# Statistiche d'ordine

Michele Pommella

Corso di Algoritmi e Strutture Dati

Prof. Stefano Avallone

Anno Accademico 2017-2018

**Sommario**

Questo elaborato mira ad analizzare gli algoritmi affrontati relativi alla selezione della *i-esima* statistica d'ordine, ovvero l'*i-esimo* elemento più piccolo di un insieme di $n$ elementi. L'attenzione è posta sul tempo di esecuzione dei diversi algoritmi. Gli algoritmi analizzati sono il RANDOMIZED-SELECT e l'OS-SELECT.

## 1 Introduzione

Gli algoritmi specifici al problema consentono di selezionare la statistica d'ordine senza effettuare un ordinamento preliminare dell'insieme, permettendo di ottenere un tempo di esecuzione migliore al limite inferiore $\Omega(n \log n)$ per l'ordinamento. L'implementazione di RANDOMIZED-SELECT e OS-SELECT è stata effettuata mediante il linguaggio di programmazione C. Le misure dei tempi di esecuzione sono state attuate attraverso le librerie offerte dal linguaggio, ed in particolare sfruttando il metodo *clock()* della libreria *time.h*. Esso restituisce il numero di *clock ticks* (unità di tempo di lunghezza costante, ma specifica del sistema) trascorsi dall'esecuzione del programma.

## 2 Randomized Select

Presentiamo un'implementazione dell'algoritmo RANDOMIZED-SELECT.

```
int RandomizedSelect(array a, int p, int r, int i) {
```

```
        if(p == r)
                return a[p];
        int q = RandomizedPartition(a, p, r);
        //int q = Partition(a, p, r); //per il worst case
        int k = q-p+1;
        if(i == k)
                return a[q];
        else {
                if(i < k)
                        return RandomizedSelect(a, p, q-1, i);
                else
                        return RandomizedSelect(a, q+1, r, i-k);
        }
}
```

L'algoritmo RANDOMIZED-SELECT presenta nel caso peggiore una complessità temporale asintotica limitata superiormente da $O(n^2)$. Ciò si verifica se la partizione, effettuata per calcolare la statistica d'ordine, avviene sempre intorno all'elemento più grande. Per la verifica di tale condizione, è stata utilizzata la funzione PARTITION sull'elemento più grande dell'insieme, evitando la scelta casuale del *pivot*.

```
int Partition(array a, int p, int r) {
        int x = a[r];
        int i = p-1;
        for(int j = p; j < r; j++) {
                if(a[j] <= x) {
                        i++;
                        Swap(&a[i], &a[j]);
                }
        }
        Swap(&a[i+1], &a[r]);
        return i+1;
}
```

Sono stati inseriti nella struttura dati gli elementi in ordine crescente (per ottenere la partizione sempre sul massimo) e si è provato a ricercare il minimo tra essi. In figura 1 si possono osservare i risultati ottenuti. Vengono fatte 100 iterazioni, nelle quali l'algoritmo è testato su un vettore che cresce di passo 100 ogni volta, fino ad arrivare a 10000 elementi. L'andamento è quadratico e l'algoritmo, già con un vettore di un migliaio di elementi, ha un tempo di esecuzione dell'ordine dei millesimi di secondo, fino ad arrivare ai decimi di secondo per migliaia elementi.

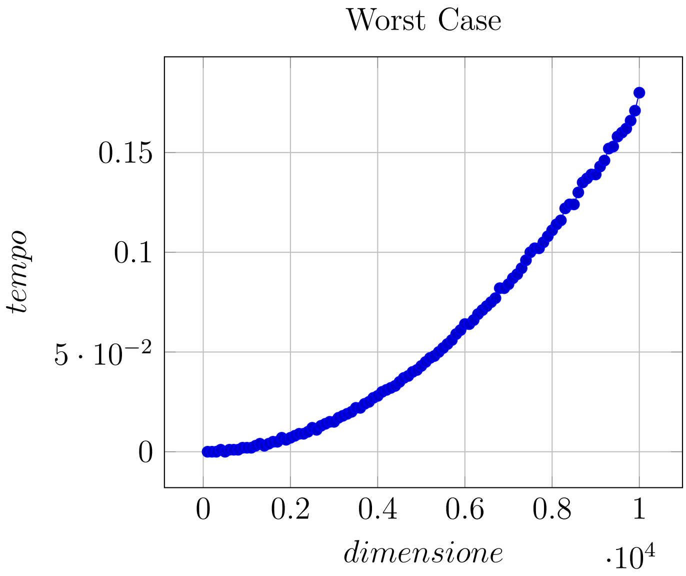


**Figura 1:** Al crescere di $n$, il tempo di esecuzione presenta un adamento quadratico nel caso peggiore. Analisi effettuata facendo crescere la dimensione del problema con passo 100 ad ogni iterazione fino a 10000.

La funzione di RANDOMIZED-PARTITION, però, evita che ci si attesti sempre nel caso peggiore. Grazie ad essa l'algoritmo presenta un tempo atteso lineare $O(n)$.

```
int RandomizedPartition(array a, int p, int r) {
        int i = Random(p, r);
        Swap(&a[r], &a[i]);
        return Partition(a, p, r);
}

int Random(int p, int r) {
        srand(time(0));
        return ((rand() % (r-p+1)) + p);
}
```

Sono stati inseriti nel vettore elementi scelti casualmente e si è provata la ricerca del minimo. In figura 2 e 3 possiamo osservare i risultati ottenuti. Esse derivano dal calcolo del tempo medio ad ogni iterazione, per 100 iterazioni. Il primo esempio si pone nelle stesse condizioni del *worst case* precedente, ma si attesta ad un tempo medio dell'ordine di $10^{-4}$ secondi. Il secondo esempio, invece, analizza un vettore di 500000 elementi, con passo di crescita di 5000 ad ogni iterazione. Al più il tempo medio raggiunge l'ordine dei millisecondi,

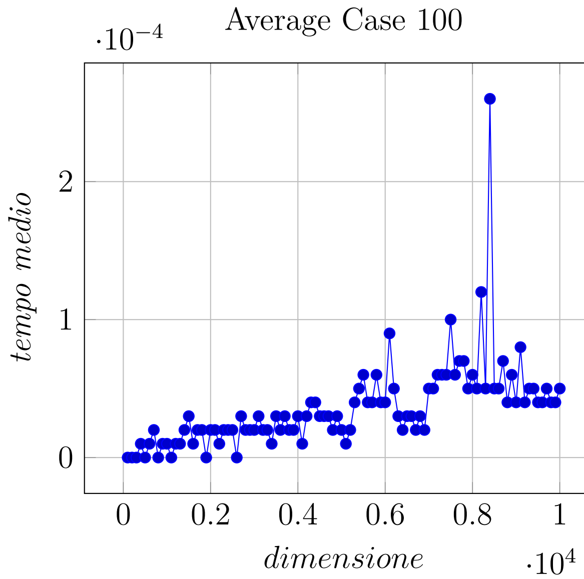


**Figura 2:** Tempo atteso relativo alla crescita della dimensione del problema con passo 100 ad ogni iterazione fino a 10000.

risultato comunque migliore del *worst case* anche se relativo ad un vettore 50 volte più grande. Ciò è possibile in virtù della complessità asintotica del caso medio che ha come limite superiore un andamento lineare.

# 3 Order Statistic Select

L'algoritmo OS-SELECT utilizza un albero *rosso-nero* per il calcolo delle statistiche d'ordine. Per fare ciò, però, viene introdotto nell'albero il dato satellite *size[x]*, che indica il numero di nodi interni nel sottoalbero con radice $x$, includendo $x$ stesso. La dimensione del sottoalbero è definita nulla per la sentinella ($size[nil[T]] = 0$), mentre per i nodi interni come:

$$size[x] = size[left[x]] + size[right[x]] + 1$$

Per tale motivo le funzioni relative all'inserimeto di un nodo nell'albero *rosso-nero* sono state modificate per aggiornare il campo *size* dei nodi dell'albero. L'algoritmo sfrutta il rango dei nodi dell'albero, ovvero la posizione che essi occupano nella sequenza ordinata degli elementi dell'insieme. Un nodo con rango $i$ avrà una chiave che va proprio a coincidere con la statistica di ordine *i-esimo*.

```
//OSSelect restituisce un puntatore al nodo che contiene
```

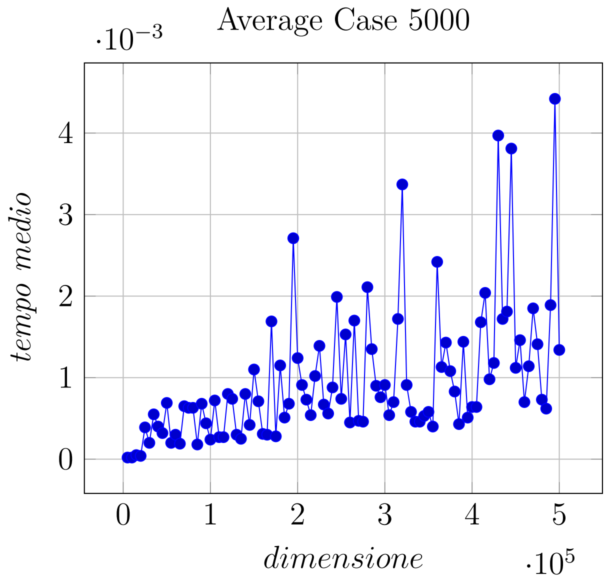


**Figura 3:** Tempo atteso relativo alla crescita della dimensione del problema con passo 5000 ad ogni iterazione fino a 500000.

```
//l'i-esima chiave piu' piccola nel sottoalbero con
//radice in x
node* OS_Select(node* x, int i, rbTree tree) {
        if(i < 1 || i > x->size)
                return tree.nil;
        int r = x->left->size + 1; //rango di x nel
                                   //sottoalbero di radice x
//la posizione di x nella sequenza ordinata degli
//elementi del sottoalbero di radice x e' quella che
//segue tutti gli elementi del suo sottoalbero sinistro
        if(i == r)
                return x; //x e' l'i-esimo elemento piu' piccolo
        else {
                if(i < r) //l'i-esimo elemento piu' piccolo e' nel
                                  //sottoalbero sinistro
                        return OS_Select(x->left, i, tree);
                else //l'i-esimo elemento piu' piccolo e' nel
                        //sottoalbero destro
                        return OS_Select(x->right, i-r, tree);
//l'i-esimo elemento piu' piccolo nel sottoalbero con
//radice in x e' l'(i-r)-esimo elemento piu' piccolo nel
```

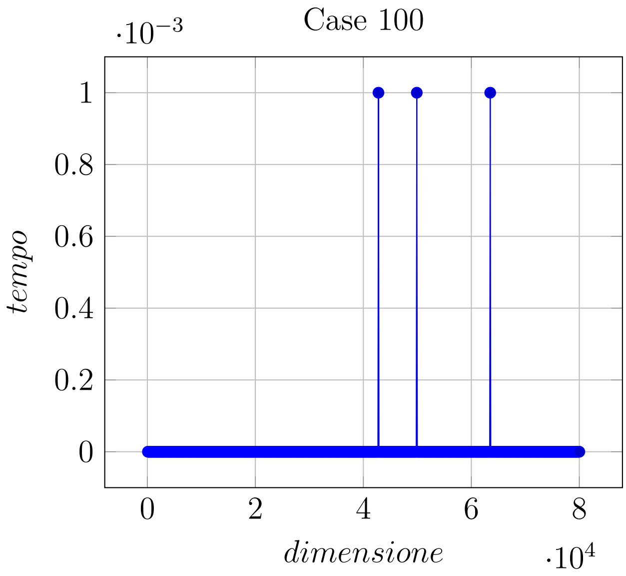


**Figura 4:** Tempo di esecuzione relativo alla crescita delle dimensioni del problema con passo 100 ad ogni iterazione fino a 80000.

```
//sottoalbero con radice x->right, poiche' ci sono r
//elementi che precedono il sottoalbero destro di x
        }
}
```

Poiché per ogni chiamata ricorsiva si scende di un livello nell'albero di statistiche d'ordine, il tempo di esecuzione di OS-SELECT nel caso peggiore è proporzionale all'altezza dell'albero, quindi $O(\log n)$ per un albero *rosso-nero*.

In figura 4 possiamo osservare i risultati ottenuti con un albero *rosso-nero* che cresce di 100 elementi ad ogni iterazione, fino a raggiungere le 80000 unità. L'esempio è ottenuto inserendo nodi dalla chiave casuale nell'albero e cercando il minimo. Il tempo di esecuzione raggiunge in sporadici casi l'ordine dei millisecondi, ma in generale si attesta sotto i microsecondi. Con la funzione *clock()*, infatti, è possibile misurare tempi dell'ordine di $10^{-6}$ secondi, e probabilmente l'algoritmo impiega un tempo inferiore a questa soglia di percezione. Da notare, però, che i vantaggi temporali si scontrano con la maggiore complessità spaziale di questa soluzione: sulla stessa macchina i limiti superiori per l'istanziazione delle strutture dati sono stati rispettivamente 500000 e 80000 per il RANDOMIZED-SELECT e l'OS-SELECT.

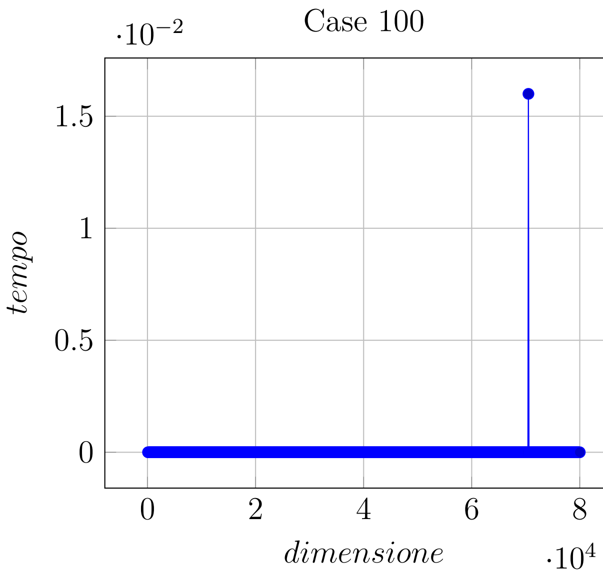


**Figura 5:** Tempo di esecuzione relativo alla crescita delle dimensioni del problema con passo 100 ad ogni iterazione fino a 80000.