|  |
| --- |
| Corso “Algoritmi e strutture dati” |
| Implementazione ed analisi di tre algoritmi di selezione |
| Relazione del 1° progetto di Laboratorio |

|  |
| --- |
| Francesco Bombassei De Bona – Cantarutti Andrea  A.A. 2019/2020 |

# Sommario

[1 Introduzione 4](#_Toc40884793)

[2 Quick select 4](#_Toc40884794)

[2.1 Obiettivo preposto 4](#_Toc40884795)

[2.2 Funzioni implementate 4](#_Toc40884796)

[2.3 quickSelect 4](#_Toc40884797)

[2.3.1 Implementazione proposta 4](#_Toc40884798)

[2.3.2 Input 5](#_Toc40884799)

[2.3.3 Esecuzione 5](#_Toc40884800)

[2.3.4 Output 5](#_Toc40884801)

[2.4 partition 5](#_Toc40884802)

[2.4.1 Implementazione proposta 6](#_Toc40884803)

[2.4.2 Input 6](#_Toc40884804)

[2.4.3 Esecuzione 6](#_Toc40884805)

[2.4.4 Output 6](#_Toc40884806)

[2.5 Complessità temporale di Quick select 6](#_Toc40884807)

[2.5.1 Individuazione e analisi del caso base 7](#_Toc40884808)

[2.5.2 Individuazione e descrizione della relazione di ricorrenza 7](#_Toc40884809)

[2.5.3 Dimostrazione del caso pessimo mediante equazione di ricorrenza 7](#_Toc40884810)

[2.5.4 Dimostrazione del caso medio mediante equazione di ricorrenza 8](#_Toc40884811)

[2.5.5 Conclusioni 8](#_Toc40884812)

[2.6 Complessità spaziale di Quick select 8](#_Toc40884813)

[3 Heap Select 9](#_Toc40884814)

[3.1 Cenni sulla binary heap 9](#_Toc40884815)

[3.2 Obiettivo preposto 9](#_Toc40884816)

[3.3 Funzioni implementate 9](#_Toc40884817)

[3.4 Heap dinamica 9](#_Toc40884818)

[3.5 Classe Pair 10](#_Toc40884819)

[3.6 HeapSelect 10](#_Toc40884820)

[3.6.1 Implementazione proposta 10](#_Toc40884821)

[3.6.2 Input 10](#_Toc40884822)

[3.6.3 Esecuzione 11](#_Toc40884823)

[3.6.4 Output 11](#_Toc40884824)

[3.7 select 11](#_Toc40884825)

[3.7.1 Implementazione proposta 12](#_Toc40884826)

[3.7.2 Input 12](#_Toc40884827)

[3.7.3 Esecuzione 12](#_Toc40884828)

[3.7.4 Output 12](#_Toc40884829)

[3.8 insert 12](#_Toc40884830)

[3.9 extract 13](#_Toc40884831)

[3.10 buildHeap 13](#_Toc40884832)

[3.11 Complessità temporale di heapSelect 13](#_Toc40884833)

[3.11.1 Complessità dei metodi di heapify 13](#_Toc40884834)

[3.11.2 Complessità del metodo di costruzione della heap 14](#_Toc40884835)

[3.11.3 Complessità del metodo insert 14](#_Toc40884836)

[3.11.4 Complessità del metodo extract 14](#_Toc40884837)

[3.11.5 Lunghezza della ArrayList h2 14](#_Toc40884838)

[3.11.6 Complessità del metodo select 15](#_Toc40884839)

[3.11.7 Complessità di HeapSelect 15](#_Toc40884840)

[3.12 Complessità spaziale di HeapSelect 16](#_Toc40884841)

[4 Median-of-medians select 16](#_Toc40884842)

[4.1 Obiettivo proposto 16](#_Toc40884843)

[4.2 Funzioni implementate 16](#_Toc40884844)

[4.2.1 Implementazione proposta 16](#_Toc40884845)

[4.2.2 Input 17](#_Toc40884846)

[4.2.3 Esecuzione 17](#_Toc40884847)

[4.2.4 Output 19](#_Toc40884848)

[4.3 medianOfMedians 19](#_Toc40884849)

[4.4 InsertionSort 19](#_Toc40884850)

[4.5 partition 19](#_Toc40884851)

[4.6 swap 19](#_Toc40884852)

[4.7 Complessità temporale di medianSelect 20](#_Toc40884853)

[4.7.1 Complessità di InsertionSort su gruppi di 5 elementi 20](#_Toc40884854)

[4.7.2 Individuazione delle mediane 20](#_Toc40884855)

[4.7.3 Complessità di medianOfMedians 20](#_Toc40884856)

[4.7.4 Complessità della chiamata ricorsiva di medianSelect 20](#_Toc40884857)

[4.7.5 Complessità totale della funzione medianSelect 21](#_Toc40884858)

[4.8 Complessità spaziale di medianSelect 21](#_Toc40884859)

[5 RACCOGLIMENTO DEI DATI 21](#_Toc40884860)

[5.1 Ottenimento della risoluzione di clock 22](#_Toc40884861)

[5.2 Warmup della JVM 23](#_Toc40884862)

[5.3 Modellazione del foglio Excel 23](#_Toc40884863)

[5.4 Ottenimento dei tempi di esecuzione 23](#_Toc40884864)

[5.4.1 Generazione dell’input 23](#_Toc40884865)

[5.4.2 Popolamento dell’input 24](#_Toc40884866)

[5.4.3 Esecuzione degli algoritmi 24](#_Toc40884867)

[6 Analisi dei dati 25](#_Toc40884868)

[7 Conclusioni 27](#_Toc40884869)

# Introduzione

Si propone un’analisi relativa a tre soluzioni implementative atte a risolvere il problema di selezione del k-esimo elemento più piccolo, in una struttura dati non ordinata di tipo array.

Gli algoritmi implementati e presi in analisi sono i seguenti:

* Quick select;
* Heap select;
* Median-of-medians select.

Nei capitoli successivi, vengono presentate le soluzioni implementative individuate e viene proposta un’analisi relativa all’andamento e alle differenze fra i tempi di esecuzione degli algoritmi.

Non viene, invece, approfondito il funzionamento di metodi “helper” preposti all’interazione con gli algoritmi, ma non strettamente legati al loro funzionamento.

# Quick select

## Obiettivo preposto

Sulla base di una variante del metodo di ordinamento “quick sort”, si sviluppa un algoritmo di selezione nel quale ogni chiamata ricorsiva su un intervallo [***i***, ***j***] del vettore fornito in input termina in tempo costante, ogniqualvolta il parametro ***k*** non sia contenuto nell’intervallo [***i***, ***j***].

## Funzioni implementate

* *quickSelect*
* *partition*

## quickSelect

Il funzionamento di ***quickSelect*** si dimostra intuitivo e, come per “quick sort”, si appoggia al calcolo di una partizione dell’input sfruttando la procedura ***partition***. In particolare, il compito di quest’ultima è il calcolo dell’indice di un elemento denominato “pivot”, impiegato per partizionare il vettore. La fondamentale caratteristica del pivot è, in particolare, quella di assumere un valore corrispondente all’indice che avrebbe all’interno del vettore ordinato. Successivamente, il pivot viene confrontato con ***k*** e, sulla base del risultato del confronto, l’algoritmo termina restituendo l’intero in posizione ***k***, oppure eseguendo una chiamata ricorsiva su una partizione dell’input.

### Implementazione proposta

*public static int* quickSelect(*int*[] array, *int* l, *int* r, *int* k) {  
 *if* (k < l || k > r)   
 System.*exit*(-1);

*int* pivot\_index = *partition*(array, l, r);

*if* (k == pivot\_index)  
 *return* array[k];

*else if*(k < pivot\_index)  
 *return quickSelect*(array, l, pivot\_index - 1, k);

*else  
 return quickSelect*(array, pivot\_index + 1, r, k);  
}

### Input

Gli argomenti della procedura sono i seguenti:

* ***array***: un vettore di interi, che si suppone essere ben formato e non vuoto;
* ***l***: indice della cella più a sinistra di ***array***;
* ***r***: indice della cella più a destra di ***array***;
* ***k***: indice dell’elemento da ricercare nel vettore.

È necessario, tuttavia, formulare alcune considerazioni sulla validità dei parametri ***k***, ***l*** ed ***r***.

In particolare, la validità di ***k*** non può essere assicurata e, di conseguenza, ***quickSelect*** termina con tempo costante se ***k*** < ***l*** o ***k*** > ***r***. La validità dei parametri ***l*** ed ***r*** è, invece, assicurata solamente nel caso in cui la prima chiamata alla procedura venga effettuata con ***l*** = 0 ed ***r*** = ***array***.length() – 1. In tal caso, le chiamate successive avranno come parametri un indice ***l*** o ***r*** già validato e un pivot.

### Esecuzione

Ad ogni chiamata di ***quickSelect*** viene verificata la correttezza dell’indice ***k*** e, in caso negativo, la procedura viene terminata con codice d’uscita -1, al fine di evitare la richiesta di risorse di calcolo per il lancio di un’eccezione (quale, ad esempio, IndexOutOfBounds).

In seguito, viene computata, tramite la procedura ***partition***, la partizione del vettore con indici di riferimento ***l*** e ***r***. Il pivot restituito assume la stessa posizione che avrebbe all’interno del vettore ordinato, con a sinistra tutti gli elementi a lui minori e a destra quelli maggiori.

Confrontando ***k*** e il pivot, si possono osservare tre fondamentali distinzioni:

* ***k*** corrisponde al pivot (in tal caso è stato selezionato il k-esimo elemento più piccolo del vettore);
* ***k*** è minore del pivot (in tal caso andrà ricercato nella partizione di sinistra);
* ***k*** è maggiore del pivot (in tal caso andrà ricercato nella partizione di destra).

Il confronto può quindi portare all’ottenimento dell’elemento situato in posizione ***k*** o, in alternativa, alla chiamata ricorsiva su una delle due partizioni delimitate da ***l*** e il pivot ***p***, oppure dal pivot ***p*** e l’indice ***r***.

### Output

L’output ottenuto corrisponde:

* al valore del k-esimo elemento più piccolo contenuto nell’array in input
* ad un exit code negativo nel caso in cui l’indice k non abbia validità.

## partition

Come specificato in precedenza, alla procedura ***partition*** è delegato il partizionamento del vettore. Dopo aver individuato un elemento indicato dalla variabile pivot, l’algoritmo permuta i valori dell’array al fine di riordinare il vettore di partenza fintanto che il pivot non abbia solamente elementi a sé minori alla sua sinistra e maggiori alla sua destra. L’implementazione adottata è quella di Nico Lomuto.

### Implementazione proposta

*public static int* partition(*int*[] array, *int* l, *int* r) {

*int* pivot = array[r];

*int* i = l;

*int* tmp;

*for*(*int* j = l; j < r; j++)  
 *if*(array[j] < pivot) {  
 tmp = array[i++];  
 array[i] = array[j];  
 array[j] = tmp;  
 }  
 tmp = array[i];  
 array[i] = array[r];  
 array[r] = tmp;

*return* i;

}

}

### Input

Gli argomenti della procedura sono i seguenti:

* ***Array***
* ***l***
* ***r***.

Per tutti i tre parametri, valgono le stesse condizioni descritte per l’input di ***quickSelect*** (**§** **2.3.2**).

### Esecuzione

L’implementazione adottata prevede la selezione dell’elemento pivot come valore nella cella di indice ***r***.

L’esecuzione richiede, inoltre, l’impiego di due variabili ausiliarie:

* ***i***, che fa riferimento alla posizione a sinistra del pivot e determina la sua posizione durante lo swap che segue la terminazione del ciclo di loop;
* ***tmp***, che permette di effettuare lo swap fra due celle del vettore.

Durante l’esecuzione del ciclo for viene in primis verificato che l’elemento contenuto nella cella di indice ***j*** sia minore del pivot. In caso positivo, quest’ultimo viene scambiato con l’elemento collocato in posizione ***i*** + 1. Tutti gli elementi minori del pivot vengono, di conseguenza, raggruppati tra le posizioni ***l*** e ***i***. In seguito, il pivot viene posizionato per mezzo dell’ultimo scambio previsto dalla funzione.

### Output

Al termine della procedura, viene restituito l’indice del pivot individuato.

## Complessità temporale di Quick select

Si suppongano gli indici atti a indicare rispettivamente

* La grandezza dell’array in input
* L’indice che verrà ricercato nell’array
* L’indice del pivot ritornato dalla procedura ***partition*** ad ogni chiamata ricorsiva dell’algoritmo.

### Individuazione e analisi del caso base

Si individuano due casi base principali:

1. Se **,** allora l’algoritmo termina con un numero costante di operazioni
2. Se , allora l’algoritmo termina con costo definito dalla chiamata della procedura ***partition***.

Di conseguenza si osserva come:



In entrambe le situazioni . Si assume, quindi, quest’ultimo come caso base

### Individuazione e descrizione della relazione di ricorrenza

Sulla base del risultato della procedura ***partition***(che ad ogni iterazione richiede un costo lineare), l’array in input viene partizionato in due sottoinsiemi rispettivamente di cardinalità e .

La successiva chiamata ricorsiva avviene sulla partizione dell’array con cardinalità maggiore.

Ne consegue la seguente equazione di ricorrenza:

### Dimostrazione del caso pessimo mediante equazione di ricorrenza

Si identificano come caso pessimo le due situazioni che ad ogni iterazione godono delle seguenti proprietà:



In tali situazioni, la disequazione ricorsiva di complessità segue il modello

Considerando iterazioni, consegue che:

Si verifica, quindi, che nel caso pessimo.

### Dimostrazione del caso medio mediante equazione di ricorrenza

Al fine di tale dimostrazione è opportuno indicare con il termine *“buon pivot”* quello che ricade nel sottoinsieme centrale dell’array, supposta la sua suddivisione in tre parti uguali.

Nel caso la condizione del *“buon pivot”* sia verificata, il costo della chiamata sarà:

Mentre, nel caso *p* non sia classificabile come buon pivot, il costo ricadrà nel caso pessimo:

Considerando le probabilità e atte a indicare, rispettivamente, la probabilità che *p* ricada o meno nei margini del *“buon pivot”*, è possibile descrivere la seguente relazione di ricorrenza per indicare il costo del caso medio dell’algoritmo.

.

Di conseguenza,

.

Si verifica, quindi, che nel caso medio.

### Conclusioni

In seguito alle precedenti dimostrazioni è possibile affermare che **l’algoritmo *“QuickSelect”*****opera, nel caso medio, con una complessità** . Tuttavia, nel caso pessimo, la complessità aumenta a .

Il primo passo è la scelta di un elemento da utilizzare come pivot, in questo caso viene scelto l’elemento contenuto nella cella di indice ***r*** (ovvero la cella più a destra del sub-array delimitato dagli indici ***l*** e ***r***).

Viene anche inizializzato l’indice ***i***, con valore ***l-1***, per tenere in memoria la posizione

Ogni elemento contenuto nelle celle dell’array tra ***l*** e ***r-1*** viene passato in rassegna per mezzo di un ciclo ***for*** e confrontato con il pivot all’interno di un **if-statement** condizionato dalla minorità del pivot rispetto all’elemento attualmente selezionato (contenuto nella cella di indice ***j***, variabile ausiliaria utilizzata nel ciclo for).

## Complessità spaziale di Quick select

L’esecuzione di ***quickSelect*** è in-place. Una volta allocata la memoria per salvare l’array in input, infatti, le successive allocazioni effettuate saranno costanti e impiegate per la memorizzazione delle sette variabili ausiliarie previste durante l’esecuzione di ***partition*** e ***quickSelect***. Non è richiesto ulteriore spazio in memoria e le operazioni previste vengono effettuate, tramite la procedura di swap, sulla stessa struttura dati ricevuta in input. Lo spazio necessario è, quindi, , dove n indica la dimensione del vettore in input.

# Heap Select

***HeapSelect*** è un algoritmo di selezione che sfrutta le proprietà di due binary heap durante la ricerca del k-esimo elemento richiesto.

## Cenni sulla binary heap

L’heap binaria è una struttura dati basata sull’implementazione di un albero binario quasi completo. Le unità fondamentali che lo costituiscono sono i nodi, caratterizzati da un valore di riferimento detto chiave e fra loro interconnessi da una relazione d’ordine.

Da ogni nodo, a partire da quello radice, si possono diramare un massimo di due nodi figli. Ci si riferisce a un nodo che ha almeno un figlio come nodo “padre”, mentre ad un nodo privo di figli come nodo “foglia”.

La due relazioni d’ordine sulle quali si basa la definizione di una binary heap sono le seguenti:

1. La chiave di un nodo padre è sempre maggiore di quella di un nodo figlio;
2. La chiave di un nodo padre è sempre minore di quella di un nodo figlio.

In particolare, nel primo caso la struttura dati risultante è identificata dal termine “Max Heap”, mentre nel secondo caso dal termine “Min Heap”.

## Obiettivo preposto

Si procede all’implementazione di due heap, denominate ***h1*** e ***h2***. La prima, è costruita a partire dal vettore fornito in input in tempo lineare e non viene modificata. La seconda, contiene inizialmente un solo nodo, corrispondente alla radice di ***h1***. All'i-esima iterazione, per ogni ***i*** appartenente all’intervallo , l'algoritmo estrae la radice di ***h2*** (corrispondente al nodo x­­i di ***h1***) e inserisce in ***h2*** i figli del nodo xi. Al termine delle iterazioni previste, la radice di ***h2*** corrisponderà al k-esimo elemento ricercato.

## Funzioni implementate

* *HeapSelect*;
* *select*;
* *insert*;
* *extract;*
* *buildHeap*;
* *maxHeapify* / *minHeapify*.

## Heap dinamica

In base al valore di ***k*** richiesto, può essere più efficiente impiegare rispettivamente una struttura dati di tipo MinHeap o una struttura dati di tipo MaxHeap. In particolare, se ***k*** è maggiore della metà della lunghezza dell’array in input, è opportuno cambiare il quesito di ricerca da “trovare il k-esimo elemento più piccolo nel vettore” a “trovare il k-esimo elemento più grande nel vettore”, ricalcolando il valore di ***k*** per adattarlo a tale richiesta.

La funzione ***HeapSelect***, tra le righe 51 e 55 della classe *“HeapSelect.java”*, si occupa di determinare il tipo di heap più efficiente per la risoluzione del problema ed eventualmente di calcolare la posizione del ***k***-esimo elemento più grande nel vettore.

## Classe Pair

Il funzionamento dell’algoritmo di selezione prevede che ad ogni iterazione venga estratto il nodo radice da ***h2*** e che vengano conseguentemente inseriti al suo interno i due nodi figli. Per ovviare al possibile problema di ricerca del nodo estratto da ***h2*** in ***h1***, le due heap vengono riempite con elementi di tipo ***Pair***.

La classe ***Pair*** consente di memorizzare, in due variabili d’istanza di un oggetto, la chiave di un nodo della heap e la sua posizione all’interno del vettore che costituisce la heap stessa.

Nella risoluzione del problema di selezione, questo è particolarmente utile in quanto, prima di estrarre da ***h2*** il nodo radice, è possibile accedere con costo alla sua posizione in ***h1***, sulla base del valore della variabile ***position*** nell’oggetto ***Pair.*** È così, possibile calcolare la posizione in ***h1*** dei nodi figli del nodo radice di ***h2***.

## HeapSelect

***heapSelect*** si occupa della creazione delle due heap, della chiamata del metodo di selezione ***select*** e della restituzione del risultato finale della selezione. Le strutture ***h1*** e ***h2***, vengono create scegliendo l’utilizzo di max o min heap a seconda del valore di ***k***. Un approccio dinamico di distinzione sul tipo di heap da utilizzare è necessario per l'ottimizzazione dei tempi di esecuzione dell’algoritmo di selezione: mentre se ***k*** è relativamente basso è opportuno ricercare il k-esimo elemento più piccolo del vettore utilizzando una min-heap, se ***k*** è elevato, cioè maggiore della metà della lunghezza dell’array in input, è necessario cambiare approccio alla risoluzione cercando l’(n-k)-esimo elemento più grande dell’array, impiegando di conseguenza una max-heap.

### Implementazione proposta

*public static int* HeapSelect(*int*[] array, *int* k) {  
 *int* h1\_size = array.length;  
 *boolean* isMinHeap = *false*;  
 *if*(k < h1\_size / 2)  
 isMinHeap = *true*;  
 *else* k = h1\_size - 1 - k;  
  
 ArrayList<Pair> h1 = *new* ArrayList<>(h1\_size);  
 *for*(*int* element : array)  
 h1.add(*new* Pair(element));  
 *build\_heap*(h1, h1\_size, isMinHeap);  
 *for*(*int* i = 0; i < h1\_size; i++)  
 h1.get(i).position = i;  
  
 ArrayList<Pair> h2 = *new* ArrayList<>(k+1);  
 h2.add(h1.get(0));  
  
 *return select*(h1, h2, k, isMinHeap);  
}

### Input

Gli argomenti della procedura sono i seguenti:

* ***array***, vettore di interi ben formato e non vuoto;
* ***k***, posizione del k-esimo elemento più piccolo del vettore da cercare, che si assume sia un indice valido rispetto al vettore in input.

### Esecuzione

Il flusso d’esecuzione di ***HeapSelect*** è divisibile in tre parti.

*boolean* isMinHeap = *false*;  
*if*(k < h1\_size / 2)  
 isMinHeap = *true*;  
*else* k = h1\_size - 1 - k;

La prima, si occupa di stabilire il tipo di struttura dati più efficiente tra una MinHeap e una MaxHeap, ricalcolando eventualmente il valore di ***k*** (**§ 3.5**).

A tal fine, la variabile booleana ***isMinHeap*** viene impiegata per specificare ai metodi successivamente chiamati quali tipi di strutture dati sono memorizzate nelle ArrayList fornite come argomenti. La variabile è inizializzata a false ma, se ***k*** fa riferimento ad una posizione nella metà di sinistra del vettore, viene portata a true, permettendo la ricerca del k-esimo elemento più piccolo nel vettore. Se ***k*** fa, invece, riferimento ad un indice nella seconda metà dell’array, allora ***isMinHeap*** non viene modificata e il valore di ***k*** viene ricalcolato in modo che rappresenti la posizione del k-esimo elemento più grande nel vettore.

ArrayList<Pair> h1 = *new* ArrayList<>(h1\_size);  
*for*(*int* element : array)  
 h1.add(*new* Pair(element));  
*build\_heap*(h1, h1\_size, isMinHeap);  
*for*(*int* i = 0; i < h1\_size; i++)  
 h1.get(i).position = i;

Successivamente, viene inizializzata, con lunghezza pari a quella dell’array in input, la ArrayList ***h1*** per rappresentare la prima heap.

Quest’ultima viene riempita da oggetti di tipo ***Pair***, generati a partire dagli elementi contenuti nell’array in input. Viene, poi, applicato l’algoritmo di costruzione della heap, al fine di ordinare gli elementi secondo la relazione d’ordine del tipo a cui la variabile ***isMinHeap*** fa riferimento.

Successivamente, viene modificato ogni elemento di tipo ***Pair*** in ***h1*** e ne viene impostata la variabile ***position***, necessaria per la successiva fase di selezione.

ArrayList<Pair> h2 = *new* ArrayList<>(k+1);  
h2.add(h1.get(0));  
  
*return select*(h1, h2, k, isMinHeap);

Infine, viene inizializzata un’ulteriore ArrayList, ***h2***, per rappresentare la seconda heap. Questa, come dimostrato in seguito, occupa una lunghezza di al più ***k*** elementi (viene passato in argomento perché ***k*** è stato precedentemente decrementato per essere una posizione valida nel vettore). In ***h2*** viene, quindi, posizionato il nodo radice di ***h1*** e viene chiamata la procedura di selezione ***select***.

### Output

La funzione ***HeapSelect*** ritorna alla funzione chiamante (main o funzione esterna alla classe) il valore passato dalla procedura ***select***, ovvero il k-esimo elemento più piccolo/grande nell’array in input.

## select

***select*** gestisce le manipolazioni su ***h2*** per trovare l’elemento cercato.

### Implementazione proposta

*private static int* select(ArrayList<Pair> h1, ArrayList<Pair> h2, *int* k, *boolean* isMinHeap) {  
 *for*(*int* i = 0; i < k; i++){  
 *int* root\_position = h2.get(0).position;  
 *int* left = root\_position \* 2 + 1;  
 *int* right = root\_position \* 2 + 2;  
  
 *extract*(h2, isMinHeap);  
  
 *if*(left < h1.size())  
 *insert*(h2, h1.get(left), isMinHeap);  
 *if*(right < h1.size())  
 *insert*(h2, h1.get(right), isMinHeap);  
 }  
 *return* h2.get(0).key;  
}

### Input

Gli argomenti della procedura sono i seguenti:

* Le due heap, ***h1*** e ***h2***, contenute nelle ArrayList di Pair, create precedentemente dalla funzione ***HeapSelect***.
* ***k***, posizione del k-esimo elemento più piccolo/grande nell’array in input all’algoritmo;
* ***isMinHeap***, variabile booleana che indica il tipo di heap utilizzato.

### Esecuzione

L’esecuzione della funzione si basa su un ciclo ***for*** ripetuto ***k*** volte. Ad ogni iterazione, viene assegnato a ***root\_position*** la posizione in ***h1*** del nodo radice di ***h2***, ottenibile ispezionando il valore della variabile d’istanza ***position***. Questo intero si utilizza per calcolare le posizioni degli eventuali figli del nodo radice di ***h2*** contenuti in ***h1*** (***left*** figlio sinistro e ***right*** figlio destro).

Si estrae il nodo radice della heap ***h2*** e viene valutata, per mezzo di due ***if-statements***, l’eventuale presenza dei due figli di ***root*** in ***h1***.

Se ***left*** è un indice valido in ***h1***, viene chiamata la procedura d’inserimento di un nuovo Pair in ***h2***. Alla funzione vengono passati ***h2***, il Pair che occupa la posizione ***left*** in ***h1*** e il boolean ***isMinHeap***. Lo stesso procedimento viene successivamente applicato per l'indice ***right***.

### Output

Il metodo ***select*** restituisce alla funzione chiamante l’elemento radice di ***h2***.

## insert

La procedura per l’inserimento può considerarsi uguale a quella “standard” per inserire un elemento in una heap:

* il nuovo elemento viene inserito in coda alla heap;
* con un approccio bottom-up si confronta il valore del nodo appena inserito con quello del corrispettivo padre;
* se la chiave di quest’ultimo è maggiore (nel caso di una MinHeap, minore altrimenti) di quella del nodo figlio allora i nodi vengono scambiati e si reitera sul nodo che precedentemente era il padre e il nuovo nodo padre;
* l’iterazione viene ripetuta fino a quando non si raggiunge il nodo radice, oppure quando viene ripristinata la relazione d’ordine dell’heap.

Anche questo metodo è in grado di differenziare l’esecuzione a seconda del valore di ***isMinHeap***.

## extract

Il metodo per l’estrazione dell’oggetto in posizione 0 di ***h2*** è quello comunemente utilizzato su una heap generica:

* il primo e l’ultimo elemento vengono scambiati;
* l’ultimo elemento viene rimosso;
* si applica il metodo di heapify adatto al tipo di heap utilizzata, sul nodo in prima posizione;

Anche questo metodo è in grado di differenziare l’esecuzione a seconda del valore di ***isMinHeap***.

## buildHeap

Il metodo per la costruzione della heap a partire da un array di elementi è quello caratteristico di una comune implementazione di una binary-heap: dato un vettore, i suoi elementi devono essere ordinati in modo che vengano garantite le relazioni d’ordine previste da tale struttura dati.

Per soddisfare tali proprietà viene chiamata iterativamente, su ogni nodo genitore (che quindi ha almeno un figlio), la procedura standard di ***heapify***, che riordina il sotto-albero con radice in corrispondenza dell’indice ***i*** in modo che soddisfi le richieste di una heap.

## Complessità temporale di heapSelect

Considerata la struttura articolata che caratterizza l’algoritmo ***HeapSelect***, è conveniente analizzare anticipatamente la complessità dei singoli metodi utilizzati.

Per le successive dimostrazioni si considera il caso peggiore in cui la heap è un albero binario completo e ***k*** fa riferimento alla posizione dell’array in input.

### Complessità dei metodi di heapify

Il tempo di esecuzione di heapify su un sottoalbero di lunghezza ***n*** con radice corrispondente al nodo ***i*** è per ottenere e ripristinare la relazione tra ***i***,il figlio sinistro e il figlio destro. A questo si aggiunte il tempo necessario per eseguire la chiamata ad heapify sul sottoalbero con radice in uno dei due nodi figlio.

Il sottoalbero ricorsivo ha grandezza massima pari a , da cui deriva la seguente ricorsiva di complessità temporale:

Nel caso peggiore, heapify verrà eseguito sul nodo radice di una heap e richiamerà sé stesso su tutti i nodi tra la radice e il nodo foglia più distante: il numero di chiamate ricorsive corrisponde, quindi, all’altezza ***h*** della heap ed equivale a . Applicando le relative sostituzioni si ottiene:

### Complessità del metodo di costruzione della heap

La funzione di build (indipendentemente dal tipo) esegue un ciclo for per ripetizioni. Ad ogni iterazione viene effettuata la chiamata ad una funzione di heapify.

Come mostrato in precedenza, il metodo heapify ha una complessità di e quindi una prima approssimazione della complessità di costruzione della heap può essere appartenente a .

Un’analisi più attenta e un approccio puntuale portano ad una valutazione più accurata della complessità delle funzioni di build:

* È possibile esprimere la complessità temporale utilizzando l’altezza h dell’albero;
* Ogni iterazione ha costo ;
* Il numero di nodi su cui viene eseguita la build è , dove è il numero massimo di nodi ad una certa altezza e la sommatoria trova il numero massimo di nodi tra il nodo radice e l’ultimo livello senza foglie.

La funzione di complessità diventa quindi:

Con n molto grande e data la serie aritmetica per :

Riassumendo:

Quindi, la complessità temporale del metodo di costruzione ***build\_heap*** è .

### Complessità del metodo insert

L’operazione di aggiunta in coda alla ArrayList ha costo , così come le inizializzazioni e assegnamenti delle variabili ***current\_node*** e ***parent\_node***. Il ciclo while, indipendentemente dal tipo di heap utilizzato, opera per un massimo di volte (viene eseguito su tutti i nodi compresi tra la radice e il nodo appena aggiunto), ovvero per un numero di iterazioni pari all’altezza della heap. La complessità è quindi .

### Complessità del metodo extract

Il metodo di estrazione di un nodo dalla heap, ***extract***, svolge tre operazioni con costo costante e chiama una delle due funzioni di heapify a seconda del tipo di heap trattato. La funzione di complessità e quindi: .

### Lunghezza della ArrayList h2

Prima di procedere ulteriormente con lo studio della complessità, è necessario analizzare un aspetto fondamentale: ***h1*** contiene n elementi, tutti quelli contenuti nell’array di interi dato in input, ma ***h2*** che dimensione massima ha?

È possibile dimostrare induttivamente che ***h2*** abbia dimensione massima pari al valore di ***k*** in ***HeapSelect***:

N.B. il ***k*** di seguito utilizzato fa riferimento a quello in input. La distinzione si rende necessaria in quanto quest'ultimo è compreso tra 1 e la lunghezza dell’array, mentre all’interno del codice lo stesso indice viene codificato da un valore compreso tra 0 e la lunghezza dell’array – 1. Di conseguenza, ***k*** viene decrementato prima di essere passato alla chiamata di ***HeapSelect***.

* Per verificare la precedente affermazione è necessario dimostrare due proprietà:
  + ad ogni iterazione il vettore ***h2*** contiene elementi
  + , dove ***i*** è la variabile utilizzata nella guardia del ciclo for in ***select*** cheincrementa ad ogni iterazione, tenendo conto del numero di esecuzioni
* **Caso base**: viene individuato con ***k*** in input uguale a 1. In tal caso, ***h2*** viene creato con capacità pari a 1 da ***HeapSelect***, per poi essere passato alla procedura ***select***. Quest’ultima esegue l’inizializzazione del ciclo for, che termina immediatamente in conseguenza alla relativa funzione di guardia , che con e non è verificata. Il vettore ***h2*** non subisce manipolazioni e mantiene, dunque, la capacità di ;
* **Passo induttivo**:
  + **Ipotesi induttiva**: ***h2***, al termine della ***k***-2-esima iterazione (), contiene al massimo elementi;
  + **Tesi induttiva**: ***h2*** contiene al massimo ***k*** elementi al termine della ***k***-1-esima iterazione (punto di terminazione del ciclo for);
  + **Dimostrazione:** in seguito alla ***k***-2-esima iterazione, per ipotesi, il numero di elementi presenti in ***h2*** è pari a . Alla successiva iterazione viene estratto un elemento e ne vengono aggiunti al più due (un nodo ha due figli nel caso peggiore). Considerando, quindi, il caso peggiore, il numero di elementi contenuti in ***h2*** aumenta a . Ne consegue la terminazione del ciclo, in quanto la sua condizione di guardia non viene verificata: , ma .

### Complessità del metodo select

Frammentando l’analisi di complessità è possibile osservare come:

* Il ciclo for venga eseguito al più volte;
* Vengano eseguite una serie di operazioni di costo ;
* La chiamata, ed esecuzione, di ***extract*** avvenga in complessità (**§ 3.11.4**), che diventa in quanto la lunghezza di ***h2*** all’ultima iterazione è ***k*** (**§ 3.11.5**);
* Il costo per l’inserimento di entrambi i nodi è . Infatti, la lunghezza di ***h2*** all’ultima iterazione è ***k*** (**§ 3.11.5**) e la complessità della procedura ***insert*** è , dove è la lunghezza della heap.

L’equazione di complessità è, di conseguenza, così definita:

### Complessità di HeapSelect

In seguito all’analisi effettuata è possibile ricostruire la funzione di complessità temporale per ***HeapSelect***:

## Complessità spaziale di HeapSelect

***HeapSelect*** richiede solamente lo spazio di memorizzazione delle due strutture dati e una quantità costante di spazio aggiuntivo. Si può pertanto affermare che si tratta di un algoritmo in-place con complessità spaziale appartenente a .

# Median-of-medians select

## Obiettivo proposto

Suddividere il vettore fornito in input in blocchi di dimensione limitata e individuare la mediana delle mediane. Più precisamente, l'algoritmo deve eseguire le seguenti operazioni:

* Divisione dell'array in blocchi di 5 elementi, escluso l'ultimo eventuale blocco che potrà contenere meno di 5 elementi;
* Ordinamento e calcolo della mediana di ciascun blocco,
* Calcolo della mediana delle mediane dei blocchi, attraverso chiamata ricorsiva allo stesso algoritmo
* Partizionamento dell'intero array attorno alla mediana, tramite una variante della procedura "partition" dell'algoritmo "quick sort"
* Chiamata ricorsiva relativa alla parte sinistra o destra dell’array rispetto alla mediana individuata, in funzione del valore ***k*** fornito in input.

## Funzioni implementate

* ***medianSelect***
* ***medianOfMedians***
* ***InsertionSort***
* ***partition***
* ***swap***

### Implementazione proposta

*public static int* medianSelect(*int*[] array, *int* l, *int* r, *int* k) {  
 *int* i = l;  
 *int* lastGroup = r - (r - l) % 5;  
 *while*(i < r) {  
 *InsertionSort*(array, i, i == lastGroup ? r : (i + 4));  
 i+=5;  
 }  
  
 *if*(r-l < 5)  
 *return* k;  
  
 i = l + 2;  
 *int* count = 0;  
 *while*(i <= r ) {  
 *swap*(array, i, l + count++);  
 *if* (i < lastGroup - 3)  
 i += 5;  
 *else* {  
 i += 3 + (r - lastGroup) / 2;  
 *swap*(array, i, l + count);  
 *break*;  
 }  
 }  
  
 *medianOfMedians*(array, l, l + count);  
  
 *int* pivot = *partition*(array, l, r);  
 *if*(pivot == k)  
 *return* pivot;  
 *else if* (k < pivot)  
 *return medianSelect*(array, l, pivot - 1, k);  
 *else  
 return medianSelect*(array, pivot + 1, r, k);  
}

### Input

* ***array***, vettore di interi non vuoto su cui viene effettuata la selezione
* ***l*** e ***r***, indici del primo e ultimo elemento del vettore. Sono necessari per la manipolazione dei sub-array del vettore in input da parte della funzione ricorsiva
* ***k***, posizione del k-esimo elemento più piccolo nel vettore. Si assume sia un indice compreso tra 0 e la lunghezza del vettore in input - 1.

### Esecuzione

La versione proposta dell’algoritmo è in-place (si analizzerà lo spazio richiesto per l’esecuzione in seguito). La gestione delle mediane non avviene, infatti, tramite l'impiego di un vettore ausiliario, ma all’interno del vettore in input stesso.

In funzione di un’osservazione puntuale atta all’analisi e descrizione del comportamento dell’algoritmo, il codice viene suddiviso in blocchi.

**Algoritmo:**

*int* first = l;

La variabile ***first*** ha lo scopo di identificare la posizione del primo elemento di una cinquina, viene inizializzata ad ***l*** (indicando così il primo elemento a sinistra del vettore) e viene successivamente incrementata all’interno di un apposito ciclo while.

*int* lastGroup = r - (r - l) % 5;

***lastGroup*** è l’indice del primo elemento dell’ultimo raggruppamento: l’ultimo raggruppamento è l’unico che può contenere meno di cinque elementi e va, quindi, trattato indipendentemente.

*while*(first < r) {  
 *InsertionSort*(array, first, first == lastGroup ? r : (first + 4));  
 first+=5;  
}

Il ciclo while opera da ***l*** a ***r*** e richiama ad ogni iterazione la funzione ***InsertionSort*** per permettere il riordinamento delle cinquine. Ad ***InsertionSort*** vengono forniti in input l’array, l’indice del primo elemento della cinquina da riordinare e l’indice del quinto elemento, che viene ricavato dinamicamente in base al valore di ***first*** e di ***lastGroup***: se ***first*** corrisponde alla posizione del primo elemento dell’ultimo gruppo di interi, allora viene passato come secondo parametro e l’ordinamento avverrà tra ***first*** e ***r***; altrimenti è confermata la presenza di cinque elementi e l’ordinamento avverrà tra ***first*** e ***first*** *+* 4. Successivamente, ad ogni iterazione viene aumentato il valore di ***first*** di 5 per passare alla cinquina successiva, o uscire quando ***first*** .

*if*(r-l < 5)  
 *return* k;

L’accesso all’if-statement avviene quando il vettore, già ordinato, contiene al più cinque elementi. In tal caso, viene restituita la posizione del ***k***-esimo elemento più piccolo, terminando l’esecuzione.

*int* median = l + 2;

La variabile ***median*** rappresenta la mediana di una cinquina e si trova sempre in terza posizione rispetto al raggruppamento. Per trovare la prima mediana, ***median*** viene inizializzata ad ***l*** + 2: ***l*** è, infatti, l’indice del primo elemento a sinistra della prima cinquina e, spostandosi a destra di due posizioni, viene individuata la relativa mediana. Successivamente basterà sommare 5 al valore di ***median*** per localizzare la mediana della cinquina adiacente.

*int* count = 0;

La variabile contatore tiene conto di quante mediane sono state individuate (che corrispondono a ***count*** + 1);

*while*(median <= r ) {  
 *swap*(array, median, l + count++);  
 *if* (median < lastGroup - 3)  
 median += 5;  
 *else* {  
 median += 3 + (r - lastGroup) / 2;  
 *swap*(array, median, l + count);  
 *break*;  
 }  
}

Il ciclo while viene utilizzato per passare in rassegna ogni mediana contenuta tra ***l*** ed ***r***. Ad ogni iterazione, la mediana in posizione ***median*** viene scambiata con l’elemento in posizione ***l***+*count*, primo da sinistra che non contiene ancora una mediana. In seguito, viene valutato come incrementare la variabile ***median*** al fine di passare alla mediana successiva: se ***median*** è minore rispetto alla posizione della penultima mediana (***lastGroup*** – 3), è possibile sommare 5 per arrivare alla posizione successiva. Altrimenti, ***median*** si trova in corrispondenza della penultima mediana e per raggiungere la successiva è necessario calcolare l’incremento corretto. In particolare, in tal caso viene sommato 3 per raggiungere il primo elemento dell’ultimo partizionamento dell’array, si somma la posizione relativa dell’ultima mediana e si procede allo swap tra mediana e prima cella sinistra non contenente una mediana. Successivamente, la procedura viene terminata con un *break-statement*. Si distinguono, di conseguenza, due sub-array del vettore iniziale: il primo, compreso tra ***l*** e ***l***+*count,* contenente tutte le mediane del vettore, il secondo, descritto dalla rimanente porzione di vettore, contenente tutti gli elementi che non sono mediane.

*medianOfMedians*(array, l, l + count);

Per calcolare la mediana delle mediane viene chiamata la funzione ricorsiva ***medianOfMedians,*** che prevede come argomenti l’array in input e gli indici ***l*** e ***l*** + count atti a identificare l’intervallo contenente le mediane;

*int* pivot = *partition*(array, l, r);  
*if*(pivot == k)  
 *return* pivot;  
*else if* (k < pivot)  
 *return medianSelect*(array, l, pivot - 1, k);  
*else*  
 *return medianSelect*(array, pivot + 1, r, k);

Dopo le eventuali chiamate ricorsive, in posizione ***l*** sarà possibile individuare la mediana delle mediane, che viene utilizzata come pivot dalla funzione ***partition***. Partizionato il vettore, il pivot si trova nella stessa posizione in cui si troverebbe se l’intero vettore venisse ordinato, con a sinistra tutti gli elementi a lui minori e a destra tutti gli elementi a lui maggiori. L’if-statement permette, inoltre, di verificare che il pivot sia l’intero cercato. In base alla condizione verificata, viene ritornata la posizione o, in alternativa, viene selezionata la partizione alla quale applicare la successiva chiamata ricorsiva, analogamente a quanto illustrato per l’algoritmo ***quickSelect*** (**§ 2.3.3**).

### Output

***medianOfMediansSelect*** restituisce in output la posizione del ***k***-esimo elemento più piccolo del vettore in due occasioni:

1. Il vettore contiene al più 5 elementi, che vengono prima ordinati con ***InsertionSort***, e che occupano, quindi, la loro posizione rispettando una relazione d’ordine crescente;
2. La posizione del pivot impiegato per la partizione corrisponde a ***k***.

## medianOfMedians

***medianOfMedians*** è una versione modificata di ***medianSelect*** atta alla manipolazione dei sub-array di mediane.

L’algoritmo per le mediane, come quello di selezione, effettua l’ordinamento degli elementi tra ***l*** ed ***r*** a gruppi di cinque elementi, applicando ***InsertionSort***. In seguito, se il sub-array contiene al più cinque elementi è possibile affermare che quello in posizione centrale corrisponda alla mediana delle mediane. Viene, quindi, portato in posizione ***l*** con uno swap. In caso contrario, il sub-array contiene più di cinque elementi ed è, quindi, necessario individuare le mediane, portarle all’inizio del sub-array e chiamare ricorsivamente la funzione ***medianOfMedians*** fino al verificarsi del primo caso.

La funzione non restituisce alcun valore, ma sposta in posizione ***l*** la mediana delle mediane. Terminate tali chiamate ricorsive, ***medianSelect*** individuerà il valore in tale posizione.

## InsertionSort

La funzione ***InsertionSort*** prevede l’implementazione in Java dell’ominomo algoritmo di ordinamento, parametrizzato ad ***l*** ed ad ***r*** affinché possa operare su sotto-vettori del vettore in input, senza richiedere l’allocazione di nuove strutture dati.

## partition

Si tratta di una variante dell’algoritmo implementato in ***quickSelect*** (**§ 2.4**). Si distingue dalla versione precedente in quanto individua come pivot l’elemento in posizione ***l***, cioè la mediana delle mediane, invece che quello in posizione ***r.*** Riduce, inoltre, il numero di operazioni di scambio tra elementi con una chiamata ad una apposita funzione di ***swap***.

## swap

Funzione che, dati un array e due interi, scambia i due elementi contenuti nelle celle corrispondenti ai due indici ***i*** e ***j***.

## Complessità temporale di medianSelect

Ai fini dimostrativi preposti, in seguito viene inteso come equivalente al valore descritto da. Il parametro indica la lunghezza del vettore, o sotto-vettore, trattato dalle relative funzioni.

### Complessità di InsertionSort su gruppi di 5 elementi

La funzione di ordinamento applicata ad un gruppo ridotto di elementi esegue in tempo costante. L’ordinamento viene ripetuto per volte (numero di cinquine nel vettore in input), pertanto la complessità del blocco di codice per il riordinamento delle cinquine appartiene a .

### Individuazione delle mediane

Il blocco di codice in ***medianSelect*** che individua le mediane effettua operazioni di costo costante (assegnazioni, scambi, valutazioni condizionali) per volte, dove corrisponde al numero di mediane rispetto nel vettore iniziale. L’individuazione delle mediane ha, quindi, complessità appartenente a .

### Complessità di medianOfMedians

La complessità per individuare la mediana delle mediane è descritta dall’equazione ricorsiva di complessità: , dove è il costo di esecuzione dell’ordinamento a gruppi di cinque, sommato al costo di individuazione della mediana delle mediane. corrisponde, invece, al costo descritto dalla successiva chiamata ricorsiva.

Di seguito vengono svolti i passaggi per la risoluzione della precedente equazione ricorsiva:

In seguito allo sviluppo precedente, è possibile concludere che la funzione ***medianOfMedians*** appartenga e che operi, quindi, in tempo lineare.

### Complessità della chiamata ricorsiva di medianSelect

Il vettore di partenza contiene mediane (una per ogni cinquina). Metà delle mediane, elementi, sono minori di ***median*** (mediana delle mediane), in quanto quest’ultima occupa una posizione centrale all’interno del raggruppamento ordinato di mediane.

All’interno di una cinquina ordinata, la mediana ricopre la posizione centrale, lasciando rispetto a sé stessa due elementi minori a sinistra e due maggiori a destra.

È, di conseguenza, sicura la presenza di cinquine con mediane minori di ***median.*** Per ogni cinquina, sono presenti tre elementi minori della mediana delle mediane. Vi è dunque la certezza che, dato un vettore di lunghezza , esso conterrà almeno elementi minori della mediana delle mediane. La stessa dimostrazione viene applicata per individuare il numero minimo di elementi maggiori di ***median***.

Supposto che:

* ***k*** sia diverso dal ***pivot***
* il sub-array più piccolo del vettore, che sia esso quello a sinistra o a destra del ***pivot***, contenga elementi
* il restante sub-array contenga elementi di conseguenza
* la chiamata ricorsiva venga effettuata su quest’ultimo sub-array

È possibile confermare come il costo di tale chiamata risulti essere .

### Complessità totale della funzione medianSelect

La complessità finale della funzione ***medianSelect*** è data dallo sviluppo della seguente espressione:

Dove è la somma delle complessità per dividere ed ordinare, a gruppi di cinque, gli elementi del vettore, per individuare e spostare ad inizio vettore tutte le mediane e per individuare la mediana delle mediane. A questa, viene sommato il costo della chiamata ricorsiv, che ha costo in base a quanto osservato in precedenza.

Supponendo che :

È possibile individuare una costante c, atta a verificare tali condizioni. Pertanto .

## Complessità spaziale di medianSelect

***medianSelect*** impiega lo spazio di memorizzazione richiesto per il vettore in input e al più un numero costante di variabili atomiche per permettere l’esecuzione. L’algoritmo risulta, quindi, in-place.

# Raccoglimento dei dati

Il metodo **public *static void main(String[] args)*** della classe Time, rende possibile l’esecuzione del progetto e si occupa autonomamente di:

1. Computare il valore di *maxError* sulla base della risoluzione di clock della macchina esecutrice
2. Inizializzare un file Excel preposto al raccoglimento dei dati che verranno successivamente ricavati
3. Raccogliere ripetutamente i tempi di esecuzione di ogni algoritmo in base al valore di k e al variare della dimensione dell’input.

## Ottenimento della risoluzione di clock

Al fine di una precisa misurazione e della minimizzazione dell’errore relativo, è opportuno considerare la risoluzione del clock durante l’esecuzione degli algoritmi. L’ottenimento di quest’ultima è delegato al metodo ***getResolution()***, contenuto all’interno della classe ***Resolution***.

*public static* Long getResolution(){  
 *long* start, end, res;  
 Vector<Long> resolutions = *new* Vector<>();  
 *for*(*int* i=1; i<=200; i++) {  
 start = System.*nanoTime*();  
 *do* {  
 end = System.*nanoTime*();  
 } *while* (start == end);  
 res = end - start;  
 resolutions.add(res);  
 }  
  
 Collections.*sort*(resolutions);  
 *return* resolutions.get(100);  
}

Per duecento iterazioni, vengono richiesti un valore temporale iniziale e finale fintanto che questi non differiscono. Soddisfatta la condizione, la differenza fra il tempo iniziale e quello finale viene inserita in una struttura dati ***Vector<Long>***. Al termine delle iterazioni, viene restituita la mediana del vettore.

Il valore ottenuto viene successivamente moltiplicato per 101 al fine di ottenere un parametro **maxError** atto a garantire un errore relativo inferiore o pari all’1% durante il calcolo dei tempi di esecuzione.

Supponendo, infatti, il verificarsi della seguente condizione (dove indica il tempo di esecuzione misurato, il tempo di esecuzione effettivo ed *R* la risoluzione del clock):

,

e indicando l’errore relativo con la seguente espressione:

,

è possibile affermare che

Di conseguenza, al fine di ottenere un errore relativo inferiore o pari all’1% è necessario che si verifichi la seguente condizione:

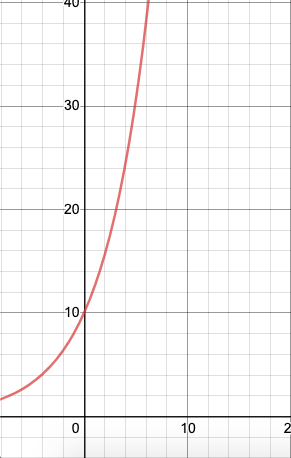
Il successivo sviluppo algebrico comporta la seguente disequazione:

## Warmup della JVM

La JVM sfrutta il meccanismo di lazy loading delle classi al fine di ottimizzare lo spazio di memoria richiesto durante il runtime: le classi vengono caricate dinamicamente in base alla porzione di codice che viene utilizzata. Tuttavia, la raccolta di dati temporali sull’esecuzione degli algoritmi di selezione richiede la riduzione del lazy loading durante le misurazioni. A tal fine, gli algoritmi vengono lanciati con input costituito da vettori generati casualmente per permettere alla JVM di caricare tutte le classi necessarie.

Nel passaggio dal meccanismo di warmup all’esecuzione della misurazione dei tempi vi sono comunque valori anomali dovuti al funzionamento a alla gestione della memoria da parte della JVM. Tuttavia, la loro effettiva influenza sulla valutazione dei tempi d’esecuzione degli algoritmi è minima.

## Modellazione del foglio Excel

La comunicazione tra il codice e il file *Time.xlsx* preposto al raccoglimento dei dati avviene tramite la libreria **Apache POI**, che permette una completa modellazione del foglio attraverso le interfacce:

* ***Workbook*** (per la modellazione completa del file)
* ***Sheet*** (per la gestione di molteplici fogli)
* ***Row*** (per la manipolazione delle righe di ogni foglio)
* ***Cell*** (per la manipolazione di ogni singola cella).

Il file viene dapprima introdotto tramite un oggetto di tipo *FileInputStream* e, successivamente, utilizzato come destinazione per l’inizializzazione di un *Workbook*. Per ogni dimensione di input viene predisposto un nuovo foglio, nel quale vengono raccolti i tempi di esecuzione dei tre algoritmi calcolati sulla base degli indici preposti.

Al termine dell’esecuzione il foglio viene salvato, reso accessibile e i dati calcolati possono essere visualizzati e analizzati tramite le funzionalità offerte dal software, applicando le opportune statistiche riassuntive.

## Ottenimento dei tempi di esecuzione

Al fine di ottenere tutti i dati necessari, viene predisposto l’ottenimento di cinquanta misurazioni di tempo per ciascuno degli algoritmi in analisi, calcolate indipendentemente per ognuno degli indici predisposti. Il procedimento viene ripetuto per cinquanta iterazioni, ad ognuna delle quali viene associata la generazione di un nuovo array di input con dimensione crescente.

### Generazione dell’input

Al fine di valutare il tempo di esecuzione degli algoritmi in base alla variabilità del vettore, vengono generati array la cui dimensione cresce esponenzialmente. Ad ogni iterazione, la dimensione dell’input viene calcolata tramite la funzione seguente:

dove *size* indica la dimensione dell’input ed *n* l’indice dell’attuale iterazione. Si osserva come, grazie alla rapida crescita, sia possibile lavorare con array di grandezza consistente dopo un basso numero di iterazioni.

### Popolamento dell’input

Al fine di ottenere un’aleatorietà sufficiente alle finalità dell’analisi, gli input vengono popolati da valori interi generati pseudo-casualmente dell’algoritmo lineare congruenziale *“Mersenne Twister”*, sviluppato nel 1997 dai matematici Makoto Matsumoto e Takuji Nishimura.

La sua principale peculiarità e diffusione è dovuta all’immenso periodo, che corrisponde al numero primo di Mersenne e alla notevole efficienza, che lo rende particolarmente veloce nella generazione. Sottoposto ai *Diehard Battery of Randomness Tests*, i valori generati sono stati in grado di superare tutti i quindici test previsti.

Gli sviluppatori hanno dimostrato, oltre al periodo, anche l’equi-distribuzione e uniformità dell’algoritmo. Di conseguenza, i valori ottenuti costituiscono campioni casuali aleatori che permettono la generalizzazione delle conclusioni ottenute durante il successivo confronto dei tempi di esecuzione ottenuti.

Al fine di garantire un ulteriore omogeneità, inoltre, vengono impiegati dei valori costanti per l’indice k, sulla base della dimensione dell’input. Questi corrispondono a:

* .

Per ogni input vengono, quindi, ricercati i valori collocati rispettivamente all’inizio, a metà, in posizione logaritmica e alla fine dell’array attuale, in relazione alla sua dimensione. In base al valore di k, gli algoritmi ricadranno nei loro rispettivi casi ottimi, medi e pessimi.

### Esecuzione degli algoritmi

L’esecuzione dei tre algoritmi e il relativo calcolo del loro tempo di esecuzione sono procedimenti delegati rispettivamente ai metodi ***getExTimeQuickSelect()***, ***getExTimeHeapSelect()*** e ***getExTimeMedianSelect()***, appartenenti alla classe Time.

In forma analoga, ogni metodo riceve in input:

1. L’***array*** di numeri casuali
2. L’indice ***k***
3. La costante ***maxError***che, computata al lancio del programma, fornisce il tempo minimo di esecuzione da garantire affinché l’errore relativo non superi il limite .

In seguito, l’algoritmo di selezione viene eseguito ripetutamente fino all’ottenimento di un tempo di esecuzione inferiore o uguale al valore di ***maxError***. Soddisfatta tale condizione, viene ritornato il tempo di esecuzione calcolato, rapportato al numero di iterazioni effettuate (indicato dalla variabile intera *count*).

In particolare, i tempi di esecuzione vengono ottenuti tramite il metodo ***public static long nanoTime()*** contenuto all’interno della classe **System** dello standard Java Development Kit. Quest’ultima, appoggiandosi alla sorgente di tempo ad alta risoluzione della Java Virtual Machine, restituisce una misurazione di tempo attuale in nanosecondi.

# Analisi dei dati

I dati grezzi raccolti in seguito al calcolo dei tempi di esecuzione dei tre algoritmi vengono, in tempo reale, inseriti all’interno del file “*Time.xlsx*”. In particolare, la struttura prevede che:

* Il file sia composto da diversi Worksheets, uno per ogni ***targetSize*** dell’array in input;
* Ogni worksheet contenga le cinquanta misurazioni relative ad uno specifico valore di ***targetSize***,per ognuno dei k utilizzati;
* Per ogni set di misurazioni, vengano prodotte opportune statistiche riassuntive quali mediana, deviazione standard e varianza.

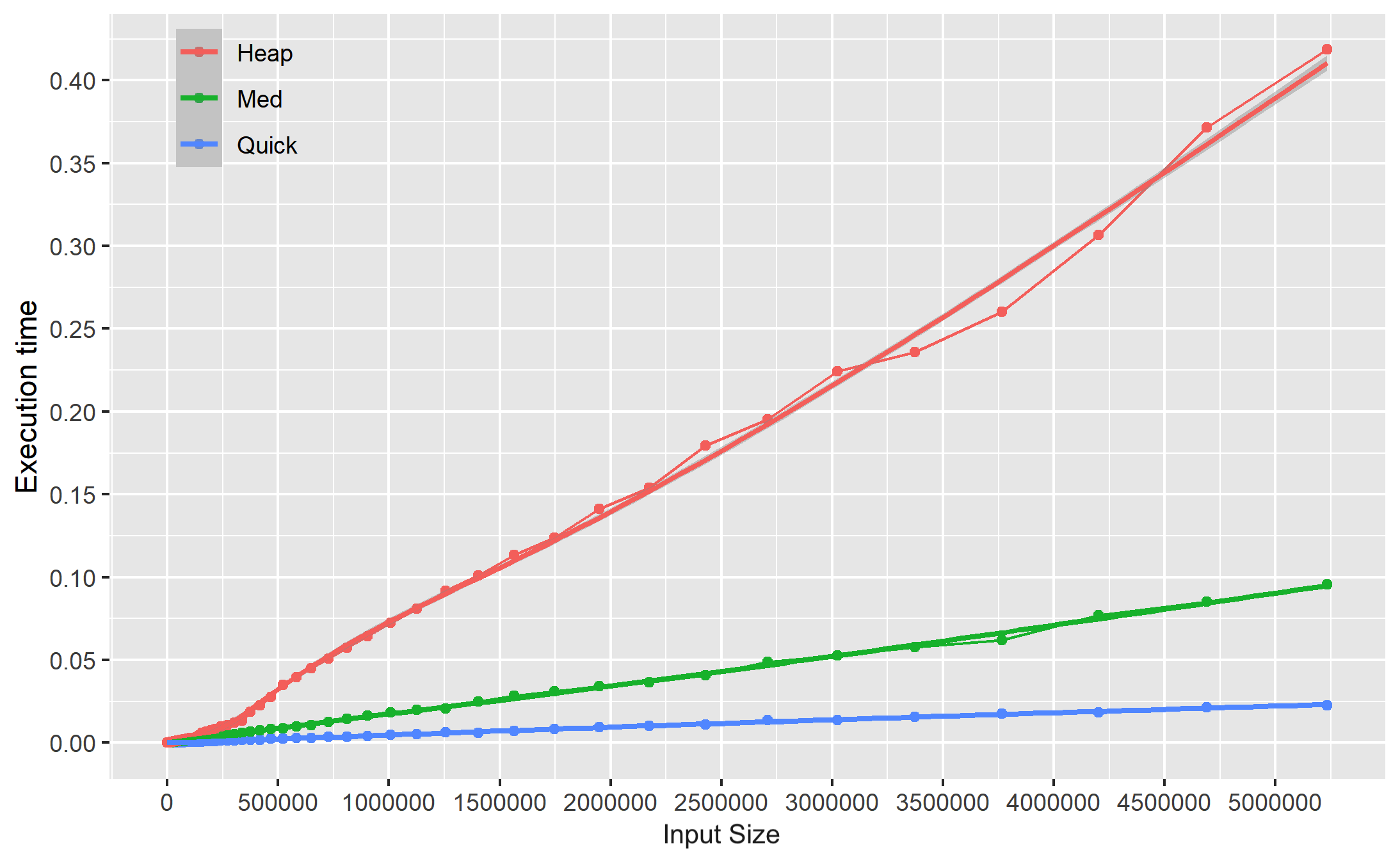
Dall’insieme delle mediane ottenute relativamente ad ogni singolo set di misurazioni, vengono successivamente estratte le ulteriori mediane. É così resa possibile un’ulteriore sintetizzazione dei dati grezzi, al fine di permettere la produzione di rappresentazioni grafiche atte a fornire un’efficace visualizzazione del comportamento degli algoritmi.

Il raggruppamento delle statistiche permette di osservare:

* L’andamento complessivo degli algoritmi in scala lineare;
* L’andamento complessivo degli algoritmi in scala logaritmica;
* L’andamento degli algoritmi in base ai valori di k selezionati.

I grafici vengono prodotti tramite appositi script realizzati in linguaggio R, all’interno dell’ambiente RStudio. In particolare, i dati vengono prelevati dal file *Time.xlsx* attraverso la libreria **readxl**, mentre la visualizzazione grafica è resa possibile dalla libreria **ggplot2**.

Il seguente grafico permette la visualizzazione, in scala lineare, del comportamento della complessità dei tre algoritmi di selezione con input di ***targetSize*** inclusa fra i 100 e i 5 milioni di unità.

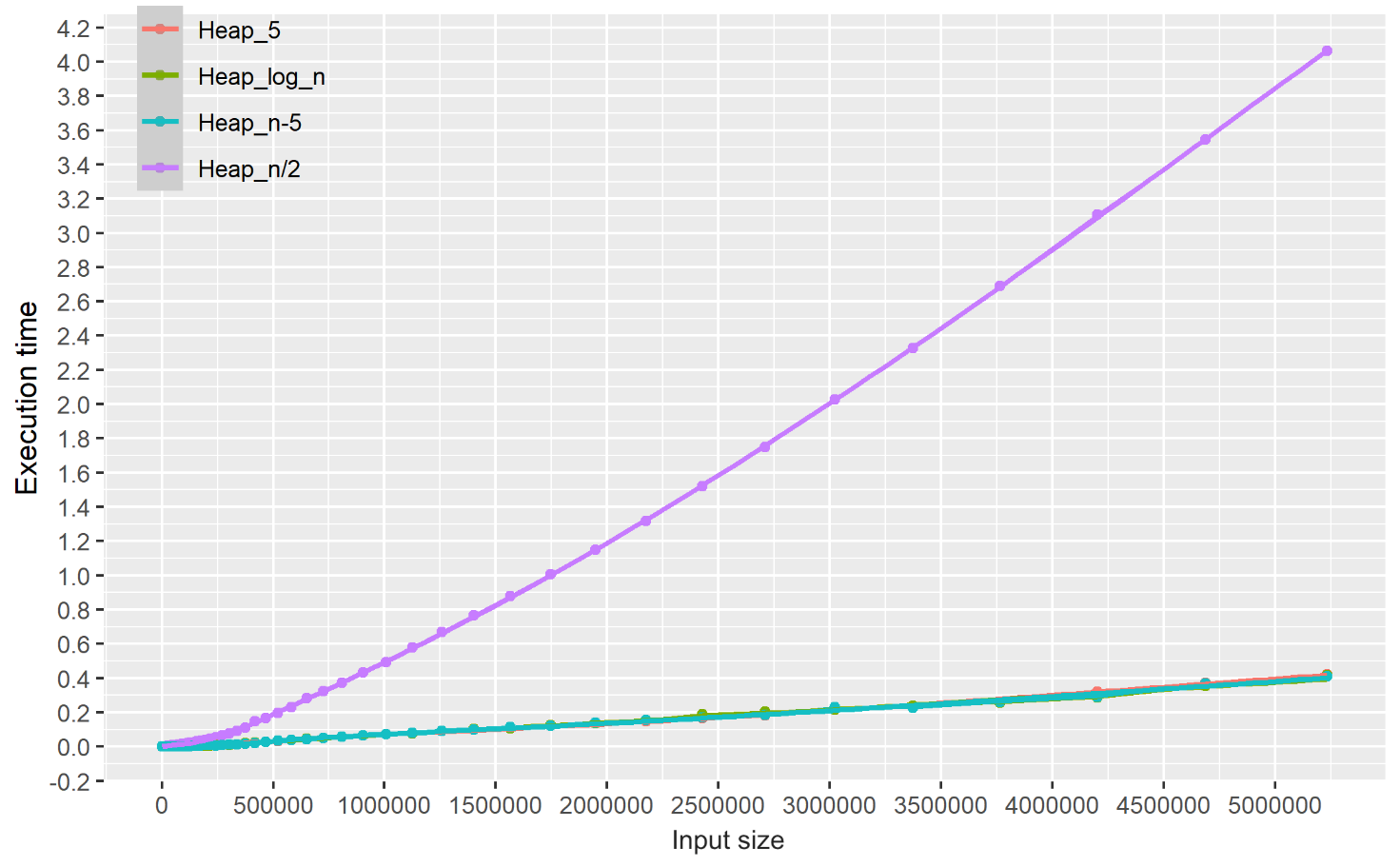


Si osserva come l’andamento descritto dagli algoritmi ***QuickSelect***e ***MedianSelect***si adatti ad un modello lineare, rispecchiando la complessità temporale precedentemente dimostrata.

É, inoltre, possibile osservare uno scostamento dall’andamento lineare (con successiva regolarizzazione) da parte dell’algoritmo ***HeapSelect***, dovuto alla complessità temporale descritta dall’insieme .

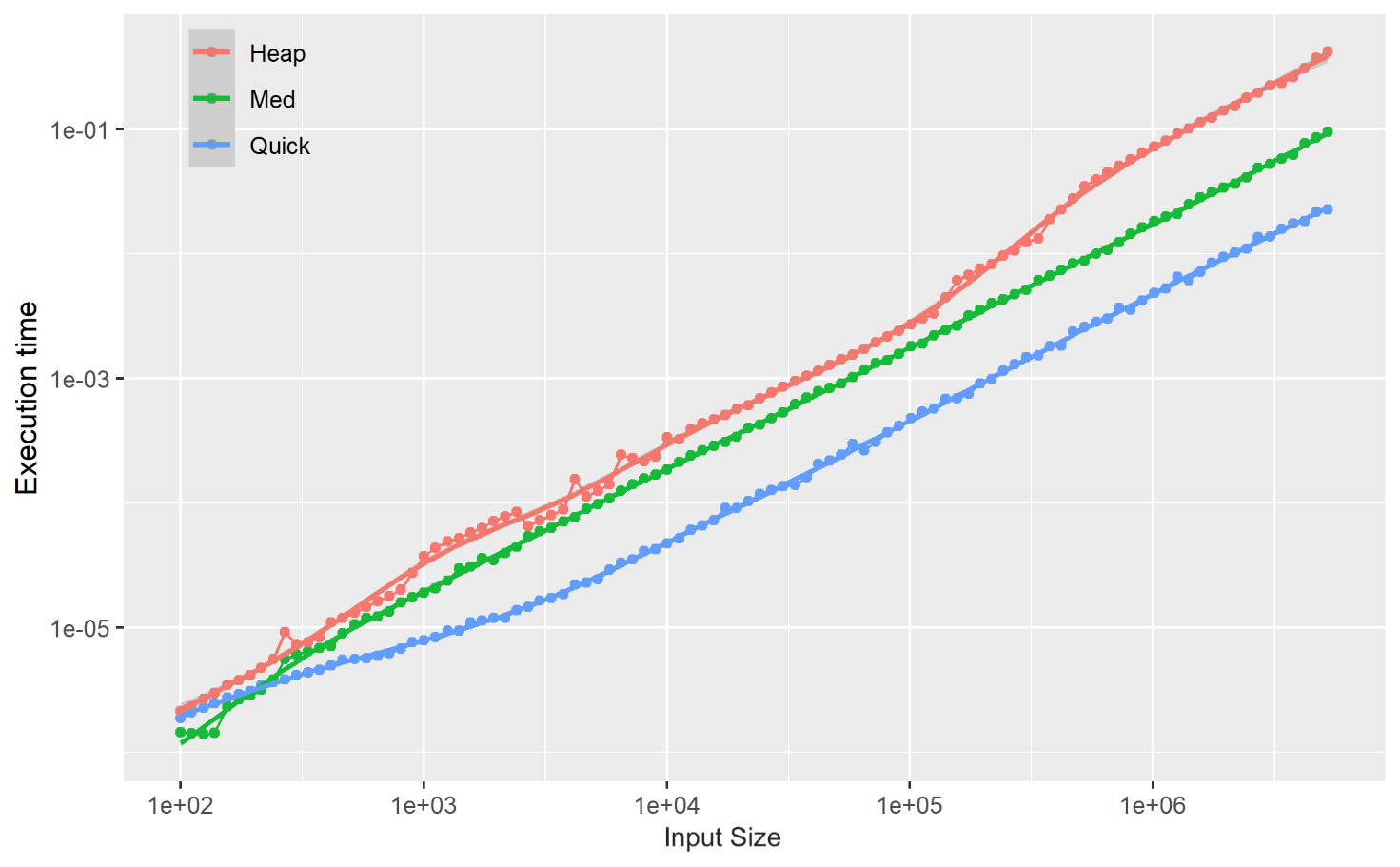
La diretta proporzionalità che sussiste fra gli indici di posizione e il parametro *targetSize* comporta, infatti, l’aumento del valore dell’indice sulla base della crescita della dimensione dell’input. In particolare, al raggiungimento di dimensioni pari o superiori alle 500.000 unità, la componente logaritmica della complessità causa uno scostamento rispetto al precedente andamento (visibile chiaramente nelle rappresentazioni grafiche). In seguito, la complessità temporale prosegue nuovamente descrivendo un modello di carattere lineare.

Osservando, in particolare, il comportamento dell’algoritmo ***HeapSelect*** sulla base degli indici ***k*** selezionati, si ottiene la seguente rappresentazione.



È possibile osservare come la ricerca di elementi di indice n, log(n) e n-5 descriva un andamento riconducibile ad un modello lineare, con una lieve curvatura dovuta al peso della componente logaritmica sulla complessità totale. La selezione di elementi di indice , corrispondente al caso pessimo dell’algoritmo, comporta invece una crescita elevata del tempo di esecuzione in forma direttamente proporzionale rispetto alla dimensionedel vettore in input.

Un ulteriore confronto della complessità dei tre algoritmi è realizzato applicando una funzione logaritmica ad entrambi gli assi cartesiani. Il risultato ottenuto conferma quanto osservato in precedenza.



# Conclusioni

Sulla base dei procedimenti presi in analisi, è stato possibile descrivere l’implementazione di tre noti algoritmi di selezione (***QuickSelect****,* ***HeapSelect****,* ***MedianOfMediansSelect****)*, dimostrandone la complessità spaziale e temporale. Successivamente, la misurazione e il raccoglimento dei tempi di esecuzione sulla base di campioni aleatori con un comprovato errore massimo di misurazione, hanno permesso l’osservazione di una netta congruenza fra le precedenti asserzioni teoriche e i risultati pratici, descritti graficamente.

Sulla base delle analisi effettuate è, inoltre, possibile confermare l’efficienza dell’algoritmo ***QuickSelect***, che ha dimostrato tempi di esecuzione medi inferiori rispetto a quelli degli altri algoritmi presi in analisi.