Relazione laboratorio Algoritmi e Strutture Dati

3 agosto 2020

Brugnera Matteo 137370@spes.uniud.it 137370
Rasera Giovanni 143395@spes.uniud.it 143395

UNIVERSITÀ
DEGLI STUDI
DI UDINE

Indice

1 Analisi QuickSelect

1.1 Introduzione

L'algoritmo **QuickSelect** è una variante dell'algoritmo **QuickSort**, utile per la selezione del k-esimo numero più piccolo all'interno di un vettore di interi non ordinati.
L'algoritmo in questione, proprio come QuickSort, sceglie un *pivot* all'interno del vettore e con esso esegue la sotto-procedura **partition**. Quest'ultima sposta tutti gli elementi minori o uguali alla sua sinistra, mentre colloca tutti gli elementi maggiori di lui alla sua destra, ritornando alla fine la posizione occupata dal pivot all'interno del vettore.
Una volta eseguito il partition, a differenza del QuickSort (dove vengono fatte due chiamate ricorsive in entrambe le porzioni del vettore), il QuickSelect controlla tre casi:

- *la posizione del pivot coincide con k*: in questo caso l'algoritmo ritorna il pivot stesso in quanto ha trovato il k-esimo elemento più piccolo;
- *la posizione del pivot è più grande rispetto a k*: in questo caso viene fatta una chiamata ricorsiva nella parte destra del vettore (rispetto al pivot);
- *la posizione del pivot è minore rispetto a k*: in questo caso viene fatta una chiamata ricorsiva nella parte sinistra del vettore (rispetto al pivot);

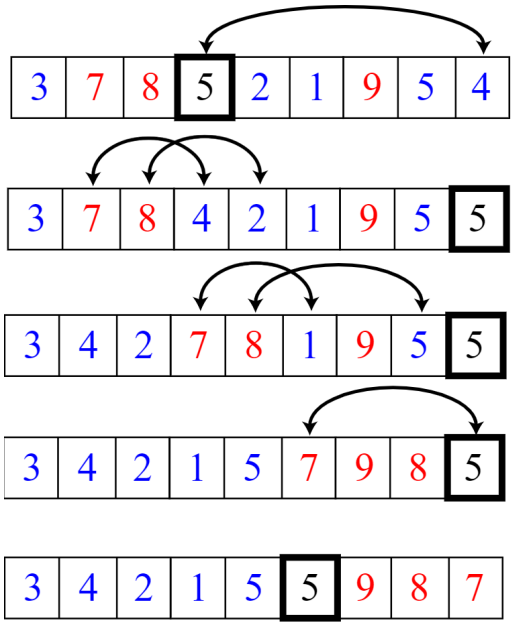

Figura 1: Rappresentazione grafica dell'esecuzione di QuickSelect

1.2 Analisi e calcolo dei tempi

L'algoritmo preso in considerazione ha una complessità temporale pari a **O(n) nel caso medio** e $\Theta(n^2)$ **nel caso pessimo**. Questo dipende dalla scelta del pivot, infatti se ne scelgo uno tale che ad ogni chiamata ricorsiva mi diminuisce il vettore di un solo elemento, arrivo ad avere un tempo quadratico.

Di seguito viene riportato un esempio di computazione eseguita con il Quick-Select, dove vengono tenuti in considerazione il numero di elementi all'interno del vettore, il k-esimo elemento, il tempo impiegato e la deviazione:

Numero elementi	K-esimo elemento	Tempo	Deviazione
2	1	3673685.0	96960.04627485618
3	1	3380865.0	5514.772105196189
4	2	3387515.0	11112.122132709397
6	3	3392090.0	12311.30858320918
9	4	3392415.0	15538.820921718934
ecc...			

Con questi dati si riesce ad ottenere quindi un grafico che rispecchia esattamente la complessità temporale sopra citata:

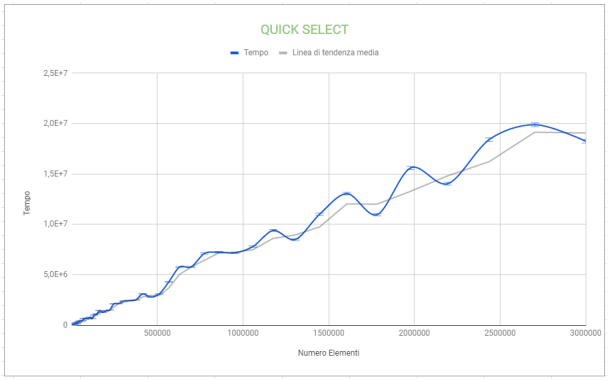

Figura 2: Grafo di QuickSelect in scala lineare

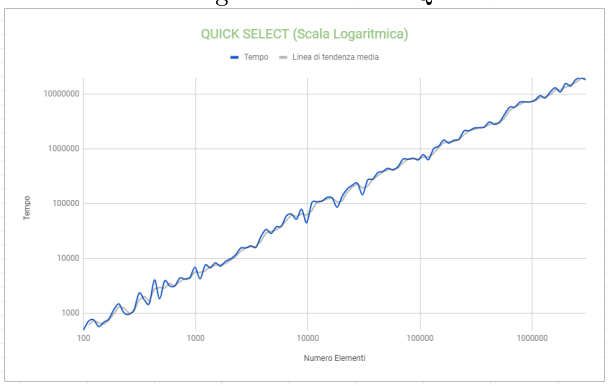

Figura 3: Grafo di QuickSelect in scala logaritmica

2 Analisi HeapSelect

2.1 Introduzione

L'algoritmo in questione sfrutta due min-Heap per calcolare il k-esimo numero più piccolo di un vettore.
L'input dato verrà trasformato nella min-Heap principale, grazie alla procedura *buildHeap*, ed essa non verrà mai modificata.
Ci sono inoltre altre due strutture di supporto che fanno sì che il codice sia più snello:

- **un vettore di booleani** lungo quanto l'input fornito, utile per salvarsi se un nodo (rappresentato dalla posizione) è stato eliminato o meno;
- **una seconda Heap**, l'unica che verrà modificata.

Inizialmente il vettore di supporto *sarà inizializzato con tutti zeri* in quanto nessuno degli elementi della Heap è stato estratto, mentre la Heap secondaria *conterrà un solo nodo corrispondente alla radice della Heap principale.*

All'i-esima iterazione, *l'algoritmo estrae la radice dalla Heap secondaria e la sostituisce con i suoi due figli* (da notare che, una volta inseriti, verrà eseguita l'operazione Heapify). *Contemporaneamente cerco dove si trova l'elemento estratto all'interno del vettore di supporto* e stabilisco il suo valore di eliminazione a **true**.
Continuo in questo modo finché il numero di chiamate a questa procedura *non coincide con k.*
A questo punto scorro tutto il vettore di supporto finché non trovo l'elemento che *coincide con quello della radice della Heap secondaria e il suo valore di eliminazione è pari a zero.*
Quello sarà quindi il mio k-esimo elemento.

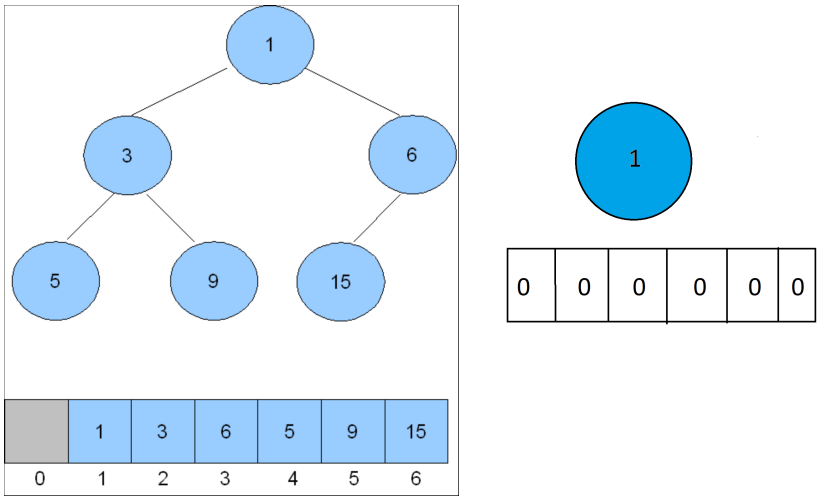

Figura 4: Rappresentazione della Heap principale, secondaria e vettore di supporto

2.2 Analisi e calcolo dei tempi

L'algoritmo in questione ha una complessità temporale pari a (O(n + klogk)) sia nel caso pessimo che nel caso medio.
Se k è piccolo rispetto alla dimensione dell'input, HeapSelect risulta essere più efficiente in termini di tempo rispetto a QuickSelect ma, come vedremo poi nell'ultimo capitolo, in media si preferisce QuickSelect.

Di seguito viene riportato un esempio di computazione eseguita con il HeapSelect, dove vengono tenuti in considerazione il numero di elementi all'interno del vettore, il k-esimo elemento, il tempo impiegato e la deviazione:

Numero elementi	K-esimo elemento	Tempo	Deviazione
4	1	146.796	16.9014
8	1	174.881	17.2108
16	2	311.335	23.7611
32	3	580.495	24.8793
64	4	1178.26	92.4898
ecc...			

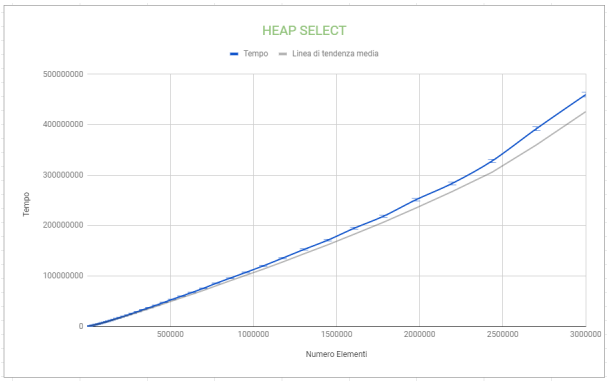

Figura 5: Grafo di HeapSelect in scala lineare

Figura 6: Grafo di HeapSelect in scala logaritmica

3 Analisi MomSelect

3.1 Introduzione

L'algoritmo **MomSelect** è un algoritmo che trova il k-esimo elemento più piccolo di un vettore, ed *esso può sfruttare sia un procedimento in-place che non.* Iniziamo con l'analizzare la *procedura non in-place.* L'idea è di **dividere il vettore in sottosezioni di 5 elementi** (presumiamo per comodità che la lunghezza del vettore sia un multiplo di 5).
Successivamente **ordiniamo questi sotto-vettori e cerchiamo l'elemento medio**, che si troverà alla posizione (NUMERO GRUPPO + 2).

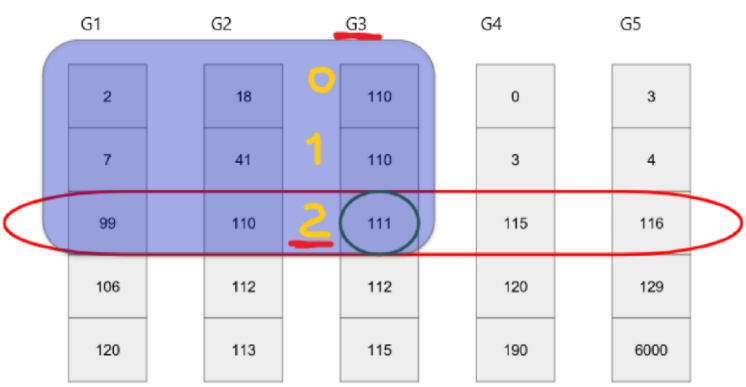

Figura 7: Rappresentazione del funzionamento della procedura MOMSelect

Tutti gli elementi mediani verranno poi salvati in un vettore grande (VETTORE.length / 5) e verrà fatta una chiamata ricorsiva (che si può togliere come vedremo più avanti) a MOMSelect(NUOVO.VETTORE, NUOVO.K).
Dopo l'i-esimo passaggio, *la procedura ci darà come risultato la posizione della mediana delle mediane* che passeremo alla funzione partition: res = partition(VETTORE, start, finish, risultatoMOM).

Qui dovremmo gestire i seguenti casi (che derivano da QuickSelect):

- **La posizione del pivot coincide con k**: in questo caso l'algoritmo ritorna il pivot stesso in quanto ha trovato il k-esimo elemento più piccolo;
- **la posizione del pivot è più grande rispetto a k**: in questo caso viene fatta una chiamata ricorsiva nella parte destra del vettore (rispetto al pivot);
- **la posizione del pivot è minore rispetto a k**: in questo caso viene fatta una chiamata ricorsiva nella parte sinistra del vettore (rispetto al pivot);

3.2 Versione in place

Invece di fare una chiamata ricorsiva, **è possibile salvare i mediani nella sezione dell'array da [0 a len/5]** e ripetere questa operazione fino a quando troviamo un solo elemento che sarà la nostra mediana delle mediane.
Infine vengono ricalcolati gli elementi (k , inizio, fine) e viene eseguita partition fino a quando si trova il k interessato.

3.3 Analisi e calcolo dei tempi

L'algoritmo MOMSelect ha complessità temporale e spazia pari a $\Theta(n)$ nel caso pessimo.
Indipendentemente da k c'e bisogno di trovare la mediana delle mediane, quindi dobbiamo sempre eseguire i raggruppamenti di 5 elementi anche se stiamo cercando un valore di k molto piccolo.
Nella tabelle vengono riportati degli esempi di tempi di esecuzione sia in-place che non:

Numero elementi	K-esimo elemento	Tempo (non in-place)	Deviazione (non in-place)
8192	4096	224003	4243.44
16384	8192	435563	12225.2
32768	16384	846667	11722
65536	32768	1.65273e+06	5267.16
ecc...			

Numero elementi	K-esimo elemento	Tempo (in-place)	Deviazione (in-place)
8192	4096	127003	949.597
16384	8192	265493	411.384
32768	16384	597973	11193.5
65536	32768	1.09748e+06	1911.02
ecc...			

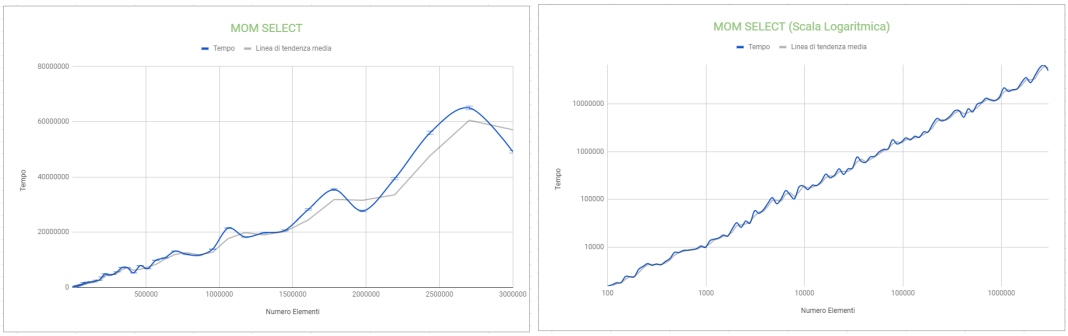

Figura 8: Rappresentazione dei grafici in scala lineare e logaritmica

4 Algoritmo per il calcolo dei tempi

Per generare dei vettori composti da valori randomici con distribuzione uniforme, abbiamo utilizzato l'algoritmo di **mt19937** presente all'interno dell'std di cpp e si chiama Mersenne Twister.
Per calcolare il tempo di esecuzione dei vari programmi, abbiamo iniziato con il misurare la **granularità del sistema**, ovvero *il livello di dettaglio con cui analizziamo gli algoritmi.*
Si calcola la granularità 1000 volte, salvandone i risultati all'interno di un vettore, per poi **prendere la media tra tutte**, ottenendo così una stima più precisa.
Successivamente si calcola Tmin.

```
//Calcolo la granularità
long risoluzione = getRobustResolution();
double erroreMassimo = 0.01;                    //1%
double tMin = risoluzione / erroreMassimo + risoluzione;
```

A questo punto possiamo calcolare *quante volte dobbiamo eseguire il codice di coppia del vettore* affinchè sia possibile *scomputarlo dall'esecuzione dell'algoritmo di selezione.* Per questo utilizziamo come prima cosa **calcolaRipTara**.

```
/*
EFFETTO: Calcola quante volte deve essere eseguita la tara affinchè venga scomputata correttamente.
PARAMETRI:
    - Prepara       : Lambda        -> funzione da eseguire per caricare il vettore
    - vettore       : int *         -> il vettore da caricare
    - lenVettore    : int           -> lunghezza del vettore
    - k             : int           -> non utile in questo caso
    - tMin          : long          -> risoluzione del sistema
*/
long calcolaRipTara(Lambda* Prepara, int* vettore, int lenVettore, int k, long tMin) {
    long t0 = millis(), t1 = millis(), rip = 1;

    //Calcolo delle ripetizioni massime
    while (t1 - t0 <= tMin) {
        //stima di rip con crescita esponenziale
        rip += 2;
        t0 = millis();
        for (int i = 0; i < rip; i++) {
            int n = Prepara->execute(vettore, 0, lenVettore, 0);
        }
        t1 = millis();
    }

    long max = rip, min = rip / 2, cicliErrati = 5;
    //Ricerca esatta del numero di ripetizioni per bisezione
    while (max - min >= cicliErrati) {
        rip = (max + min) / 2;
        t0 = millis();
        for (int i = 0; i < rip; i++) {
            int n = Prepara->execute(vettore, 0, lenVettore, 0);
        }
        t1 = millis();
        if (t1 - t0 <= tMin)
            min = rip;
        else
            max = rip;
    }
    return max;
}
```

Figura 9: Codice del calcolo del numero di ripetizioni della tara

Successivamente calcoliamo il **lordo**: *esecuzione tara(carica vettore) più esecuzione netto(algoritmo selezione).*

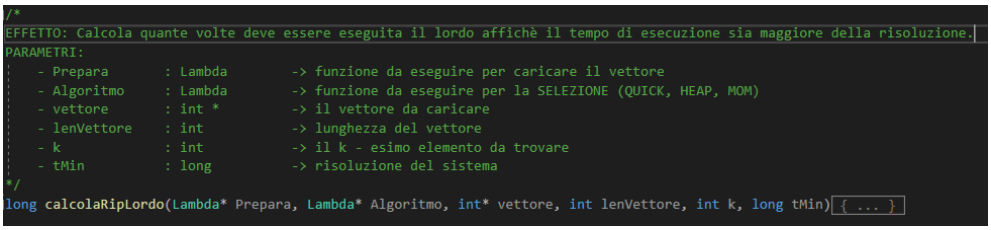

Figura 10: Codice del calcolo del numero di ripetizioni del lordo

I due valori precedentemente ottenuti ci servono per la prossima funzione che ci permette di *calcolare il tempo vero e proprio di esecuzione dell'algoritmo di selezione desiderato.*

Al termine di tutto, **viene calcolato il tempo di esecuzione finale**.

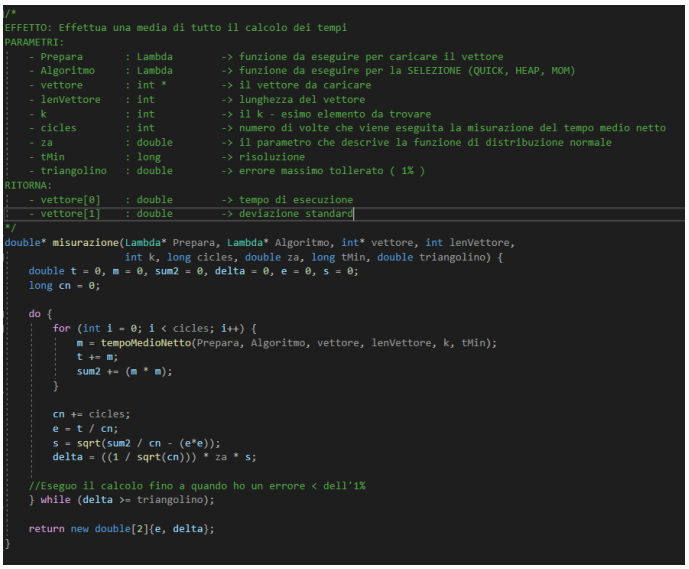

Figura 11: Codice del calcolo dei tempi

Molto importante è stato l'utilizzo di una distribuzione geometrica per calcolare il numero di elementi in modo da ottenere un buon campionamento.

5 Confronto tra gli algoritmi

Dopo aver analizzato individualmente il ragionamento, la composizione e la struttura di tutti gli algoritmi, passiamo all'ultima analisi, ovvero le differenze che ci sono tra i vari grafici ottenuti con il calcolo dei tempi.
Di seguito riporto:

- la tabella che comprende il calcolo dei *tempi medi di esecuzione* dei singoli algoritmi, messi a confronto un con l'altro;

Numero elementi	**QuickSelect**	**HeapSelect**	**MomSelect**
4	22.5042	146.796	139.599
8	43.4123	174.881	184.609
16	98.2555	311.335	243.656
...			
524288	3.94463e+06	5.51726e+07	1.25584e+07
1048576	8.19776e+06	1.19636e+08	2.48078e+07
2097152	1.84084e+07	2.66823e+08	4.96787e+07
ecc...			

- la tabella che comprende il calcolo della *deviazione standard* di ogni algoritmo;

Numero elementi	**QuickSelect**	**HeapSelect**	**MomSelect**
4	5.47613	16.9014	18.7699
8	7.28564	17.2108	10.2336
16	8.06308	23.7611	24.2416
...			
524288	3020.52	36592.6	81448.8
1048576	10400.3	156894	41204
2097152	19354.6	895968	239738
ecc...			

Si può notare fin da subito come l'**HeapSelect** sia l'algoritmo che cresce più velocemente tra i tre, in quanto, come visto nella sua spiegazione, ha una complessità temporale pari a **O(n + klogk)**.
Il caso più interessante riguarda il confronto tra l'**QuickSelect** e il **MomSelect**. Entrambi infatti hanno *complessità temporale lineare*, però *il QuickSelect risulta*, nel caso medio, *più efficiente rispetto al MomSelect* in quanto esso è O(n), mentre il QuickSelect è $\Theta(n)$.
Può capitare però che il QuickSelect sia più lento di entrambi gli algoritmi, cosa che accade solo se mi trovo nel caso peggiore.
Tutte queste differenze si possono notare in modo più evidente grazie ai grafici, espressi sia in **scala lineare** che in **scala logaritmica**:

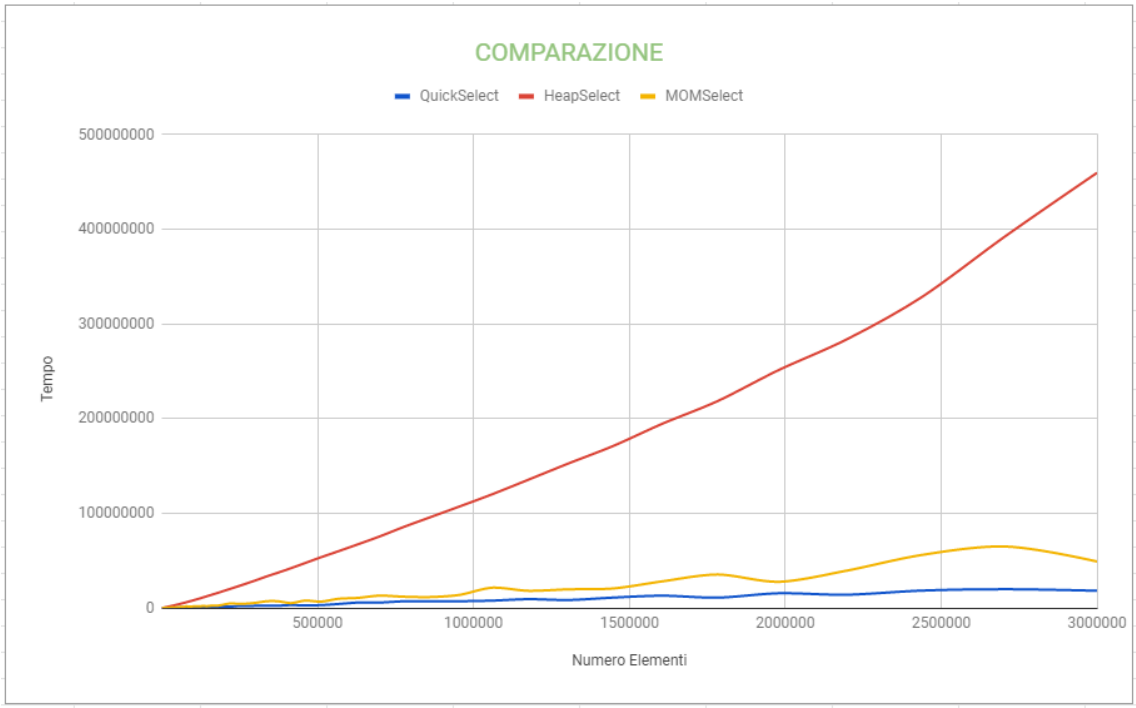

Figura 12: Grafo di tutti gli algoritmi in scala lineare

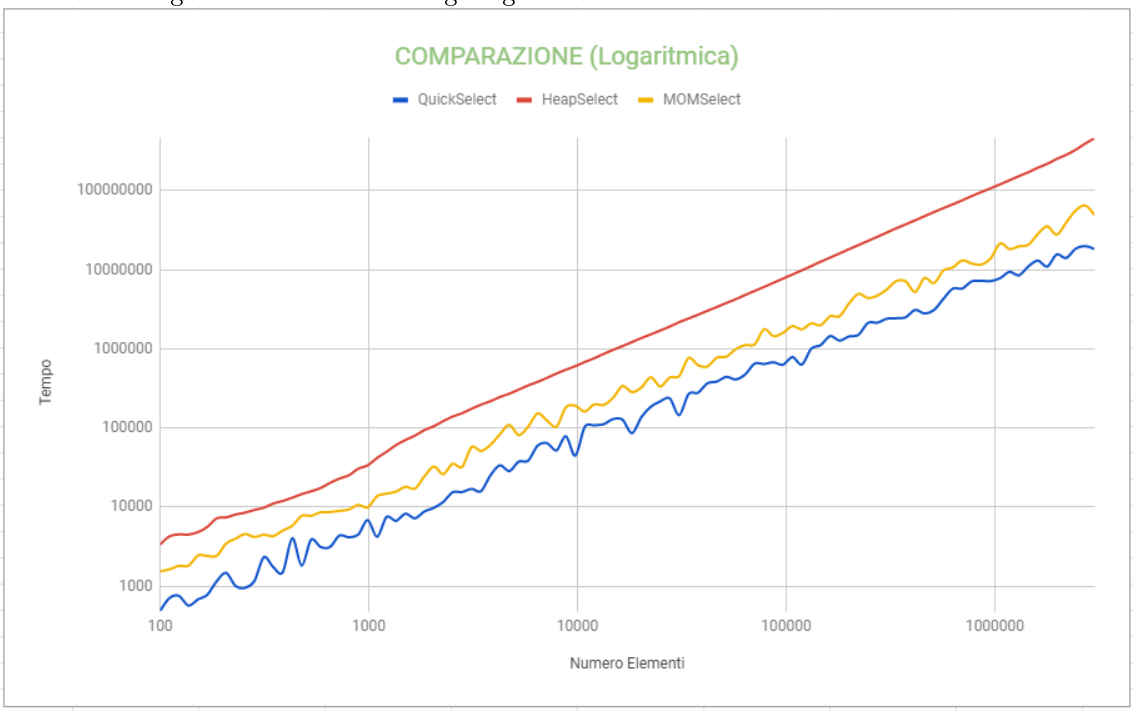

Figura 13: Grafo di tutti gli algoritmi in scala logaritmica