Università degli studi di Udine

Docenti *Piazza Carla* e *Puppis Gabriele*

Relazione di Laboratorio di Algoritmi e Strutture Dati

Anno accademico *2021-2022*

0. COMPONENTI DEL GRUPPO

Carpi Giulia	153207	153207@spes.uniud.it
Gerotto Alessandro	153736	153736@spes.uniud.it
Marchiol Pietro	152488	152488@spes.uniud.it
Mazzega Gabriele	152937	152937@spes.uniud.it

1. INTRODUZIONE

1.1 Obiettivo

In questo progetto ci poniamo come obiettivo l'implementazione, l'analisi e il confronto dei tempi di esecuzione medi di tre algoritmi di selezione (dove con il termine "selezione" si intende il calcolo del k-esimo più piccolo elemento del vettore, ovvero quello che finirebbe in posizione k se il vettore venisse ordinato). I tre algoritmi considerati sono: QuickSelect, HeapSelect e Medians-of-medians select.

1.2 Realizzazione

I tre algoritmi verranno implementati utilizzando il linguaggio C (scelto perché rinomato per la sua efficienza e migliore gestione della memoria rispetto agli altri linguaggi, che consentirà di ottenere risultati svincolati da fattori che non siano l'algoritmo in sé) e, per il calcolo dei tempi medi, realizzeremo un programma apposito, anch'esso in linguaggio C. Per la raccolta dei dati sperimentali sono stati utilizzati in input vettori di dimensione variabile, più precisamente è stata calcolata una funzione esponenziale per 100 dimensioni diverse, comprese tra 100 e 5000000.

Successivamente, per ognuno degli algoritmi, verrà effettuata un'analisi dei tempi di esecuzione in due versioni: tabellare e grafica. La prima per poter leggere in dettaglio i risultati ottenuti; la seconda per osservare qualitativamente l'andamento generale dell'algoritmo preso in considerazione.

Le tabelle saranno costruite utilizzando 15 campioni casuali, sorteggiati tra le dimensioni disponibili dei vettori. Conterranno i seguenti campi:

- la dimensione del vettore;
- il tempo trascorso dall'inizio alla fine dell'esecuzione dell'algoritmo (in nanosecondi);
- il numero di iterazioni dell'algoritmo effettuate prima del raggiungimento del tempo minimo Tmin[1], calcolato in precedenza;
- il tempo medio di esecuzione dell'algoritmo su un input della dimensione sorteggiata;

Questi ultimi tre valori vengono calcolati tramite la seguente formula:

$$x_{medio\ complessivo} = \frac{\sum_{i=1}^{1000} x_i}{1000},$$

$$\forall x \in \{tempo\ trascorso, n°\ iterazioni, tempo\ esecuzione\ medio\}$$

I grafici verranno realizzati con Excel, in scala lineare e logaritmica.

In conclusione, verranno analizzati i dati degli algoritmi singolarmente; successivamente verranno comparati tra loro evidenziando eventuali differenze nelle rispettive esecuzioni.

1. $T_{min} = R\left(\frac{1}{E} + 1\right)$

2. ALGORITMO QUICKSELECT

2.1 Descrizione

Quick Select è un algoritmo di selezione, basato sull'algoritmo Quick Sort.

2.1.1 Quick Sort

Basato sul paradigma "divide et impera", Quick Sort è un algoritmo di ordinamento il cui tempo di esecuzione nel caso peggiore è pari a $\Theta(n^2)$, con un array di input di n numeri. Nonostante la sua lentezza nel caso peggiore, è mediamente molto efficiente, infatti spesso è considerato tra le soluzioni migliori per effettuare un ordinamento, avendo caso migliore e medio pari a $\Theta(n * \log(n))$.

L'implementazione dell'algoritmo consiste nel considerare inizialmente l'ultimo elemento del vettore come primo pivot (chiamato anche perno), partizionando l'array intorno ad esso. Quest'ultima operazione viene effettuata confrontando il perno con gli altri elementi, posizionandoli alla sua sinistra se minori o alla sua destra se maggiori. A questo punto, avremo il perno nella sua posizione definitiva e si effettuerà una chiamata ricorsiva sulla parte del vettore alla sua sinistra e una sulla parte alla sua destra.

2.1.2 Quick Select

Quick Select è un algoritmo di selezione che si basa su Quick Sort. Come in esso, l'idea di base è partizionare ricorsivamente l'array di input. Diversamente da Quick Sort però, Quick Select opera soltanto su un lato della partizione, in base alla posizione del perno: se la posizione cercata è minore di quella del perno, verrà effettuata la chiamata a sinistra, al contrario, se maggiore, verrà effettuata a destra.

2.2 Implementazione

Il seguente codice per Quick Select restituisce il k-esimo elemento più piccolo dell'array array[i…j].

```
32  int quickSelect(int *array, int i, int j, int k) {
33
34      if(k > 0 && k <= j-i+1){
35
36          int pivot = partition(array, i, j);
37
38          if(pivot-i == k-1) {
39              return array[pivot];
40          }
41          else if(pivot-i > k-1) {
42              quickSelect(array, i, pivot-1, k);
43          }
44          else{
45              quickSelect(array, pivot+1, j, k-pivot+i-1);
46          }
47      }
48      else {
49          return INT_MIN;
50      }
51  }
```

Dopo l'esecuzione di "partition", eseguito alla riga 36, la variabile "pivot" contiene la posizione che avrebbe il valore array[j] (l'ultimo elemento), se il vettore fosse ordinato; inoltre il vettore chiamato "array" è diviso in due sotto-vettori, array[i…pivot-1] ed

array[pivot+1…j], tali che ogni elemento di array[i…pivot-1] è minore o uguale ad array[pivot], che a sua volta è minore a array[pivot+1…j]

$$\forall x \in array[i \dots pivot - 1], \forall y \in array[pivot + 1 \dots j]$$

$$\{x \leq array[piuvot] < y\}$$

La riga 38 controlla se la posizione trovata è la stessa di quella cercata (ovvero k), terminando la ricerca e restituendo il valore salvato nella posizione trovata; altrimenti viene effettuata una chiamata ricorsiva a sinistra (riga 41) o a destra (riga 44) del vettore, in base al valore che ha assunto la variabile "pivot".

2.3 Tempi

2.3.1 Tempi previsti

L'equazione di complessità dell'algoritmo Quick Select risulta:

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

dove m dipende dalla posizione che assumerebbe il pivot se il vettore fosse ordinato e $1 \leq m \leq n$.

Il caso migliore e quello medio presentano una complessità pari a $\Theta(n)$, mentre quello peggiore risulta $\Theta(n^2)$

2.3.2 Tempi analizzati

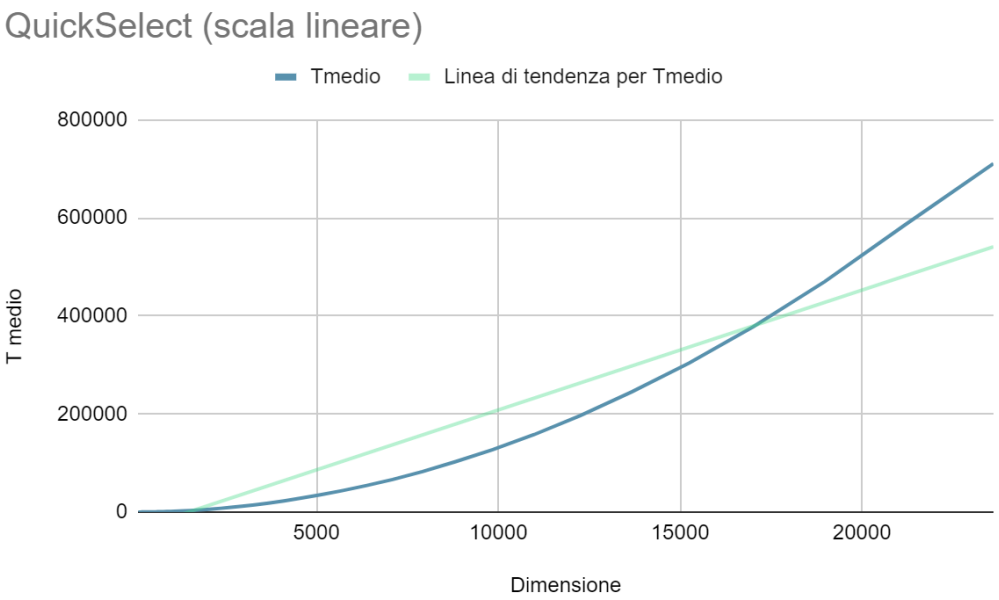
Il programma è stato eseguito per un'ora; tempo che si è ritenuto garantisse una stima adeguata dell'andamento dei tempi di esecuzione.

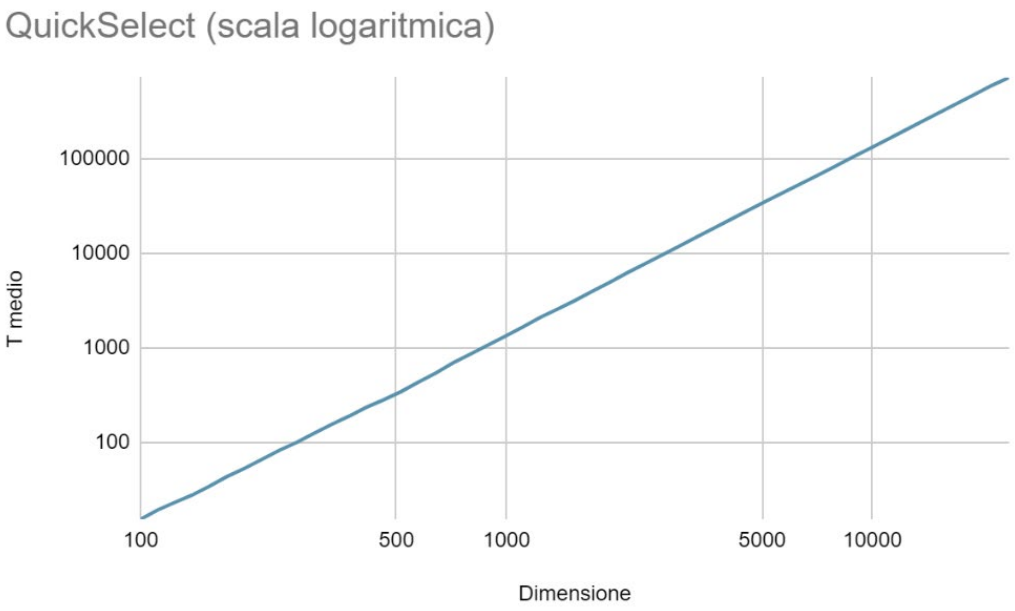
Dai dati raccolti è possibile osservare come il numero di iterazioni effettuate in un lasso di tempo pari a T_{min} si normalizzi a 1 dopo poche esecuzioni, considerando il numero totale di esse che il programma effettua.

Inoltre, come si può osservare dalle tabelle poste alla fine della relazione, in un'ora di tempo l'algoritmo itera attraverso poche dimensioni diverse del vettore (52/100 totali). Questo risultato soddisfa le aspettative, dato che la complessità dell'algoritmo Quick Select nel caso peggiore è pari a $\Theta(n^2)$, come sopra riportato.

N° elementi	Tempo trascorso	N° di iterazioni	Tempo medio di esecuzione
100	64,361	3,914	15,620
214	83,193	1	83,193
332	155,539	0,999	155,539
414	237,046	1	237,046
641	544,085	1	544,085
992	1332,720	1	1332,720
1377	2560,563	1	2560,563
1912	4912,811	0,999	4912,811
2960	11783,702	1	11783,702
6362	53771,236	1	53771,236
8831	102900,543	1	102900,543
10989	158014,986	1	158014,986
13673	245204,749	1	245204,749
17014	377621,133	0,999	377621,133
23616	710896,852	1,022	710896,852

2.4 Grafici


QuickSelect (scala lineare)
Tmedio
Linea di tendenza per Tmedio
800000
600000
400000
200000
0
T medio
5000
10000
15000
20000
Dimensione


QuickSelect (scala logaritmica)
100000
10000
1000
100
T medio
100
500
1000
5000
10000
Dimensione

3. HEAPSELECT

3.1 Descrizione

3.1.1 Heap

La heap è un albero binario quasi completo (ovvero tutti i livelli dell'albero sono completi a parte l'ultimo, che è riempito da sinistra verso destra), dove ogni nodo ha una priorità e il genitore ha sempre priorità maggiore o uguale al figlio (nel caso di max-heap), altrimenti ha priorità minore o uguale (nel caso di min-heap). Questa struttura dati viene memorizzata tramite un vettore V sovradimensionato, contenente le chiavi che la costituiscono. É caratterizzata inoltre due attributi: lunghezza[V] (che indica la dimensione del vettore) e heapsize[V](che indica il numero di elementi della heap effettivamente presenti nel vettore).

Infine, essendo un albero binario, ogni nodo costituente la heap ha:

- priorità o chiave
- puntatori al genitore, al figlio sinistro ed al figlio destro

3.1.2 Heap Select

L'algoritmo Heap Select utilizza due min-heap denominate H1 e H2. La prima heap H1 è costruita usando il vettore dato in input e non viene mai modificata durante l'esecuzione dell'algoritmo; la seconda heap H2 all'inizio contiene solamente un nodo, che è la radice di H1. All'i-esima iterazione della procedura, con i che va da 1 a k-1, l'algoritmo estrae la radice di H2, che corrisponde a un nodo x-i in H1, e reinserisce in H2 i nodi figli (sinistro e destro) di x-i nella heap H1. Dopo k-1 iterazioni la radice di H2 corrisponderà al k-esimo più piccolo elemento del vettore che è stato fornito in input. L'algoritmo ha complessità $\Theta(n + k * log(k))$ sia nel caso peggiore, che nel caso medio.

3.2 Implementazione

Nell'implementazione dell'algoritmo sono state utilizzate procedure (oltre a quella principale spiegata di seguito) congruenti con quelle viste nella parte teorica del corso, quali:

- heapify: assume che il figlio sinistro e destro del nodo preso in considerazione siano radici di heap, scambiandole eventualmente con il nodo stesso, in modo da ristabilire la proprietà delle heap;
- buildHeap: dato un vettore, scambia i nodi in modo tale da avere una heap;
- extractMin: estrae e rimuove il minimo dalla heap (nel caso di minHeap si tratta della radice);
- heapInsert: inserisce un nodo nella heap, eventualmente scambiandolo con il genitore, per ristabilire la proprietà delle heap

```
172 int heapSelect(Node *h1, Node *h2, int size1, int k) {
173
174     int size2 = 0;
175     Node leftChild, rightChild;
176
177     h2[size2] = h1[0];
178
179     size2++;
180
181     for(int i=0; i<k-1; i++) {
182
183
184         Node min = extractMin(h2, &size2);
185
186         if(left(min.index) < size1){
187
188             leftChild = h1[left(min.index)];
189             heapInsert(h2, leftChild, &size2);
190         }
191
192         if(right(min.index) < size1){
193
194             rightChild = h1[right(min.index)];
195             heapInsert(h2, rightChild, &size2);
196         }
197
198     }
199
200     return h2[0].key;
201 }
```

Nella riga 177 si può notare come la radice di H2 sia la medesima di quella di H1, osservato che, essendo H1 una minHeap, la sua radice è il minimo valore fra quelli inseriti.

Per k-1 volte, alla riga 184 viene estratto il valore minimo della minHeap H2, che corrisponderà alla sua radice, inserendo successivamente, solo nel caso in cui esistano in H1, il figlio sinistro e quello destro del nodo appena estratto.

Questo garantisce che, dopo k-1 iterazioni, la radice di H2 contenga il k-esimo elemento più piccolo del vettore.

3.3 Tempi

3.3.1 Tempi previsti

L'equazione di complessità dell'algoritmo Heap Select risulta:

$$T(n,k) = \begin{cases} \Theta(1), & n \leq 1, \\ k * \log(k) + \Theta(n), & n > 1 \end{cases}$$

La complessità, data dalla somma dei costi delle procedure utilizzate, risulta essere $O(k * \log(k) + n)$ in tutti i casi, dove k rappresenta il k-esimo elemento più piccolo.

Il fattore $k * \log(k)$ dipende dal fatto che le procedure di estrazione e inserimento vengano effettuate su H2, la quale avrà sempre dimensione pari a $c * k$,dove c è una costante.

3.3.2 Tempi analizzati

Il programma è stato eseguito per un'ora; tempo che si è ritenuto garantisse una stima adeguata dell'andamento dei tempi di esecuzione.

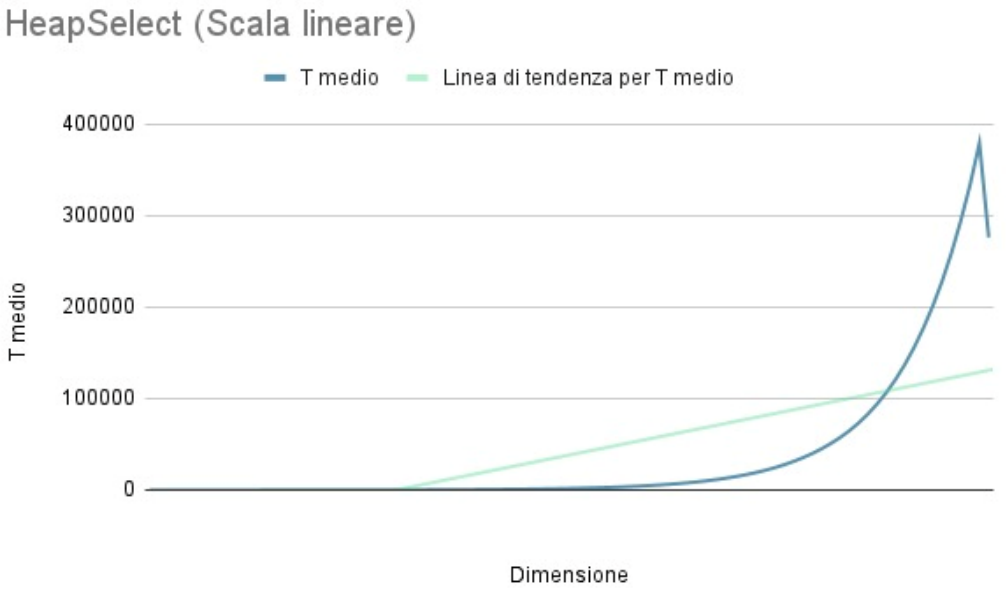
Dai dati raccolti è possibile osservare come il numero di iterazioni effettuate in un lasso di tempo pari a T_{min} si normalizzi a 1 dopo poche esecuzioni, considerando il numero totale di esse che il programma effettua. Risulta però migliore di Quick Select, osservando che il numero di cambiamenti di dimensione, prima che il valore delle iterazioni si normalizzi a 1, sono circa 10 in più rispetto all'algoritmo sopracitato.

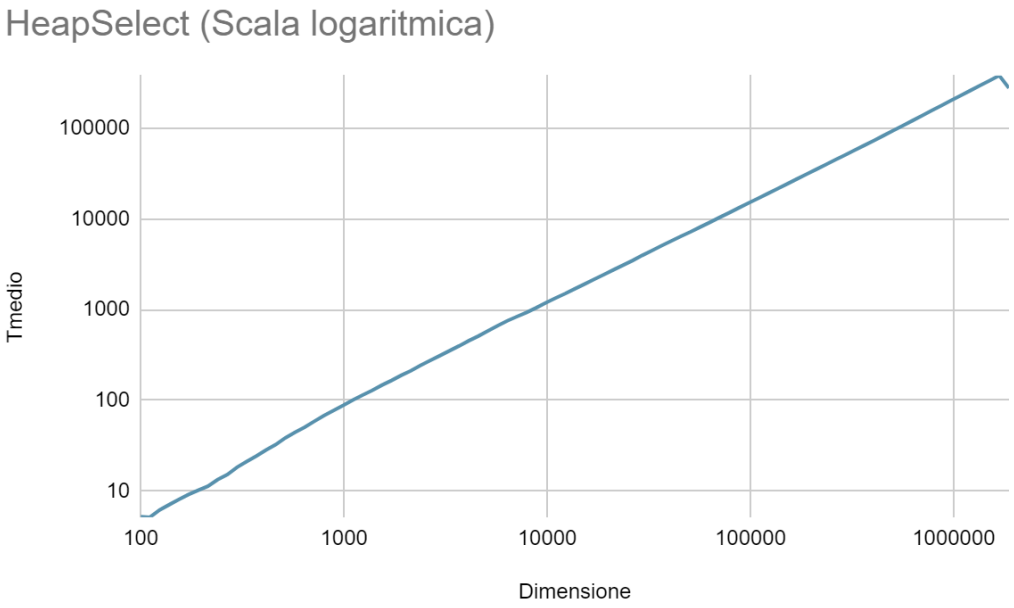
Inoltre, come si può osservare nelle tabelle poste sotto, in un'ora di tempo l'algoritmo itera attraverso 91/100 dimensioni diverse del vettore, che rappresentano la quasi totalità.

Questo risultato soddisfa le aspettative, dato che la complessità dell'algoritmo Heap Select è minore di quella di Quick Select; dunque, nello stesso tempo è riuscito a terminare quasi il doppio di cambiamenti di dimensione del vettore rispetto al precedente.

N° elementi	Tempo trascorso	N° di iterazioni	Tempo medio di esecuzione
100	51,291	10,724	5,211
214	57,153	4,959	11,286
414	55,686	1,995	28,313
641	50,477	1	50,477
992	87,822	1	87,822
1377	128,242	1	128,242
1912	188,171	1	188,171
2960	309,983	1	309,983
6362	750,703	1	750,703
15253	1907,830	1	1907,830
32780	4466,715	1	4466,715
63153	9151,311	1	9151,311
325357	56936,162	1	56936,162
1082602	228333,593	1	228333,593
1869781	276132,450	1	276132,450

3.4 Grafici


HeapSelect (Scala lineare)
T medio
Linea di tendenza per T medio
400000
300000
200000
100000
0
T medio
Dimensione


HeapSelect (Scala logaritmica)
100000
10000
1000
100
10
Tmedio
100
1000
10000
100000
1000000
Dimensione

4. ALGORITMO MEDIAN-OF-MEDIANS SELECT

4.0 Versione implementata

La versione dell'algoritmo implementata è quella definita "quasi in-place", ovvero non viene istanziato un nuovo vettore in cui salvare i mediani, ma viene fatta la ricorsione sul vettore iniziale, spostando eventualmente gli elementi.

4.1 Descrizione

Come per gli algoritmi precedenti, l'idea alla base di Median of Medians è quella di partizionare ricorsivamente l'array in input; la peculiarità di questo algoritmo è quella di garantire una buona ripartizione dell'array, in modo tale da non ricadere mai nel caso pessimo di Quick Sort.

1. inizialmente si divide l'array in blocchi di dimensione 5, ottenendo n/5 blocchi;
2. si trovano n/5 mediani dei blocchi, che vengono spostati nelle prime n/5 posizioni del vettore, continuando ad iterare ricorsivamente fino a quando rimane 1 solo blocco di valori;
3. trovato il mediano dei mediani, si calcola la posizione che avrebbe se il vettore fosse ordinato (pivot)
4. se il pivot corrisponde alla chiave k passata, si è trovato l'elemento cercato, altrimenti si continua con una ricorsione a sinistra o destra in base al valore di esso

N.B.: Dalle nozioni teoriche si è osservato che nel calcolo della complessità della procedura, con blocchi di dimensione $dim \geq 5$, la serie geometrica che si ricava è di ragione < 1, dunque eliminabile nel calcolo. Per questo si è scelto una dimensione effettivamente pari a 5.

4.2 Implementazione

```
298  int medianOfMedians(int* arr, int l, int r, int k, int size)
299  {
300      if (k > 0)
301      {
302          int n = r - l + 1, pos, i, median;
303          int medOfMeds;
304
305          for (i = 0; i < n / blockSize; i++) {
306              median = findMedian(arr + l + i * blockSize, blockSize);
307              swap(arr + l + i * blockSize + median, arr + i + l);
308
309          }
310
311          if (i * 5 < n) {
312              median = findMedian(arr + l + i * blockSize, n % blockSize);
313              swap(arr + l + i * blockSize + median, arr + i + l);
314
315              i++;
316          }
317
318          if (i == 1) {
319
320              medOfMeds = arr[l + i - 1];
321          }
322
323          else {
324
325              return medOfMeds = medianOfMedians(arr, l, l + i - 1, k, size);
326          }
327
```

Dalla riga 305, possiamo osservare come venga effettuata la divisione del vettore in blocchetti da 5. Per ognuno, viene trovato l'indice del mediano (utilizzando la procedura "findMedian") e viene spostato nelle prime posizioni, in modo tale che quando verrà fatta la chiamata ricorsiva per trovare il mediano dei mediani, la si farà solo sulle prime n/5 posizioni.

La procedura findMedian viene implementata ordinando[1] il blocco da 5 elementi, trovando il mediano, e restituendo la posizione del mediano nel vettore totale passato (il quale non è ordinato).

La condizione alla riga 311 effettua il controllo e la ricerca del mediano nel caso in cui l'ultimo blocco non abbia effettivamente 5 valori, ma $n\ mod(5)$.

La porzione di codice alla riga 318 controlla il caso base della ricorsione; infatti, se è stata eseguita una sola iterazione, allora è presente solo un blocco; questo significa che il suo mediano è il mediano dei mediani. Al contrario viene continuata la ricerca ricorsiva sulle prime i posizioni, dove i rappresenta il numero di mediani trovati fino a quel momento.

```
328         pos = partition(arr, l, size, medOfMeds);
329         if (pos - l == k - 1) {
330
331             return medOfMeds;
332         }
333
334         else if (pos - l > k - 1) {
335             return medianOfMedians(arr, l, pos - 1, k, pos);
336         }
337         else {
338             return medianOfMedians(arr, pos + 1, size, k - pos + l - 1, size);
339         }
340     }
```

Alla riga 328 viene calcolata e salvata nella variabile "pos" la posizione che avrebbe il mediano dei mediani se il vettore fosse ordinato.

Successivamente, si confronta il valore di "pos" con la posizione cercata(k):

- i due valori combaciano: il mediano dei mediani calcolato è proprio il valore che cercavamo
- pos > k: viene effettuata la chiamata ricorsiva a sinistra, perché il valore cercato ha posizione minore di quella del mediano, risultando quindi più piccolo di esso
- pos < k: viene effettuata la chiamata ricorsiva a destra, perché il valore cercato ha posizione maggiore di quella del mediano, risultando quindi più grande di esso

1. il blocco viene ordinato da una procedura esterna alla findMedian; dunque, risulterà ordinato solo localmente alla procedura. L'algoritmo di ordinamento scelto è Insertion Sort, poiché lavorando con blocchi di dimensione costante, il costo è trascurabile.

4.3 Tempi

4.3.1 Tempi previsti

L'equazione di complessità dell'algoritmo Median of Medians Select risulta:

$$T(n) = \begin{cases} \Theta(1), & n \leq 1, \\ \mathrm{T}\left(\frac{\mathrm{n}}{5}\right) + T\left(\frac{3}{4}n\right) + \Theta(n), & n > 1 \end{cases}$$

La soluzione dell'equazione è pari a $\Theta(n)$, poiché $\frac{1}{5} + \frac{3}{4} < 1$, dunque la serie geometrica è eliminabile nel calcolo.

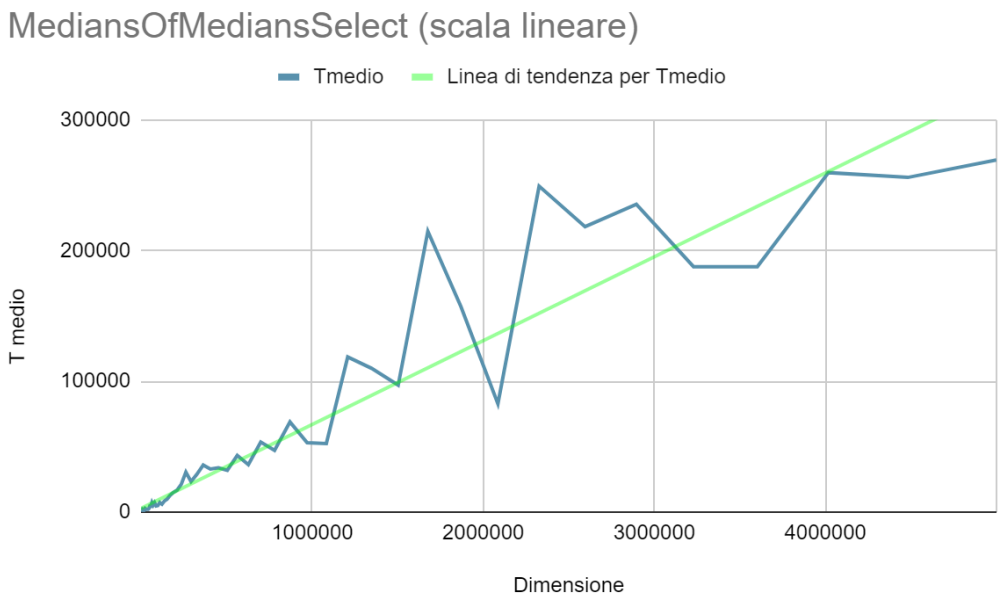
Secondo le stime della complessità, i tempi medi di esecuzione di Median of Medians dovrebbero essere i più bassi tra i tre algoritmi descritti.

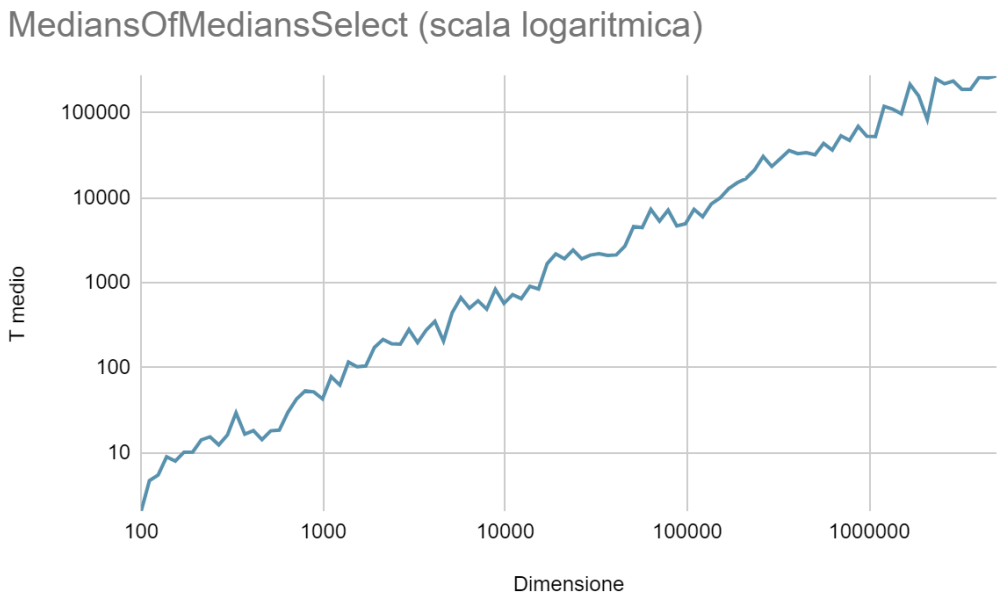
4.3.2 Tempi analizzati

Il programma è stato eseguito per un'ora; tempo che si è ritenuto garantisse una stima adeguata dell'andamento dei tempi di esecuzione. Come si può analizzare dalle tabelle sotto riportate (e dalla tabella sottostante), questo algoritmo risulta l'unico fra i tre ad avere completato le 100 esecuzioni di variazione della dimensione. È, inoltre, possibile osservare come il numero di iterazioni effettuate in un lasso di tempo pari a T_{min} si normalizzi a 1 dopo molte più esecuzioni rispetto agli algoritmi precedenti.

N° elementi	Tempo Trascorso	N° di iterazioni	Tempo medio di esecuzione
100	5,358	21,137	2,076
214	55,202	3,959	14,239
414	53,473	2,977	18,337
889	54,460	1,045	52,165
1714	105,377	1	105,377
4109	351,762	1	351,762
9851	571,815	1	571,815
23616	2422,745	1	2422,745
56615	4459,005	1	4459,005
135720	8450,542	1	8450,542
325357	28826,085	1	28826,085
970520	52955,227	1	52955,227
2085716	82873,885	1	82873,885
4018290	259619,419	1	259619,419
5000000	269238,266	1	269238,266

4.4 Grafici





L'andamento generale della curva è corretto, prendendo in considerazione la complessità dell'algoritmo, ma non risulta essere precisa, presentando alcuni punti con dei picchi anomali.

5. PROGRAMMA PER LA RACCOLTA DEI DATI

5.1 Generazione dei numeri casuali

Per la generazione dei numeri casuali sono state usate due procedure principali, che forniscono due metodi distinti per generare un numero.

```
uint64_t xoshiro256ss(struct xoshiro256ss_state* state)
{
    uint64_t* s = state->s;
    uint64_t const result = rol64(s[1] * 5, 7) * 9;
    uint64_t const t = s[1] << 17;

    s[2] ^= s[0];
    s[3] ^= s[1];
    s[1] ^= s[2];
    s[0] ^= s[3];

    s[2] ^= t;
    s[3] = rol64(s[3], 45);

    return result;
}
```

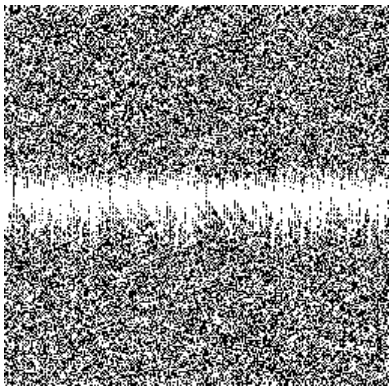
La procedura determina un numero, generato in base allo stato "s" fornito, in modo pseudo casuale. Una volta determinato il numero, viene modificato lo stato per la generazione dei numeri successivi.

Il generatore xoshiro256** è stato scelto per due motivi:

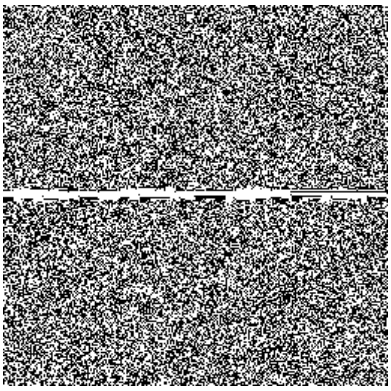
- l'ottima velocità di generazione garantisce una rapida popolazione dell'array con conseguente riduzione dei tempi di esecuzione totali
- la solidità dei principali test di "fitness" per PRNG garantisce una generazione priva di errori.

Sono inoltre stati presi in considerazione due punti:

- La generazione di questo algoritmo presenta una zona chiamata "zeroland" abbastanza vasta ovvero un possibile cluster di risultati simili in cui la maggior parte dei bit è zero (come visto nel randogramma sotto). Questo problema è stato preso in considerazione ma è stato valutato non influente sul risultato in quanto improbabile e non un aumento di complessità per un algoritmo di ordinamento.



Il nostro generatore



Jenkins's JSF PRNG

- Inoltre, visto che tutti i PRNG sono periodici, è stata considerata la possibilità che la sequenza di numeri generati, o periodo, si ripeta. Per questo algoritmo il periodo è tuttavia sufficientemente grande, perché questa occorrenza si verifichi nelle nostre simulazioni.

```
uint64_t next(uint64_t x) {
    uint64_t z = (x += 0x9e3779b97f4a7c15);
    z = (z ^ (z >> 30)) * 0xbf58476d1ce4e5b9;
    z = (z ^ (z >> 27)) * 0x94d049bb133111eb;
    return z ^ (z >> 31);
}
```

Il generatore xoshiro256** necessita di uno stato con tutti i campi non nulli. Per questo scopo utilizziamo il PRNG Splitmix64 come generatore, e questo perché l'inizializzazione deve essere effettuata con un generatore molto diverso da quello inizializzato, in modo da evitare correlazione nei semi di generazione simili.

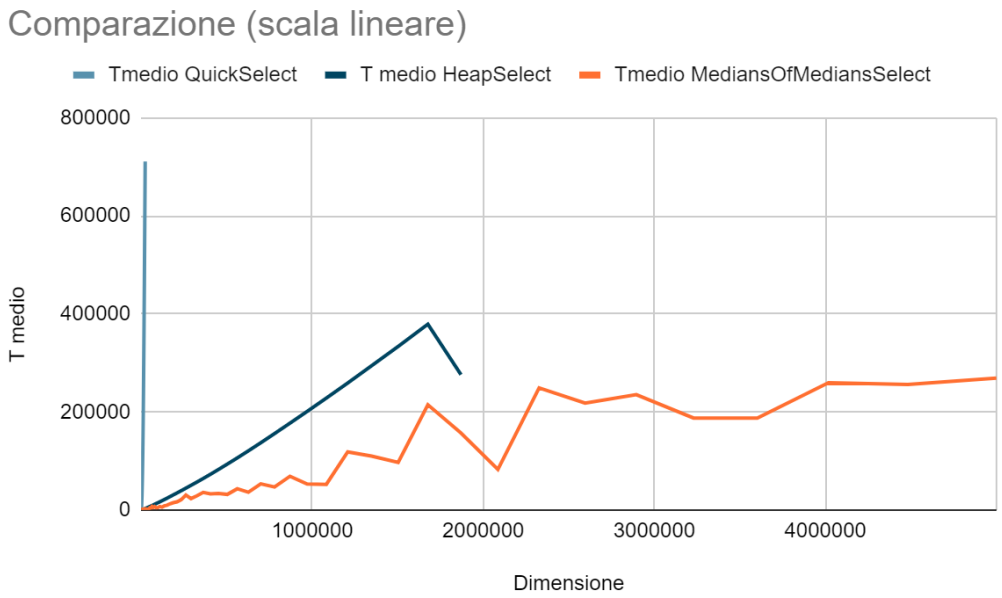
5.2 Rilevazione dei tempi

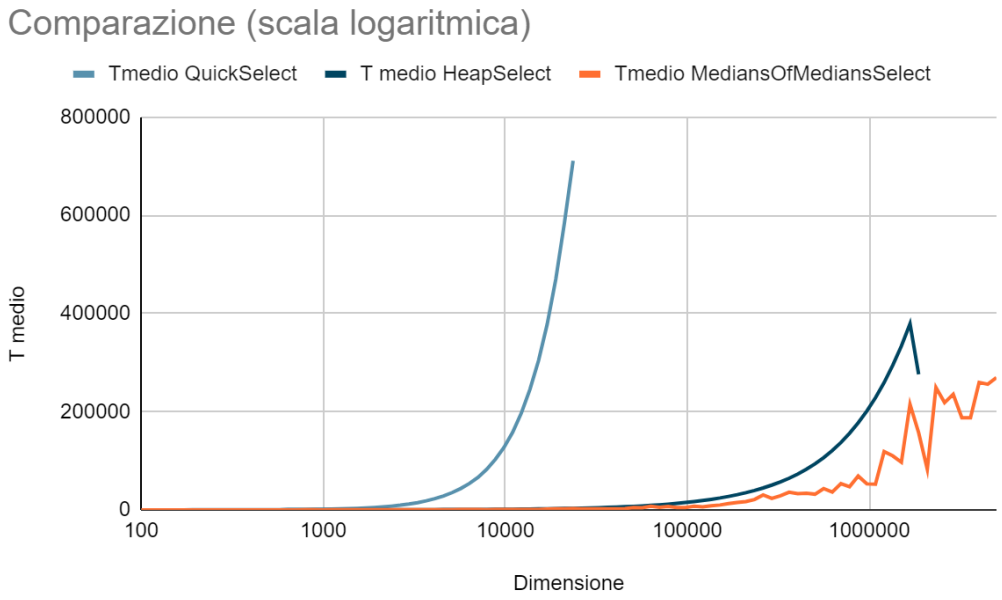
Per la rilevazione dei tempi sono state utilizzate le due funzioni fornite dal Professore, "duration" e "getResolution". Nel main, sono stati utilizzati due for: il primo itera tra tutte e 100 le dimensioni del vettore (calcolate con la seguente formula: $size = A * 2^{B*i}$, dove A e B sono opportune costanti con valori iniziali pari a $A = 100, B = \frac{\log_2 \frac{5000000}{A}}{99}$, poiché la dimensione del vettore deve variare tra 100 e 5000000), il secondo serve ad eseguire l'algoritmo fino a quando il tempo di esecuzione non supera una certa soglia (Tmin), anch'essa calcolata con una formula.

```
73      for (int i = 0; i < 100; ++i) {
74          size = A * pow(2, B * i);
75          for (int j = 0; j < POOL_SIZE; j++) {
76              cur_time = 0;
77              iter = 0;
78              do {
79                  setState(&state, key);
80
81                  buildArray(&state, arr, size);
82
83                  clock_gettime(CLOCK_MONOTONIC, &start);
84
85                  quickSelect(arr, 0, size - 1, size / 2);
86
87                  clock_gettime(CLOCK_MONOTONIC, &end);
88
89                  cur_time = cur_time + duration(start, end);
90
91                  iter++;
92
93              } while (cur_time < tmin);
94
95              fprintf(file, "Size: %d, Cur time: %f, Iter: %f\n", size, cur_time, iter);
96              fprintf(file, "%f\n\n", cur_time / iter);
97          }
98          fprintf(file, "\n\n");
99      }
```

6.ANALISI DEI DATI RACCOLTI

6.1 Grafici comparativi





6.2 Analisi dei dati

Come si può osservare dai grafici e come appreso dall'analisi dei singoli, Quick Select risulta il peggiore tra i tre algoritmi di selezione. Pur avendoli eseguiti tutti e tre per lo stesso lasso di tempo, è quello che effettua meno esecuzioni, dato il fatto che impiega più tempo ad effettuarne una, rispetto agli altri. Inoltre, data la sua complessità dell'ordine n^2, si può evincere come la curva cresca molto velocemente, al contrario degli altri che presentano una curva meno pendente.

Per quanto riguarda Heap Select, risulta l'algoritmo medio fra i tre, riuscendo ad effettuare iterazioni con dimensione crescente del vettore fino ad un valore finale pari a 1869781, il quale rappresenta il 91/100 delle dimensioni totali disponibili.

Nel primo grafico si osserva una sua discesa inaspettata, poiché per l'ultimo campione, l'algoritmo non è riuscito ad effettuare le 1000 iterazioni dell'ultima dimensione, usate per calcolare con una buona stima il tempo medio.

L'ultimo algoritmo analizzato è Median of Medians Select, il quale si riscontra essere il migliore, come rilevato dall'analisi teorica delle complessità. Il suo andamento risulta il più preciso, avendo completato tutte le dimensioni del vettore disponibili. Inoltre, la sua curva non supera mai nessuna delle altre due, identificandolo come il migliore algoritmo su tutte le dimensioni analizzate.

7. STRUMENTI UTILIZZATI

Ambiente per i codici:

- Virtual machine VMware Workstation 16 Player
- SO: Linux Ubuntu 20.04
- RAM: 4 GB
- CPU: Intel core I7 8550 @ 1.80 GHz

Ambiente di esecuzione dei codici:

- SO: Linux Ubuntu
- RAM: 16GB
- CPU: Intel core I5 10600kf

Ambiente di elaborazione dei dati:

- Microsoft Word
- Microsoft Excel
- Google docs

TABELLE CON I DATI DEI TEMPI MEDI

QuickSelect		HeapSelect		MediansOfMediansSelect	
Size	Tmedio	Size	T medio	Size	Tmedio
100	15,62	100	5,211	100	2,076
111	19,461	111	5,144	111	4,719
124	23,51	124	6,199	124	5,516
138	28,012	138	7,067	138	9,029
154	34,77	154	8,066	154	8,038
172	43,971	172	9,156	172	10,2
192	53,553	192	10,201	192	10,259
214	66,47	214	11,286	214	14,239
239	83,193	239	13,354	239	15,462
267	100,97	267	15,21	267	12,443
298	126,246	298	18,288	298	16,282
332	155,696	332	21,131	332	29,608
371	190,643	371	24,319	371	16,624
414	237,046	414	28,313	414	18,337
461	283,078	461	32,45	461	14,315
515	346,494	515	38,574	515	18,203
574	436,488	574	44,416	574	18,532
641	544,085	641	50,477	641	30,174

715	697,329	715	58,687	715	43,19
797	863,405	797	67,811	797	53,779
889	1072,805	889	77,27	889	52,165
992	1332,72	992	87,822	992	43,193
1107	1661,726	1107	100,835	1107	78,574
1235	2097,037	1235	114,225	1235	62,73
1377	2560,563	1377	128,242	1377	116,944
1536	3156,892	1536	146,751	1536	103,119
1714	3948,323	1714	165,314	1714	105,377
1912	4917,696	1912	188,171	1912	173,704
2133	6178,297	2133	211,548	2133	215,676
2379	7665,423	2379	242,341	2379	192,422
2654	9513,996	2654	274,229	2654	190,028
2960	11783,702	2960	309,983	2960	282,003
3302	14733,073	3302	351,105	3302	198,12
3684	18330,293	3684	397,144	3684	278,204
4109	22795,672	4109	452,534	4109	351,762
4584	28357,617	4584	509,802	4584	206,635
5113	35189,412	5113	581,258	5113	445,04
5704	43548,761	5704	664,676	5704	670,08
6362	53771,236	6362	750,703	6362	502,505
7097	66365,871	7097	836,051	7097	614,685
7917	82611,904	7917	929,483	7917	490,353
8831	102900,543	8831	1045,934	8831	835,905
9851	127248,93	9851	1189,155	9851	571,815
10989	158014,986	10989	1339,747	10989	725,851
12258	196780,977	12258	1499,468	12258	651,682
13673	245204,749	13673	1691,525	13673	909,309
15253	304122,92	15253	1907,83	15253	844,385
17014	377998,736	17014	2150,302	17014	1677,08
18979	470174,627	18979	2421,653	18979	2189,654
21171	585045,871	21171	2735,83	21171	1922,415
23616	710896,852	23616	3081,369	23616	2422,745
		26343	3473,484	26343	1913,887
		29386	3957,458	29386	2115,968
		32780	4466,715	32780	2193,802
		36565	5052,089	36565	2094,985
		40788	5698,317	40788	2127,869
		45499	6420,998	45499	2697,694
		50753	7191,333	50753	4585,247
		56615	8113,1	56615	4459,005
		63153	9150,114	63153	7285,777
		70446	10348,041	70446	5302,559

78582	11668,996	78582	7144,315
87657	13196,925	87657	4663,63
97780	14890,34	97780	4944,962
109073	16817,16	109073	7304,844
121669	18974,916	121669	5981,6
135720	21461,236	135720	8450,542
151394	24216,267	151394	9897,51
168878	27408,549	168878	12747,271
188382	30968,2	188382	15003,733
210137	34964,41	210137	16786,976
234405	39511,437	234405	21168,412
261476	44641,075	261476	30526,857
291673	50453,98	291673	23303,266
325357	56936,162	325357	28826,085
362932	64369,902	362932	35935,551
404846	72887,065	404846	32903,872
451600	82620,314	451600	33804,922
503754	93739,307	503754	31898,51
561931	106283,624	561931	43358,905
626826	120738,225	626826	36280,556
699216	137184,793	699216	53459,844
779966	155989,726	779966	47167,152
870042	177217,503	870042	68911,663
970520	201182,385	970520	52955,227
1082602	228333,593	1082602	52406,955
1207629	259115,231	1207629	118582,758
1347093	294325,98	1347093	109946,262
1502665	333743,293	1502665	97044,029
1676202	379088,115	1676202	214775,319
1869781	276132,45	1869781	157085,645
		2085716	82873,885
		2326588	249215,572
		2595278	218285,612
		2894998	235391,963
		3229331	187688,2
		3602275	187618,815
		4018290	259619,419
		4482348	255953,31
		5000000	269238,266

8. BIBLIOGRAFIA

- https://vigna.di.unimi.it/papers.php#BlVSLPNG
- https://www.pcg-random.org/posts/xoshiro-repeat-flaws.html
- https://vigna.di.unimi.it/
- https://prng.di.unimi.it/
- https://rosettacode.org/wiki/Pseudo-random_numbers/Splitmix64
- Introduzione agli algoritmi e strutture dati (T. H. Cormin, C. E. Leiserson, R. L. Rivest, C. Stein) McGraw-Hill - 3° edizione
- Appunti di teoria della Professoressa Carla Piazza
- https://www.geeksforgeeks.org/