
- È necessario uno sforzo creativo

# Sommario

- 1 Approcci algoritmici
- 2 Divide-et-impera
- 3 Quicksort

# Quicksort (Hoare, 1961)

## Algoritmo di ordinamento basato su divide-et-impera

- Caso medio:  $O(n \log n)$
- Caso pessimo:  $O(n^2)$

## Caso medio vs caso pessimo

- Il fattore costante di Quicksort è migliore di Merge Sort
- "In-place": non utilizza memoria aggiuntiva
- Tecniche "euristiche" per evitare il caso pessimo
- Quindi spesso è preferito ad altri algoritmi

R. Sedgwick, "*Implementing Quicksort Programs*". Communications of the ACM, 21(10):847-857, 1978. <http://portal.acm.org/citation.cfm?id=359631>

# Quicksort

## Input

- Vettore  $A[1 \dots n]$ ,
- Indici  $lo, hi$  tali che  $1 \leq lo \leq hi \leq n$

## Divide

- Sceglie un valore  $p \in A[lo \dots hi]$  detto **perno** (**pivot**)
- Sposta gli elementi del vettore  $A[lo \dots hi]$  in modo che:

13	14	15	12	20	27	29	30	21	25	28
----	----	----	----	----	----	----	----	----	----	----

$$\begin{array}{ccc}
 A[lo \dots j-1] & j & A[j+1 \dots hi] \\
 A[i] < A[j] & p = A[j] & A[j] < A[i]
 \end{array}$$

- L'indice  $j$  del perno va calcolato opportunamente

# Quicksort

## Impera

Ordina i due sottovettori  $A[lo \dots j - 1]$  e  $A[j + 1 \dots hi]$  richiamando ricorsivamente Quicksort

## Combina

Non fa nulla: infatti,

- il primo sottovettore,
- $A[j]$ ,
- il secondo sottovettore

formano già un vettore ordinato

# Quicksort – pivot()

---

```
int pivot(ITEM[] A, int lo, int hi)
```

---

```
int pivotIndex = lo
```

```
for i = lo + 1 to hi do
```

```
    if  $A[i] < A[pivotIndex]$  then
```

```
        pivotIndex = pivotIndex + 1
```

```
        swap(A, i, pivotIndex)
```

```
swap(A, lo, pivotIndex)
```

```
return pivotIndex
```

---



---

```
swap(ITEM[] A, int i, int j)
```

---

```
ITEM temp = A[i]
```

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

```
A[j] = temp
```

---

# Funzionamento pivot()

i	20	14	28	29	15	27	12	30	21	25	13
j											

$$A[i] \geq x$$

i	20	14	28	29	15	27	12	30	21	25	13
j											

$$A[i] < x: \quad j \leftarrow j+1, A[i] \leftrightarrow A[j]$$

i	20	14	28	29	15	27	12	30	21	25	13
j											

$$A[i] \geq x$$

i	20	14	28	29	15	27	12	30	21	25	13
j											

$$A[i] \geq x$$

i	20	14	28	29	15	27	12	30	21	25	13
j											

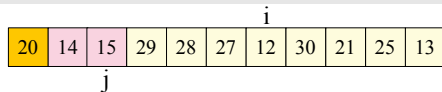
$$A[i] < x: \quad j \leftarrow j+1, A[i] \leftrightarrow A[j]$$

i	20	14	15	29	28	27	12	30	21	25	13
j											

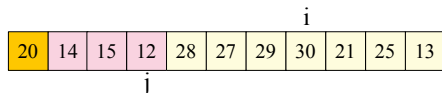
$$A[i] \geq x$$



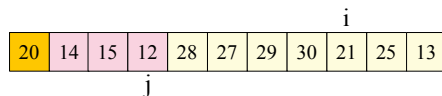
# Funzionamento pivot()



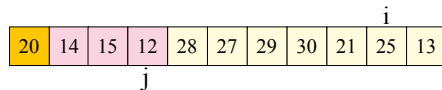
$$A[i] < x: \quad j \leftarrow j+1, A[i] \leftrightarrow A[j]$$



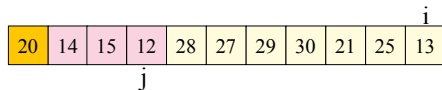
$$A[i] \geq x$$



$$A[i] \geq x$$

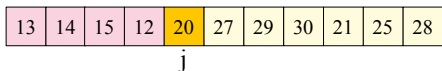


$$A[i] \geq x$$



$$A[i] < x: \quad j \leftarrow j+1, A[i] \leftrightarrow A[j]$$

$$A[lo] \leftarrow A[j]; A[j] \leftarrow x$$



# Hoare Partition Scheme

---

```
int pivot(ITEM[ ] A, int lo, int hi)
```

---

```
pivotValue = A[lo]
```

```
left = lo + 1
```

```
right = hi
```

```
while left <= right do
```

```
    while left <= right and A[left] < pivotValue do
```

```
        left = left + 1
```

```
    while left <= right and A[right] > pivotValue do
```

```
        right = right - 1
```

```
    if left < right then
```

```
        swap(A, left, right)
```

```
        left = left + 1
```

```
        right = right - 1
```

```
swap(A, low, right)
```

```
return right
```

---

# Quicksort – Procedura principale

---

```
QuickSort(ITEM[] A, int lo, int hi)
```

---

```
if lo < hi then
```

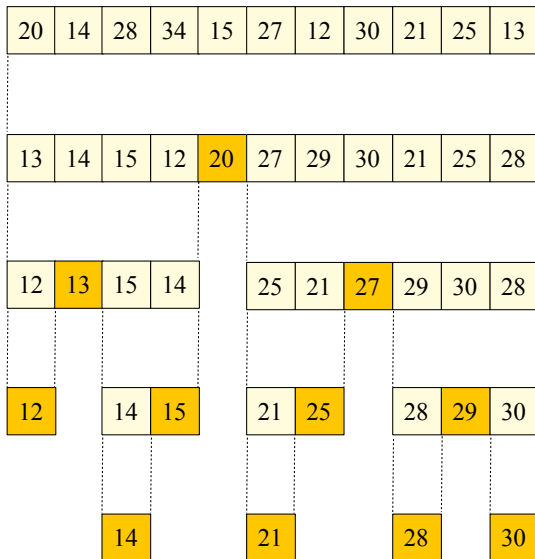
```
    int j = pivot(A, lo, hi)
```

```
    QuickSort(A, lo, j - 1)
```

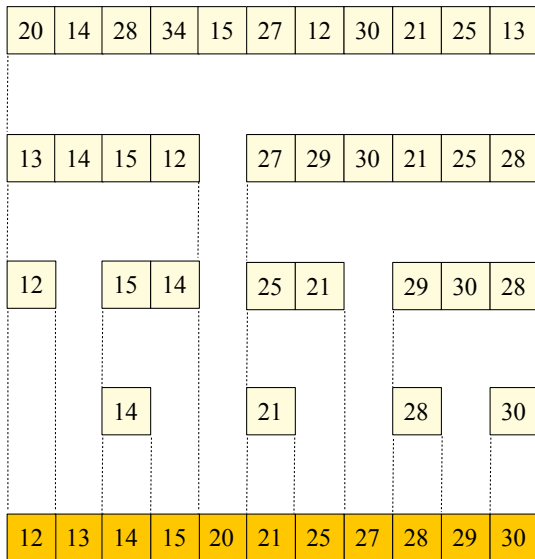
```
    QuickSort(A, j + 1, hi)
```

---

# Svolgimento ricorsione



# Svolgimento ricorsione



# Quicksort: Complessità computazionale

Costo di `pivot()`?

# Quicksort: Complessità computazionale

Costo di pivot()?

- $\Theta(n)$

Costo Quicksort: caso pessimo?

# Quicksort: Complessità computazionale

Costo di `pivot()`?

- $\Theta(n)$

Costo Quicksort: caso pessimo?

- Il vettore di dimensione  $n$  viene diviso in due sottovettori di dimensione 0 e  $n - 1$
- $T(n) = T(n - 1) + T(0) + \Theta(n) = T(n - 1) + \Theta(n) = \Theta(n^2)$

Costo Quicksort: caso ottimo?



# Quicksort: Complessità computazionale

Costo di `pivot()`?

- $\Theta(n)$

Costo Quicksort: caso pessimo?

- Il vettore di dimensione  $n$  viene diviso in due sottovettori di dimensione 0 e  $n - 1$
- $T(n) = T(n - 1) + T(0) + \Theta(n) = T(n - 1) + \Theta(n) = \Theta(n^2)$

Costo Quicksort: caso ottimo?

- Dato un vettore di dimensione  $n$ , viene sempre diviso in due sottoproblemi di dimensione  $n/2$
- $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$

# Quicksort: Complessità computazionale

## Partizionamenti parzialmente bilanciati

- Il partizionamento nel caso medio di Quicksort è molto più vicino al caso ottimo che al caso peggiore
- Esempio: Partizionamento 9-a-1:

$$T(n) = T(n/10) + T(9n/10) + cn = \Theta(n \log n)$$

- Esempio: Partizionamento 99-a-1:

$$T(n) = T(n/100) + T(99n/100) + cn = \Theta(n \log n)$$

## Note

- In questi esempi, il partizionamento ha proporzionalità limitata
- I fattori moltiplicativi possono essere importanti

# Quicksort: Complessità computazionale

## Caso medio

- Il costo dipende dall'ordine degli elementi, non dai loro valori
- Dobbiamo considerare tutte le possibili permutazioni
- Difficile dal punto di vista analitico

## Caso medio: un'intuizione

- Alcuni partizionamenti saranno parzialmente bilanciati
- Altri saranno pessimi
- In media, questi si alterneranno nella sequenza di partizionamenti
- I partizionamenti parzialmente bilanciati “dominano” quelli pessimi

## Quicksort: Selezione pivot euristica

Tecnica euristica: selezionare il valore mediano fra il primo elemento, l'ultimo elemento e il valore nella posizione centrale

---

```
int pivot(ITEM[] A, int lo, int hi)
```

---

```
int m =  $\lfloor (lo + hi)/2 \rfloor$ 
```

```
if A[lo] > A[hi] then           % Sposta il massimo in ultima posizione
    swap(A, lo, hi)
```

```
if A[m] > A[hi] then           % Sposta il massimo in ultima posizione
    swap(A, m, hi)
```

```
if A[m] > A[lo] then           % Sposta il mediano in prima posizione
    swap(A, m, lo)
```

```
ITEM pivot = A[lo]
```

```
[...]
```

---