

16.1 Algoritmi probabilistici

Idea. Se non sapete cosa fare fate una scelta casuale.

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

16.2 Algoritmi Montecarlo

Idea. Sono algoritmi in cui la correttezza è probabilistica.

MATERIALE MANCANTE

16.2.1 Test di primalità

Teorema (Piccolo teorema di Fermat). *Per ogni numero primo n e ogni $b \in [2, \dots, n-1]$: $b^{n-1} \bmod n = 1$.*

Algoritmo 1: Test di primalità

```
isPrime(int n)
|
|   b ← random(2, n - 1)
|   if  $b^{n-1} \bmod n \neq 1$  then
|       |   // non può essere un numero primo
|       |   return false
|   // è un numero primo
|   return true
```

16.3 Algoritmi Las Vegas

Idea. Sono algoritmi corretti, in cui il funzionamento è probabilistico.

MATERIALE MANCANTE

16.3.1 Statistiche d'ordine

MATERIALE MANCANTE

Algoritmo 2: inserisci didascalia

```
heapSelect(ITEM[] A, int n, int k)
|
|   heapBuild(A)
|   for  $i = 1$  until  $k - 1$  do
|       |   deleteMin(A, n)
|       // restituisce il  $k$ -esimo valore
|   return deleteMin(A, n)
```

Analisi della complessità Non va bene per $k = n/2$, perchè viene $\mathcal{O}(n + n/2 \log n) = \mathcal{O}(n \log n)$

Cambiamo tattica. Utilizziamo una tecnica dividi-et-impera simile al quickSort. Ma, a differenza di quest'ultimo, non è necessario cercare in entrambe le partizioni.

Algoritmo 3: inserisci didascalìa

```

selection(ITEM[] A, int start, int end, int k)
|
|   if start == end then
|   |   // caso base: ho un solo elemento
|   |   return A[start]
|   else
|   |   // calcolo indici
|   |   int j = perno(A, start, end) // perno
|   |   int q = j - start + 1 // mediana
|   |   if k == q then
|   |   |   // ho trovato il mio elemento
|   |   |   return A[j]
|   |   else if k < q then
|   |   |   // cerco a sinistra
|   |   |   return selection(A, start, j - 1, k)
|   |   else
|   |   |   // cerco a destra
|   |   |   return selection(A, j + 1, end, k - q)
|
|
|

```

Analisi della complessità Effettuiamo un'analisi simile a quella fatta per il quickSort.

Caso pessimo Il caso pessimo è lo stesso del quickSort (che è il caso in cui il vettore sia già ordinato). In quanto dividiamo il vettore con 0 elementi a sinistra ed n elementi a destra della partizione.

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

$$T(n) = \mathcal{O}(n^2)$$

Quindi nel caso pessimo questo algoritmo risulta peggiore di ordinare il vettore ($\mathcal{O}(n \log n)$) e di andare a fare la ricerca.

Caso ottimo Nel caso ottimo ogni volta divido esattamente a metà

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

$$T(n) = \mathcal{O}(n)$$

Caso medio Assumiamo che perno restituisca con la stessa probabilità una qualsiasi posizione j del vettore A :

$$T(n) = n - 1 + \frac{1}{n} \sum_{q=1}^n T(\max\{q-1, n-q\})$$

$$\leq n - 1 + \frac{2}{n} \sum_{q=\lceil n/2 \rceil}^{n-1} T(q)$$

per $n > 1$

$\frac{1}{n}$ la media su tutti i possibili valori di q $\sum_{q=1}^n$ rappresenta la dimensionalità di j , $q-1$ gli elementi a sinistra, $n-q$ a destra rispettivamente.

$\exists c > 0, \exists m \geq 0 : T(n) \leq cn, \forall n \geq m$

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

divido la sommatoria a metà
algebra
semplifico
eseguo i calcoli
eseguo i calcoli
la mia disequazione risulta vera

Siamo partiti dall'assunzione che j assuma equiprobabilisticamente tutti i valori compresi fra 1 ed n . Se questa assunzione non fosse vera allora i nostri calcoli non avrebbero alcun fondamento. Forziamo quindi l'assunzione iniziale scegliendo un valore casuale come perno ($A[\text{random}(start, end)]$) a differenza di prima dove prendevamo il primo valore ($A[start]$). Questo accorgimento vale anche per **quickSort**. La complessità nel caso medio diventa quindi $\mathcal{O}(n)$.

16.3.2 Selezione deterministica

Supponiamo di avere un algoritmo "black box" che mi ritorni un valore che dista al più $\frac{3}{10}n$ dal mediano nell'ordinamento.

Implementazione L'implementazione dell'algoritmo consiste nei seguenti passaggi:

- 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 sul vettore opportuno, come nella **selection randomizzata**
- quando la dimensione scende sotto una certa dimensione, possiamo utilizzare un algoritmo di ordinamento per trovare il mediano

inserisci didascalia

```
ITEM[] select(ITEM[] A, int primo, int ultimo, int k)
    // se la dimensione è inferiore ad una soglia (10), ordina il vettore e
    // restituisci il k-esimo elemento di A[primo][ultimo]
    if ultimo - primo + 1 ≤ 10 then
        | insertionSort(A, primo, ultimo) // Versione con indici inizio/fine
        | return A[primo + k - 1]

    // divide A in ⌈n/5⌉ sottovettori di dim. 5 e ne calcola la mediana
    M ← new int[1...⌈n/5⌉]
    (1) from i ← 1 until ⌈n/5⌉ do
        | M[i] ← median5(A, primo + (i - 1) · 5, ultimo)

    // individua la mediana delle mediane e usala come perno
    (2) ITEM m ← select(M, 1, ⌈n/5⌉, ⌈⌈n/5⌉/2⌉)

    int j ← perno(A, primo, ultimo, m) // Versione con m in input

    // calcola l'indice q di m in [primo...ultimo]
    int q ← j - primo + 1

    // confronta q con l'indice cercato e ritorna il valore conseguente
    if q == k then
        | return m
    else if q < k then
    (3) | return select(A, primo, q - 1, k)
    else
    (3) | return select(A, q + 1, ultimo, k - q)

    // calcola la mediana fra 5 elementi
    int median5
    | // Restituisco la mediana
    | return m
```

Analisi della complessità

- ① il calcolo dei mediani $M[]$ richiede al più $6 \lceil n/5 \rceil$ confronti;
- ② la prima chiamata ricorsiva dell'algoritmo `select` viene effettuata su $\lceil n/5 \rceil$ elementi;
- ③ la seconda chiamata ricorsiva dell'algoritmo `select` viene effettuata al massimo su $7 \frac{n}{10}$ elementi (esattamente $n - 3 \lceil \frac{n/5}{2} \rceil$).

L'algoritmo `select` esegue nel caso pessimo $O(n)$ confronti:

$$T(n) = T\left(\frac{n}{5}\right) + T\left(7 \frac{n}{10}\right) + \frac{11}{5}n = O(n)$$