

# Algoritmi e Strutture Dati

## Algoritmi probabilistici

Alberto Montresor

Università di Trento

2018/05/21

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



# Sommario

- 1 Introduzione
- 2 Algoritmi Montecarlo
- 3 Algoritmi Las Vegas

# Introduzione

“Audentes furtuna iuvat” - Virgilio

- Se non sapete quale strada prendere, fate una scelta casuale

Casualità negli algoritmi visti finora

- Analisi del caso medio
  - Si calcola la media su tutti i possibili dati di ingresso in base ad una distribuzione di probabilità
  - Esempio: caso medio Quicksort, si assume che tutte le permutazioni siano equiprobabili
- Hashing
  - Le funzioni hash equivalgono a "randomizzare" le chiavi
  - Distribuzione uniforme

# Introduzione

## Algoritmi probabilistici

Il calcolo delle probabilità è applicato non ai dati di input, ma ai dati di output

- Algoritmi corretti, il cui tempo di funzionamento è probabilistico (**Las Vegas**)
- Algoritmi la cui correttezza è probabilistica (**Montecarlo**)

# Test di primalità

## Piccolo teorema di Fermat

Per ogni numero primo  $n$  e ogni  $b \in [2, \dots, n-1]$ :  $b^{n-1} \bmod n = 1$

Test di primalità di Fermat  
(V1)

---

**isPrime(int  $n$ )**

---

$b = \text{random}(2, n-1)$   
**if**  $b^{n-1} \bmod n \neq 1$  **then**  
   └ **return false**  
**return true**

---

Questo algoritmo è corretto?

Esistono numeri composti tali che:

$\exists b \in [2, \dots, n-1] : b^{n-1} \bmod n = 1$

Output	$n$
<b>false</b>	sicuramente composto
<b>true</b>	possibilmente primo

# Test di primalità

## Piccolo teorema di Fermat

Per ogni numero primo  $n$  e ogni  $b \in [2, \dots, n-1]$ :  $b^{n-1} \bmod n = 1$

Test di primalità di Fermat  
(V2)

---

**isPrime(int  $n$ )**

---

```

for  $i = 1$  to  $k$  do
     $b = \text{random}(2, n - 1)$ 
    if  $b^{n-1} \bmod n \neq 1$  then
        return false
return true

```

---

Questo algoritmo è corretto?

Output	$n$
<b>false</b>	sicuramente composto
<b>true</b>	probabilmente primo

Esistono dei numeri composti (**numeri di Carmichael**), tali che:

$$\forall b \in [2, \dots, n-1] : b^{n-1} \bmod n = 1$$

# Test di Miller-Rabin

- Esprimiamo  $n - 1$  come:  $n - 1 = m2^v$  con  $m$  dispari
  - la rappresentazione binaria di  $n - 1$  è uguale alla rappresentazione binaria del numero dispari  $m$  seguito da  $v$  zeri
- Sia  $b$  un numero in  $[1, \dots, n - 1]$

Se un numero  $n$  è **primo**, allora valgono entrambe le seguenti condizioni

- ①  $\text{mcd}(n, b) = 1$
- ②  $b^m \bmod n = 1$  oppure  $\exists i, 0 \leq i \leq v - 1 : b^{2^i m} \bmod n = -1$

# Test di Miller-Rabin

- Esprimiamo  $n - 1$  come:  $n - 1 = m2^v$  con  $m$  dispari
  - la rappresentazione binaria di  $n - 1$  è uguale alla rappresentazione binaria del numero dispari  $m$  seguito da  $v$  zeri
- Sia  $b$  un numero in  $[1, \dots, n - 1]$

Se un numero  $n$  è **composto**, allora almeno una delle seguenti condizioni è falsa

- ①  $\text{mcd}(n, b) = 1$
- ②  $b^m \bmod n = 1$  oppure  $\exists i, 0 \leq i \leq v - 1 : b^{2^i m} \bmod n = -1$

# Test di Miller-Rabin

- Rabin ha dimostrato che se  $n$  è composto, allora ci sono almeno  $3/4(n - 1)$  valori in  $[1, \dots, n - 1]$  per i quali una delle condizioni sopra è falsa
- Il test di compostezza ha una probabilità inferiore a  $1/4$  di rispondere erroneamente

---

```
isPrime(int n)
```

---

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

```
    b = random(2, n - 1)
```

```
    if isComposite(n, b) then
```

```
        return false
```

---

```
return true
```

---

# Test di Miller-Rabin

## Riassunto

- Complessità:  $O(k \log^2 n \log \log n \log \log \log n)$
- Probabilità di errore:  $(1/4)^k$
- Algoritmo di tipo Montecarlo

## Algoritmi probabilistici vs algoritmi deterministici

- Dal 2002, esiste l'algoritmo deterministico AKS di complessità  $O(\log^{6+\epsilon})$
- I fattori moltiplicativi coinvolti sono molto alti
- Si preferisce quindi l'algoritmo di Miller-Rabin

# Esempio – Espressione polinomiale nulla

## Problema

Data un'espressione algebrica polinomiale  $p(x_1, \dots, x_n)$  in  $n$  variabili, determinare se  $p$  è identicamente nulla oppure no.

## Discussione

- Assumiamo che non sia in forma di monomi - altrimenti è banale
- Gli algoritmi basati su semplificazioni sono molto complessi

# Esempio – Espressione polinomiale nulla

## Algoritmo

- Si genera una  $n$ -pla di valori  $v_1, \dots, v_n$
- Si calcola  $x = p(v_1, \dots, v_n)$ 
  - Se  $x \neq 0$ ,  $p$  non è identicamente nulla
  - Se  $x = 0$ ,  $p$  **potrebbe** essere identicamente nulla
- Se ogni  $v_i$  è un valore intero compreso casuale fra 1 e  $2d$ , dove  $d$  è il grado del polinomio, allora la probabilità di errore non supera  $1/2$ .
- Si ripete  $k$  volte, riducendo la probabilità di errore a  $(1/2)^k$

# Statistica

## Algoritmi statistici su vettori

Estraggono alcune caratteristiche statisticamente rilevanti da un vettore numerico

### Esempi

- **Media:**  $\mu = \frac{1}{n} \sum_{i=1}^n A[i]$
- **Varianza:**  $\sigma^2 = \frac{1}{n} \sum_{i=1}^n (A[i] - \mu)^2$
- **Moda:** il valore (o i valori) più frequenti

# Statistiche d'ordine

## Selezione

Dato un array  $A$  contenente  $n$  valori e un valore  $1 \leq k \leq n$ , trovare l'elemento che si occuperebbe la posizione  $k$  se il vettore fosse ordinato

## Mediana

Il problema del calcolo della mediana è un sottoproblema del problema della selezione con  $k = \lceil n/2 \rceil$ .

# Selezione per piccoli valori di $k$

## Intuizione

- L'albero del torneo può essere “simulato” da uno heap
- L'algoritmo può essere generalizzato a valori generici di  $k > 2$

---

```
heapSelect(ITEM [] A, int n, int k)
```

---

```
buildHeap(A)
```

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

```
  _ deleteMin(A, n)
```

---

```
return deleteMin(A, n)
```

---

Complessità computazionale

- $O(n + k \log n)$
- Se  $k = O(n/\log n)$ , il costo è  $O(n)$
- Non va bene per  $k = n/2$

## Idea

- Approccio divide-et-impera simile al Quicksort
- Essendo un problema di ricerca, non è necessario cercare in entrambe le partizioni, basta cercare in una sola di esse
- Bisogna fare attenzione agli indici

# Algoritmo di selezione

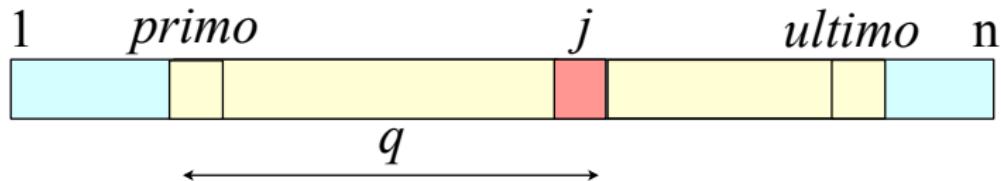
---

```
ITEM selection(ITEM[] A, int start, int end, int k)
```

---

```
if start == end then
    | return A[start]
else
    int j = pivot(A, start, end)
    int q = j - start + 1
    if k == q then
        | return A[j]
    else if k < q then
        | return selection(A, start, j - 1, k)
    else
        | return selection(A, j + 1, end, k - q)
```

---



# Complessità

Caso pessimo

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(n-1) + n & n > 1 \end{cases}$$

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

Caso ottimo

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(n/2) + n & n > 1 \end{cases}$$

$$T(n) = O(n)$$

Caso medio

Assumiamo che `pivot()` restituisca con la stessa probabilità una qualsiasi posizione  $j$  del vettore  $A$

$$\begin{aligned} T(n) &= n - 1 + \frac{1}{n} \sum_{q=1}^n T(\max\{q-1, n-q\}) \\ &\leq n - 1 + \frac{2}{n} \sum_{q=\lfloor n/2 \rfloor}^{n-1} T(q), \quad \text{per } n > 1 \end{aligned}$$

# Complessità

$$\begin{aligned}
 T(n) &\leq n - 1 + \frac{2c}{n} \sum_{q=\lfloor n/2 \rfloor}^{n-1} q \\
 &\leq n + \frac{2c}{n} \left( \sum_{q=1}^{n-1} q - \sum_{q=1}^{\lfloor n/2 \rfloor - 1} q \right) \\
 &= n + \frac{2c}{n} \left( \frac{n(n-1)}{2} - \frac{\lfloor n/2 \rfloor (\lfloor n/2 \rfloor - 1)}{2} \right) \\
 &\leq n + \frac{c}{n} \left( n(n-1) - (n/2 + 1)(n/2) \right) \\
 &= n + c(n-1) - (c/2)(n/2 + 1) = n + cn - c - cn/4 - c/2 \\
 &= cn \left( \frac{1}{c} + \frac{3}{4} - \frac{3}{2n} \right) \leq cn \left( \frac{1}{c} + \frac{3}{4} \right) \leq cn
 \end{aligned}$$

# Complessità

- Siamo partiti dall'assunzione
  - $j$  assume equiprobabilisticamente tutti i valori compresi fra 1 e  $n$
- E se non fosse vero?
- Lo forziamo noi!
  - $A[\text{random}(start, end)] \leftrightarrow A[start]$
- Questo accorgimento vale anche per QuickSort
- La complessità nel caso medio:
  - $O(n)$  nel caso della Selezione
  - $O(n \log n)$  nel caso dell'Ordinamento

# Selezione deterministica

## Algoritmo black-box

Supponiamo di avere un algoritmo “black box” che mi ritorni il mediano di  $n$  valori in tempo  $O(n)$

## Domande

- Potrei utilizzarlo per ottimizzare il problema della selezione?
- Che complessità otterrei?

# Selezione deterministica

Se conoscessi tale algoritmo

- il problema della selezione sarebbe quindi risolto...
- ... ma dove lo trovo un simile algoritmo?

Rilassiamo le nostre pretese

- Supponiamo di avere un algoritmo “black box” che mi ritorni un valore che dista al più  $\frac{3}{10}n$  dal mediano (nell’ordinamento)
- Potrei utilizzarlo per ottimizzare il problema della selezione?
- Che complessità otterrei?

# Selezione deterministica

## Idea

- Suddividi i valori in gruppi di 5. Chiameremo l' $i$ -esimo gruppo  $S_i$ , con  $i \in [1, \lceil n/5 \rceil]$
- Trova il mediano  $M_i$  di ogni gruppo  $S_i$
- Tramite una chiamata ricorsiva, trova il mediano  $m$  dei mediani  $[M_1, M_2, \dots, M_{\lceil n/5 \rceil}]$
- Usa  $M$  come pivot e richiama l'algoritmo ricorsivamente sull'array opportuno, come nella `selection()` randomizzata
- Quando la dimensione scende sotto una certa dimensione, possiamo utilizzare un algoritmo di ordinamento per trovare il mediano

## Selezione deterministica

---

ITEM **select**(ITEM[]  $A$ , int  $start$ , int  $end$ , int  $k$ )

---

% Se la dimensione è inferiore ad una soglia (10), ordina il vettore e  
% restituisci il  $k$ -esimo elemento di  $A[start \dots end]$

**if**  $end - start + 1 \leq 10$  **then**

**InsertionSort**( $A, start, end$ )         % Versione con indici inizio/fine  
    **return**  $A[start + k - 1]$

% Divide  $A$  in  $\lceil n/5 \rceil$  sottovettori di dim. 5 e ne calcola la mediana

$M = \text{new int}[1 \dots \lceil n/5 \rceil]$

**for**  $i = 1$  **to**  $\lceil n/5 \rceil$  **do**

$M[i] = \text{median5}(A, start + (i - 1) \cdot 5, end)$

% Individua la mediana delle mediane e usala come perno

ITEM  $m = \text{select}(M, 1, \lceil n/5 \rceil, \lceil \lceil n/5 \rceil / 2 \rceil)$

int  $j = \text{pivot}(A, start, end, m)$                   % Versione con  $m$  in input

[...]

---

# Selezione deterministica

---

ITEM **select**(ITEM[] *A*, int *start*, int *end*, int *k*)

---

[...]

```
% Calcola l'indice q di m in [start . . . end]  
% Confronta q con l'indice cercato e ritorna il valore conseguente  
int q = j - start + 1  
if q == k then  
|   return m  
else if q < k then  
|   return select(A, start, q - 1, k)  
else  
|   return select(A, q + 1, end, k - q)
```

---

## Selezione deterministica

- Il calcolo dei mediani  $M[]$  richiede al più  $6\lceil n/5 \rceil$  confronti.
- La prima chiamata ricorsiva dell'algoritmo `select()` viene effettuata su  $\lceil n/5 \rceil$  elementi
- La seconda chiamata ricorsiva dell'algoritmo `select()` viene effettuata al massimo su  $7n/10$  elementi  
(esattamente  $n - 3\lceil \lceil n/5 \rceil / 2 \rceil$ )
- L'algoritmo `select()` esegue nel caso pessimo  $O(n)$  confronti

$$T(n) = T(n/5) + T(7n/10) + 11/5n$$

1	4	7	10	13	2	5	8	11	14	3	6	9	12	15
---	---	---	----	----	---	---	---	----	----	---	---	---	----	----

# Conclusioni

Quale preferire?

- Algoritmo probabilistico Las Vegas in tempo atteso  $O(n)$
- Algoritmo deterministico in tempo  $O(n)$ , con fattori moltiplicativi più alti

Note storiche

- Nel 1883 Lewis Carroll (!) notò che il secondo premio nei tornei di tennis non veniva assegnato in maniera equa.
- Nel 1932, Schreier dimostrò che  $n + \log n - 2$  incontri sono sempre sufficienti per trovare il secondo posto
- Nel 1973, a opera di Blum, Floyd, Pratt, Rivest e Tarjan, appare il primo algoritmo deterministico