Relazione Laboratorio Algoritmi e Strutture Dati

Roccaforte Daniele 915088

Santoro Gabriele 9299594

Santella Carmine 950984

Università Degli Studi di Torino

A.A 2021/2022

INDICE

1 - Algoritmi di ordinamento

Implementazione di una libreria che offre due algoritmi di ordinamento *Quicksort* e *Binary Insertion Sort.* Si ordinino i *record* contenuti nel file records in ordine non decrescente secondo i valori contenuti nei tre campi "field".

1.1 - Metodologia

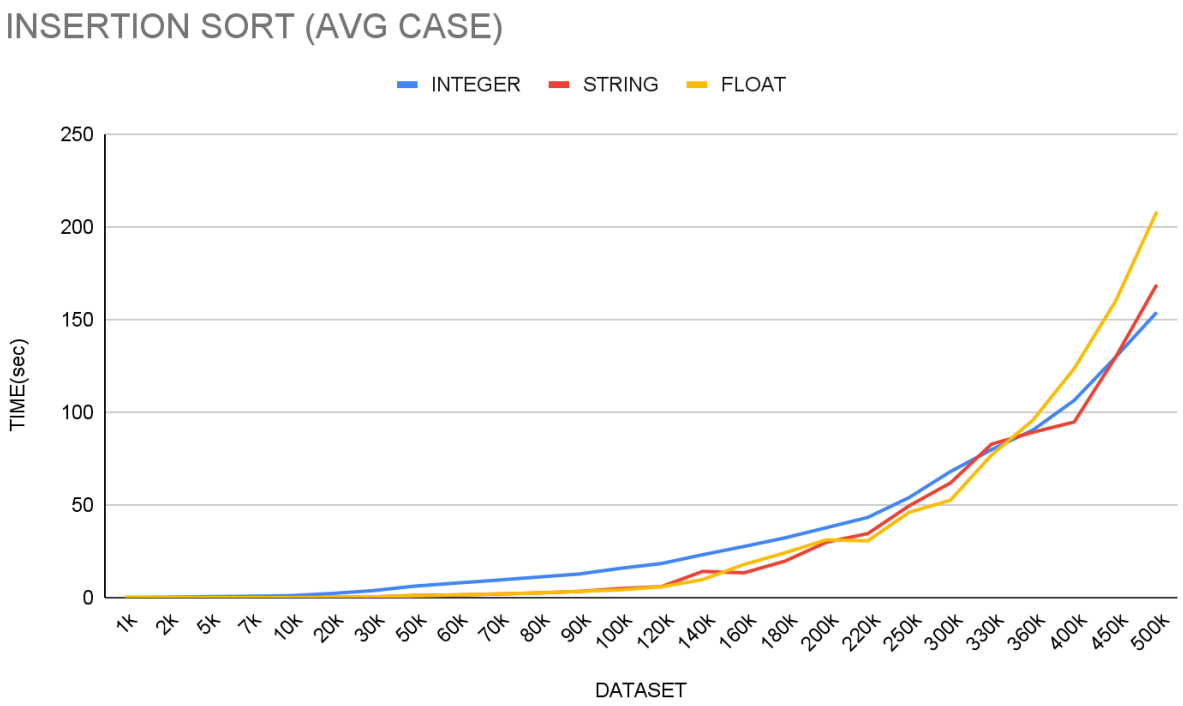
1.1.1 - Binary Insertion Sort

Con Binary Insertion Sort ci riferiamo a una versione dell'algoritmo Insertion Sort in cui la posizione all'interno della sezione ordinata del vettore in cui inserire l'elemento corrente è determinata tramite una ricerca binaria.

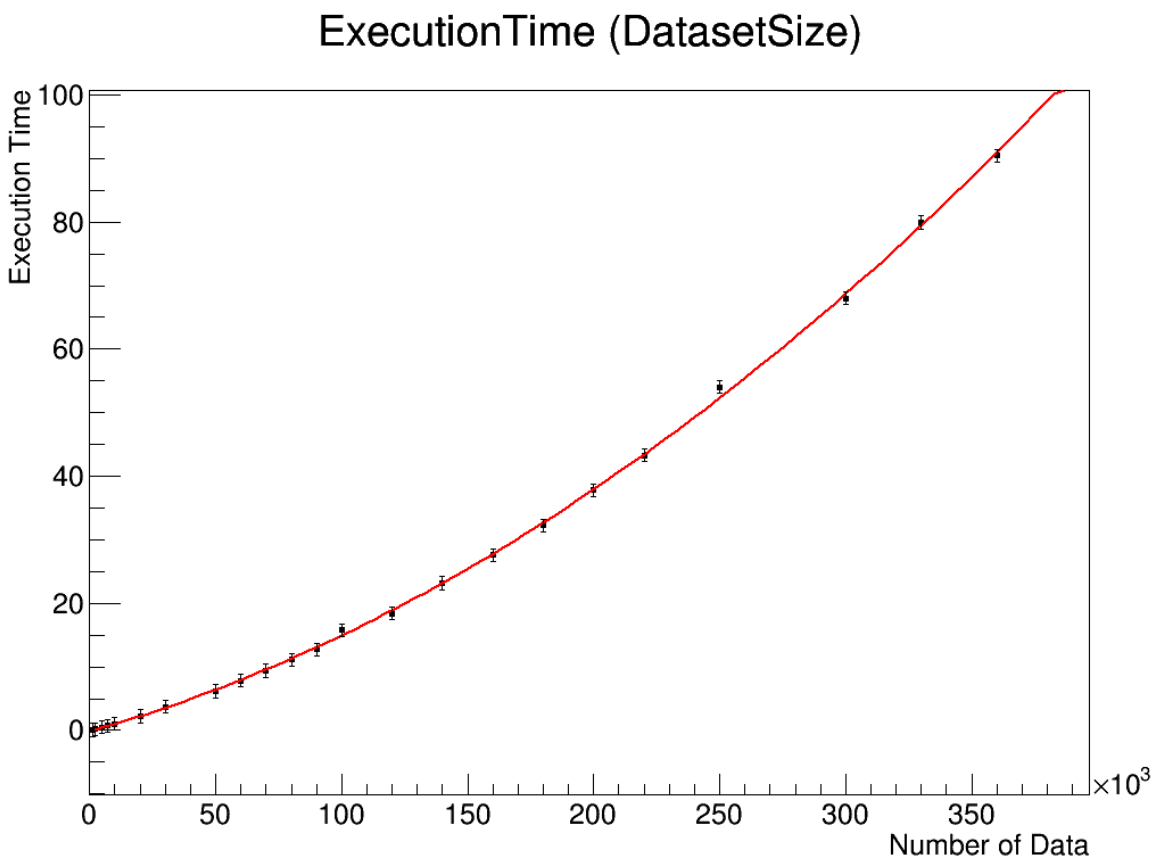
La libreria *"b_insertionsort"* offre il metodo pubblico *b_insertionsort* che, a partire da un *Array Struct* e da un criterio di comparazione, ordina i valori di un record.

1.1.1.1 - Risultato caso medio/peggiore

Lo studio della complessità temporale ha mostrato che la versione implementata nel caso peggiore ha complessità $O(n^2)$, quindi in linea con quella attesa. In particolare, dopo aver estratto dei risultati temporali da un numero svariato di test eseguiti su 26 diverse dimensioni del record fornito e sui tre tipi di parametri "field" da ordinare (int, float e string), abbiamo ottenuto i seguenti risultati:



Uno studio statistico ha confermato in maniera analitica la complessità dell'algoritmo. In particolare il **metodo dei minimi quadrati** ci ha permesso di trovare una funzione, rappresentata da una curva ottima, che si avvicini il più possibile al nostro insieme di dati. In questo caso essendo la funzione non lineare, non è possibile indicare un modo certo per ottenere i parametri. Per questo abbiamo utilizzato un framework di analisi dati sviluppato al CERN chiamato **ROOT:**

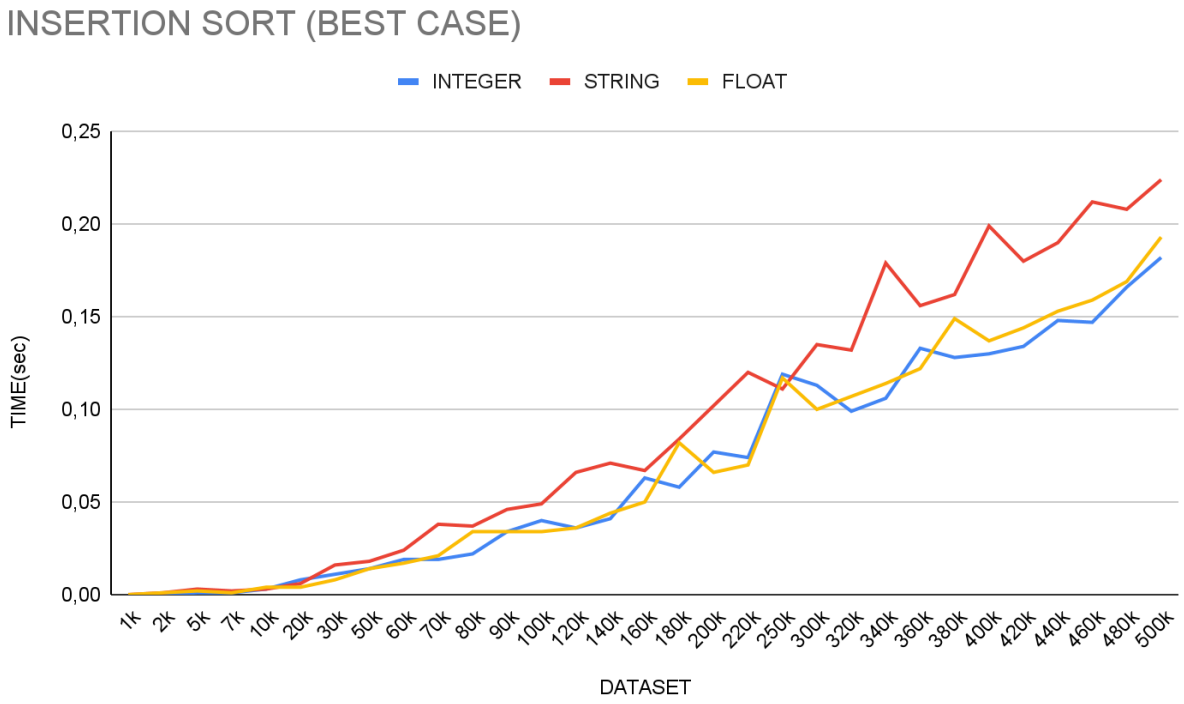


Il grafico in questione rappresenta con dei puntini neri i singoli dati estratti, all'interno di un intervallo di errore stimato, mentre la curva in rosso rappresenta la funzione che più approssima il nostro dataset.

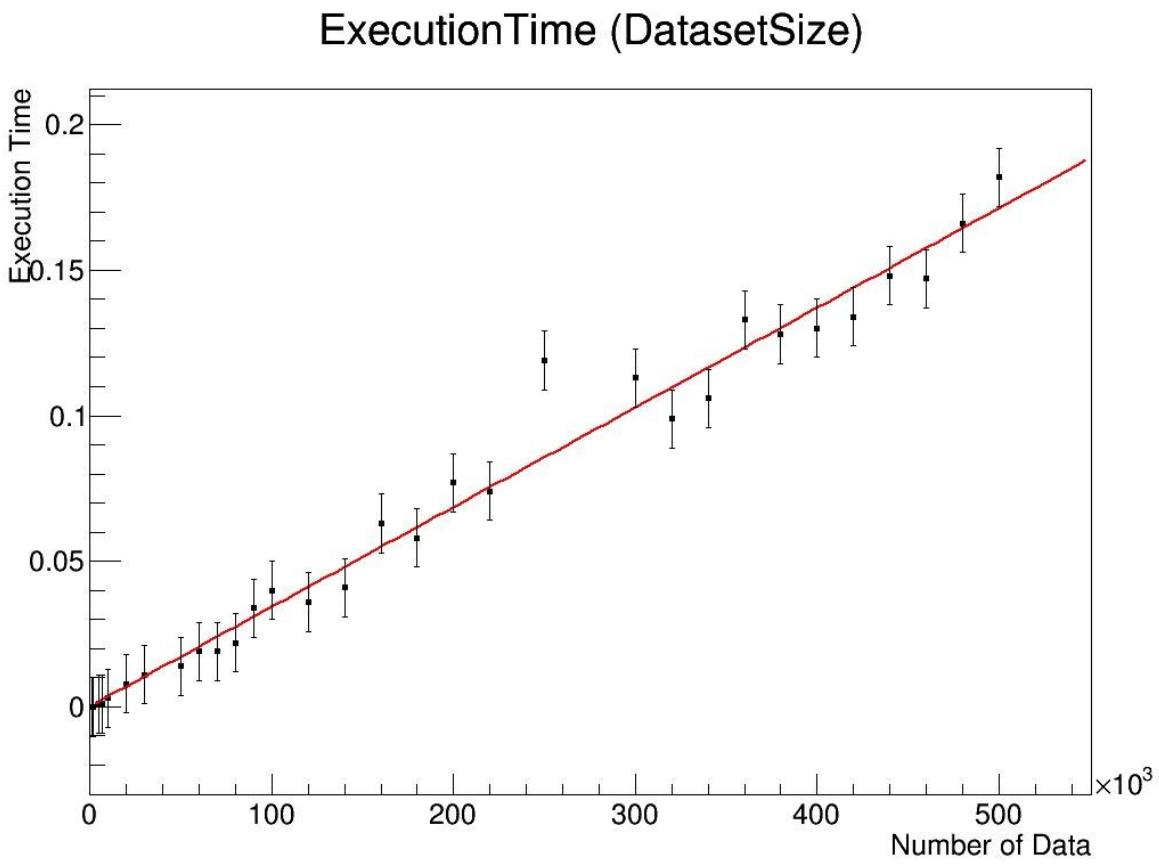
Senza entrare in troppi particolari matematici che non ci competono e che con grande onestà intellettuale non saremmo in grado di spiegare dettagliatamente, il framework utilizzato ci ha confermato l'andamento quadratico dei dati, con un p-value = 0,999587. Rigettando l'ipotesi di buon accordo con i dati, si avrebbe quindi una probabilità di commettere errore statistico di prima specie (ossia commettere falso negativo) maggiore rispetto alla massima fissata (0,05).

1.1.1.2 - Risultato caso migliore

Per poter testare l'algoritmo nel caso migliore, abbiamo testato l'algoritmo in questione con un insieme più numeroso di dati (31 elementi), questa volta già ordinati in modo da confermare la complessità lineare dell'algoritmo. Le modalità di test sono le stesse utilizzate precedentemente e queste hanno portato i seguenti risultati:



Anche se la funzione è lineare, abbiamo comunque deciso di utilizzare il framework precedentemente illustrato eseguendo in maniera pressoché uguale il **metodo dei minimi quadrati.**



I risultati che il framework ha restituito, hanno riportato un valore p-value = 0,7928.

1.1.2 - QuickSort

Quicksort è un algoritmo di ordinamento ricorsivo in place non stabile. Tale procedura ricorsiva viene comunemente detta Partition: preso un elemento chiamato "pivot" da una struttura dati si pongono gli elementi minori a sinistra rispetto al pivot e gli elementi maggiori a destra. L'operazione viene quindi reiterata sui due insiemi risultanti fino al completo ordinamento della struttura.

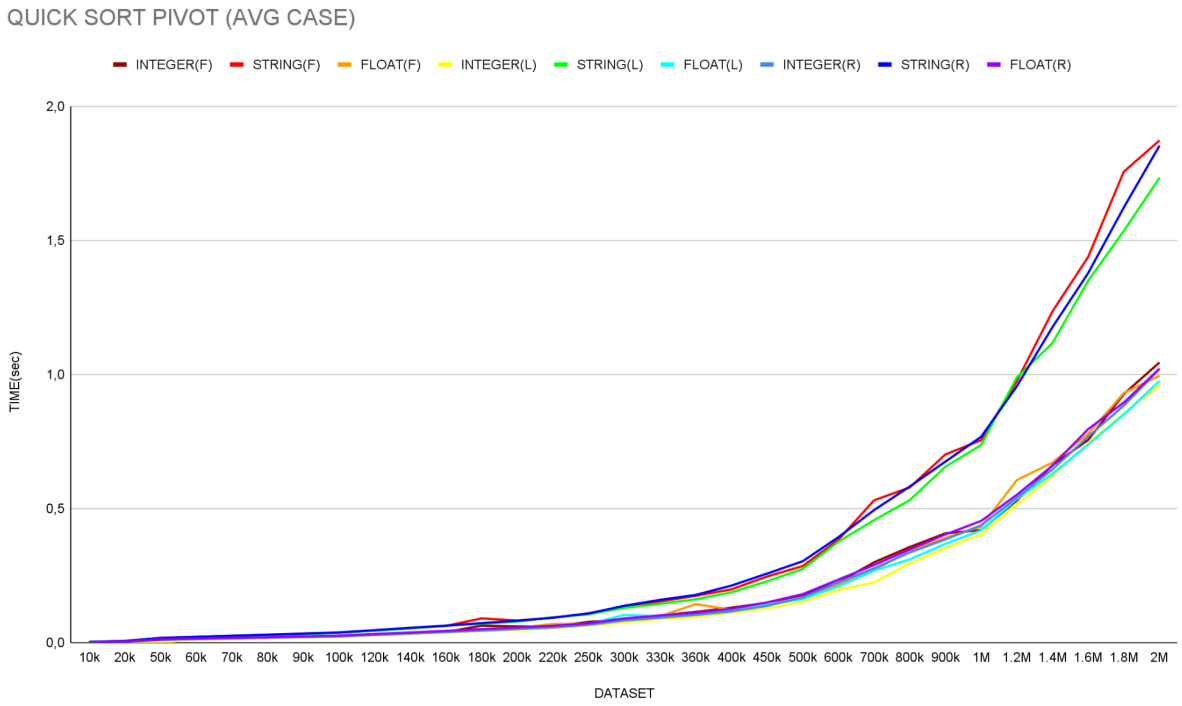
La libreria *"quicksort.h"* offre il metodo *quick_sort* che riceve come parametri: un ArrayStruct, un criterio di comparazione, i valori degli intervalli su cui applicare l'algoritmo ed il criterio di scelta dell'elemento pivot, il quale può variare tra: 0 - Primo elemento, 1 - Ultimo elemento, 2 - Elemento casuale.

All'interno della libreria, vi sono poi tre diverse versioni della funzione *Partition*, ognuna delle quali viene richiamata autonomamente a seconda della scelta del pivot.

1.1.2.1 - Risultato caso medio/migliore

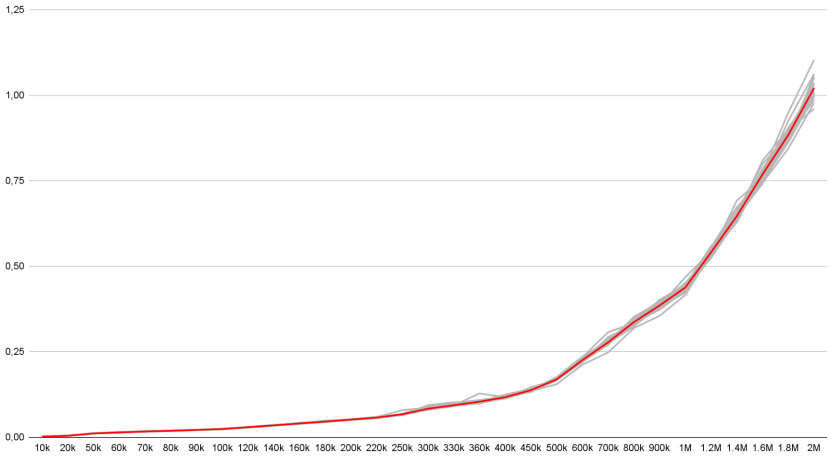
Lo studio della complessità temporale ha mostrato che la versione implementata nel caso medio/migliore è O($nlogn$), quindi in linea con quella attesa.

I testi sono stati effettuati su un insieme di 31 dataset differenti che hanno portato al seguente risultato:

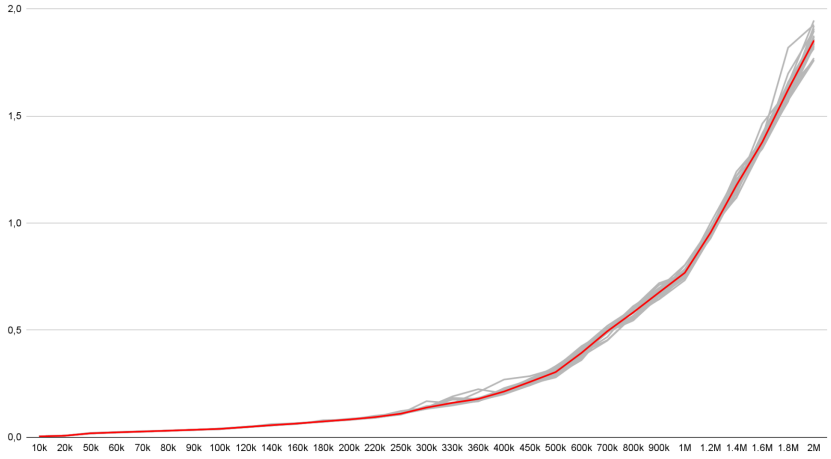


I risultati per la scelta del pivot in maniera casuale sono stati ricavati attraverso la media aritmetica di 25 test svolti su uno stesso dataset, ripetuti per tutti e 31 i dataset. In questo modo possiamo appunto ottenere un valore meno influenzato sul singolo test:

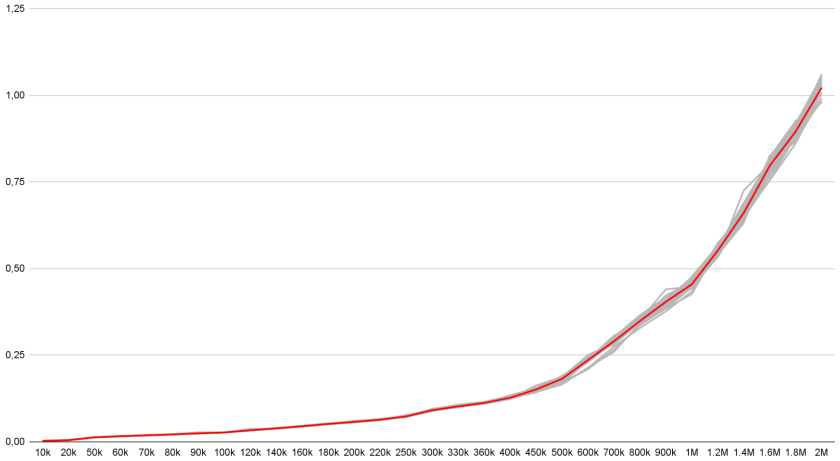
QUICK SORT (AVG CASE) INTEGER PIVOT = RANDOM

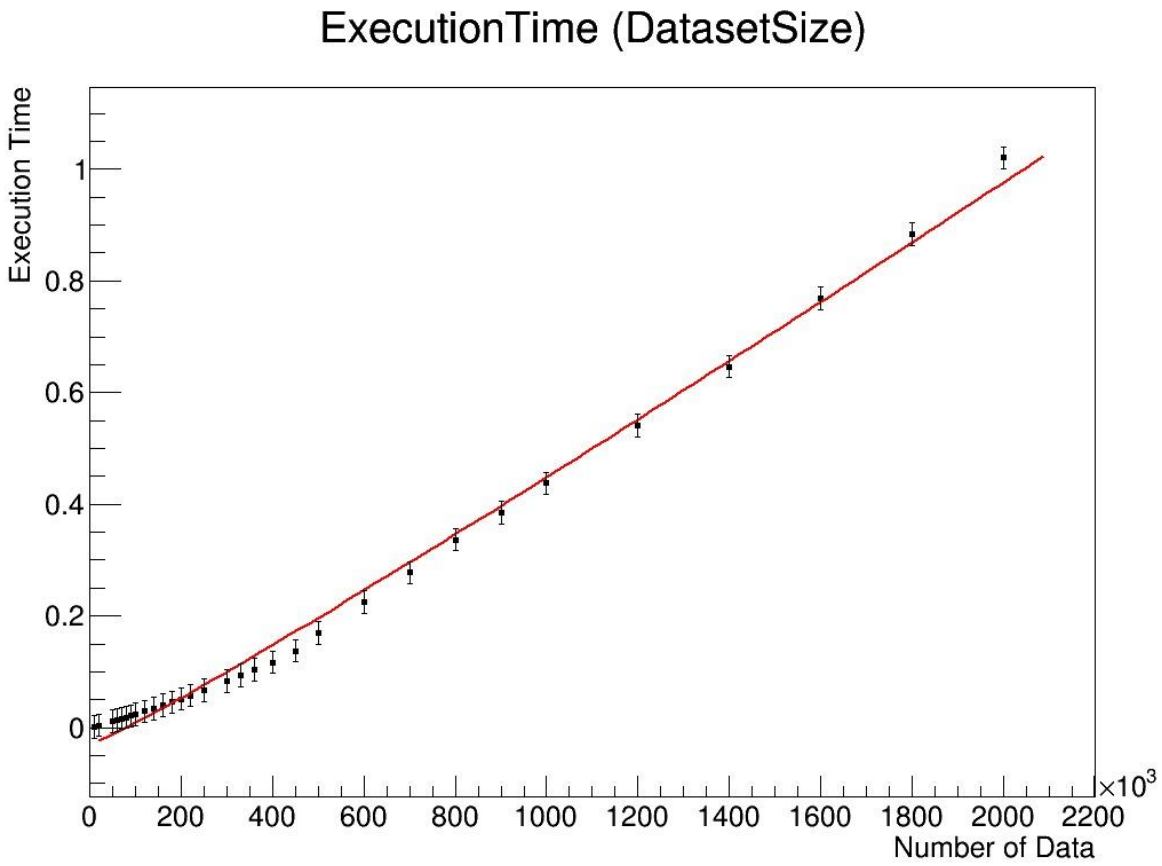


QUICK SORT (AVG CASE) STRING PIVOT = RANDOM



QUICK SORT (AVG CASE) FLOAT PIVOT = RANDOM



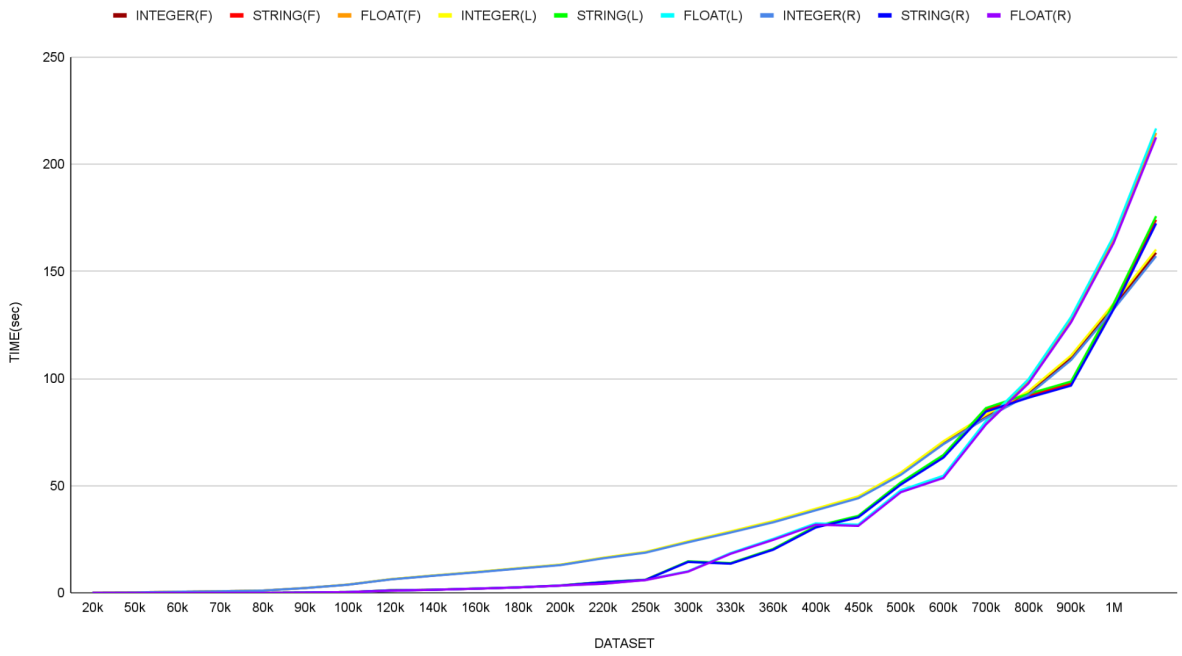


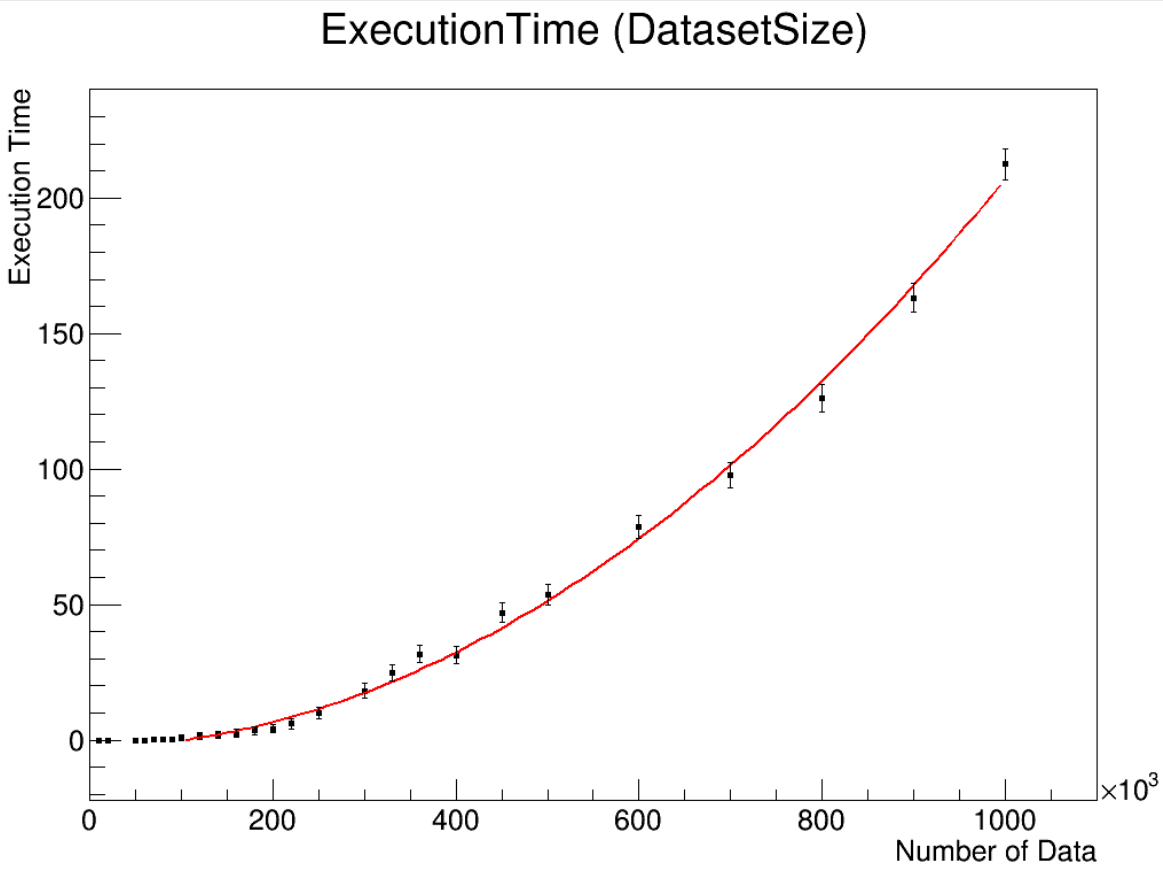
I risultati che il framework ha restituito attraverso il metodo dei minimi quadrati, hanno riportato un valore p-value = 0,38995.

1.1.2.2 - Risultato caso peggiore

Per poter testare il caso peggiore sono stati utilizzati 26 data-set ordinati di lunghezza differente, che hanno mostrato l'andamento esponenziale dell'algoritmo. Come per il caso precedente, per avere un valore realistico e poco derivante dalla scelta del pivot randomico, sono stati effettuati 25 test con la scelta del pivot in maniera casuale sullo stesso dataset; tutto ciò ripetuto per tutti i data-set.

QUICK SORT PIVOT (WORST CASE)





I risultati che il framework ha restituito attraverso il metodo dei minimi quadrati, hanno riportato un valore p-value = 0.200855.

Anche se il metodo dei minimi quadrati abbia restituito dei valori del pivot apparentemente bassi. Questi sono comunque sufficienti a poter confermare l'andamento atteso delle nostre funzioni, questo perché in questo test si assume come valore minimo sufficiente quello di 0,05 ed i nostri valori hanno comunque superato ampiamente questo valore.

2 - Skip List

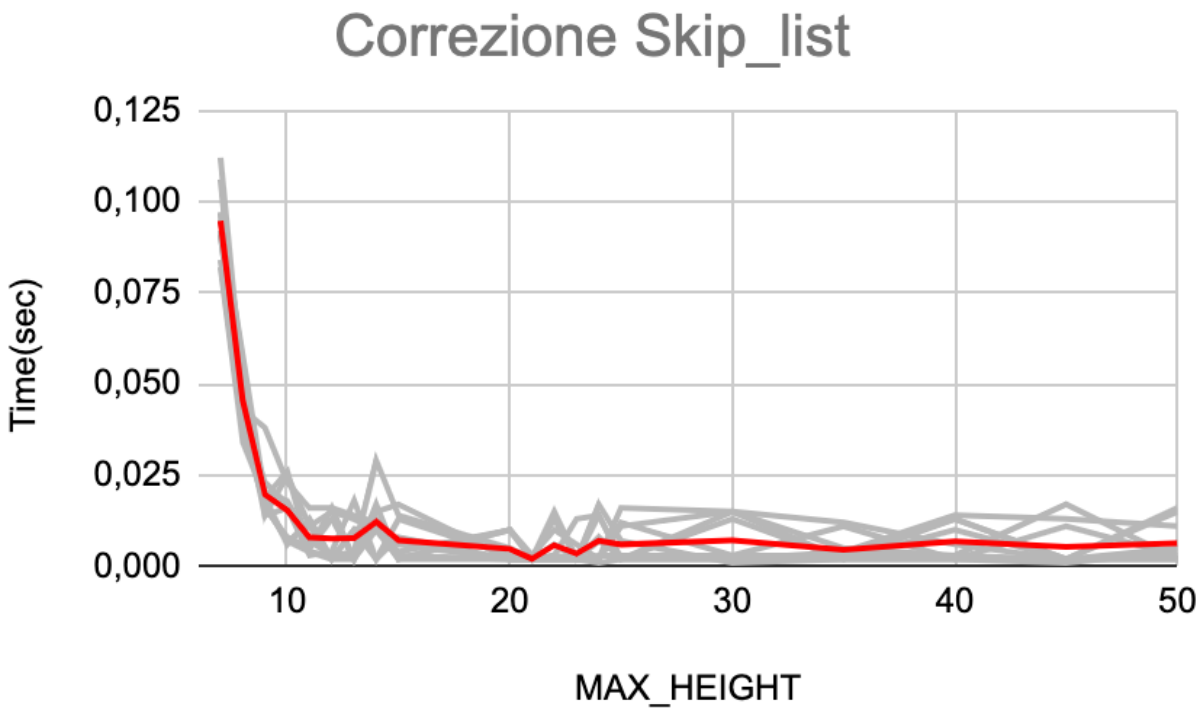
Realizzare una struttura dati chiamati "Skip List". La skip list è un tipo di lista concatenata che memorizza una lista ordinata di elementi. Grazie all'array di puntatori mantenuto da ogni nodo della lista è possibile, nelle operazioni che richiedono una ricerca/scansione saltare parte dei nodi, per questo motivo la skip list è utilizzata spesso per indicizzare i dati.

2.1 - Metodologia

Per l'implementazione abbiamo realizzato due tipologie di liste: la prima è la skip list richiesta dalla consegna che conterrà il data set di parole contenute nel dizionario, che verranno scansionate; la seconda è una semplice lista che conterrà il data set di parole da confrontare con il dizionario e che verrà scansionata in modo lineare. Nella skip list è possibile inoltre definire il valore di MAX_HEIGHT.

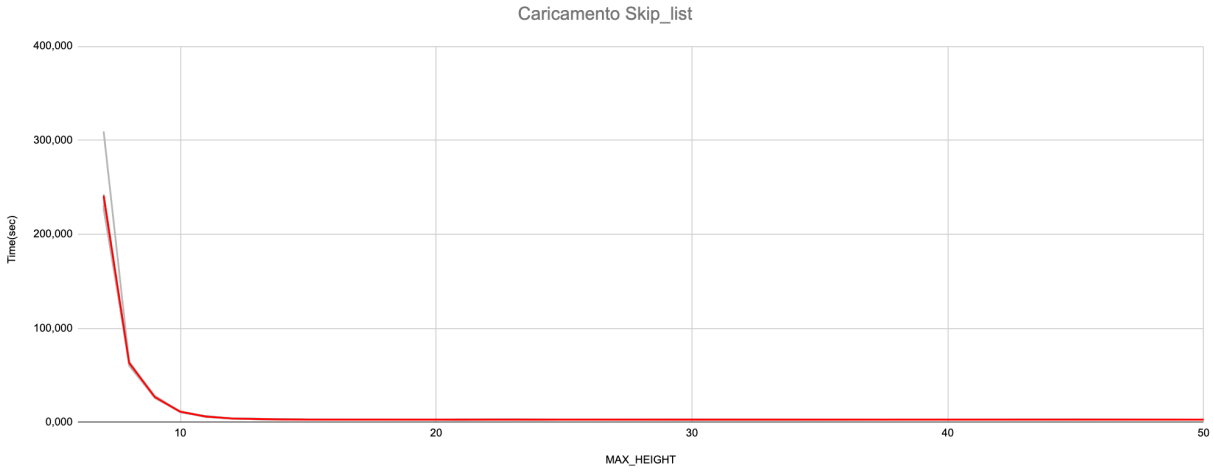
Abbiamo effettuato i test su una struttura dati di 600k elementi (stringhe) di parole di una frase ed analizzato le tempistiche richieste per la ricerca e il caricamento, al variare di MAX_HEIGHT. In particolare sono stati effettuati per lo stesso valore di MAX_HEIGHT 10 test, questo perché anche se il valore è fissato, alla creazione della skip list, ogni "nodo" avrà un valore casuale di altezza scelto tra 1 e MAX_HEIGHT che potrebbe influenzare anche se non di molto i risultati. L'analisi sui tempi di scansione/correzione ha restituito i seguenti risultati:

2.1.1 - Risultati



La linea tratteggiata rossa rappresenta il valore medio dei 10 test. Si può notare come per un valore di MAX_HEIGHT minore di 10 i tempi di correzione aumentino in maniera vertiginosa. E' dunque conveniente scegliere un valore di MAX_HEIGHT maggiore almeno di 10.

Lo stesso vale per i tempi di caricamento ed ordinamento della skip list:



Anche per questo caso i tempi di caricamento crescono in maniera repentina per valori di MAX_HEIGHT minori di 10.

2.2 - Miglioramento

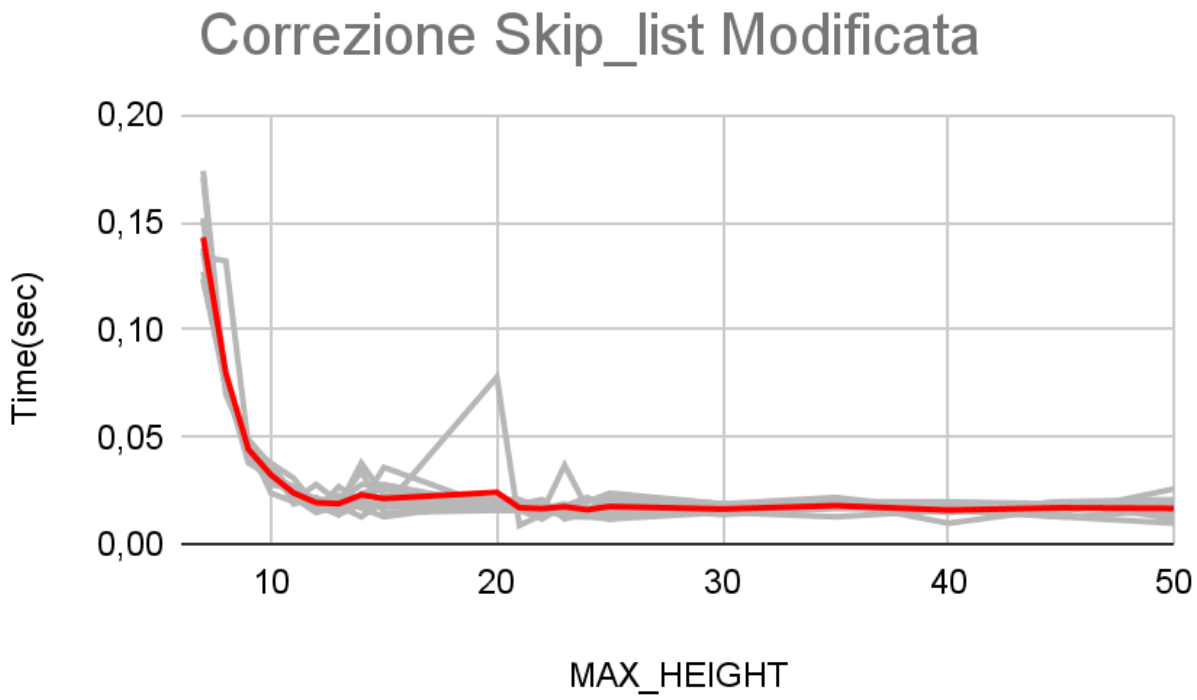
Abbiamo successivamente individuato, in discussione col professore, una possibile ottimizzazione nella ricerca di un elemento nella Skip List

L'idea è quella di verificare che, dopo essere usciti dal while avendo violato la condizione *(skiplist->compare(x->next[i]->item, I)) == 1)*, ovvero avendo trovato un elemento >= dell'elemento cercato, l'elemento trovato(successivo) non sia uguale all'elemento cercato, evitando così di dover diminuire di livello nel nodo fino a dover arrivare all'altezza 1.

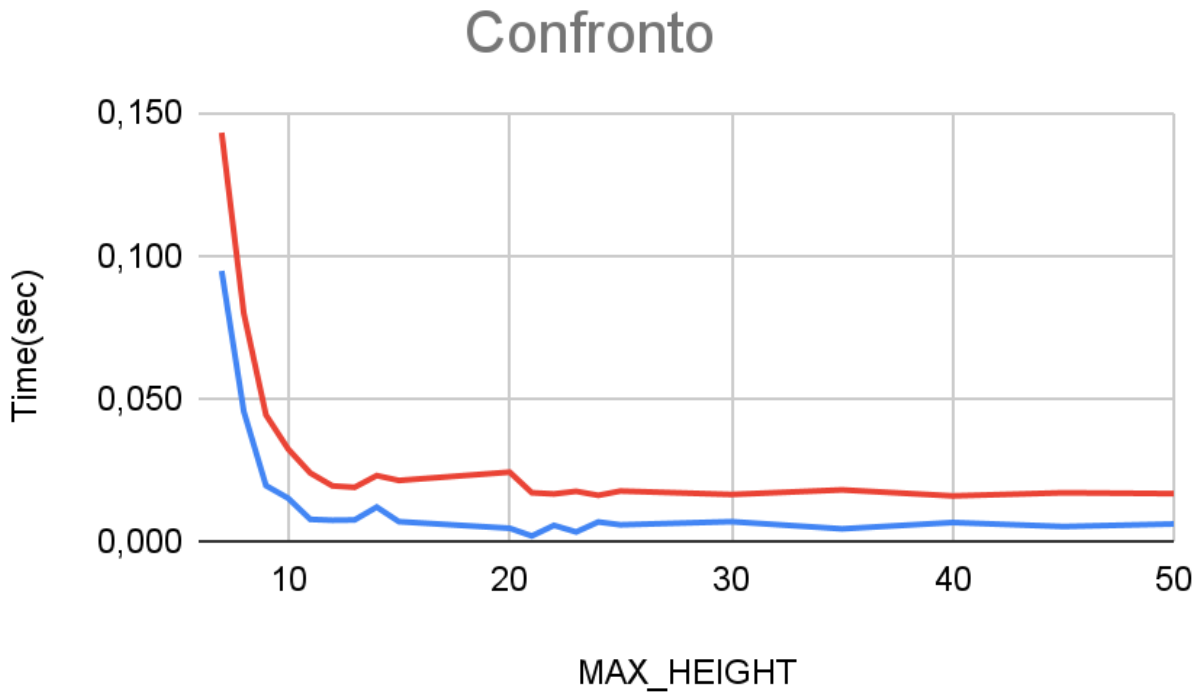
```
void* searchSkipList_modified(SkipList *skiplist, void* I){
  Node *x = skiplist->head;
  // loop invariant: x->item < I
  for(int i = (skiplist->max_level)-1; i > -1; i--){
    if(x->next[i] == NULL){
        //Puntatore di livello i è uguale a NULL
    }else{
      while((x->next[i] != NULL) && (skiplist->compare(x->next[i]->item, I)) == 1){
        x = x->next[i];
      }
      if((x->next[i] != NULL)){
        if((skiplist->compare(x->next[i]->item, I)) == 0){
          return x->next[i]->item;
        }
      }
    }
  }
  if(x->next[0] != NULL){
    x = x->next[0];
    if ((skiplist->compare(I, x->item)) == 0){
      return x->item;
    }else{
      return NULL;
    }
  }else{
    return NULL;
  }
}
```

2.2.1 - Risultati

Abbiamo quindi testato nelle medesime condizioni questa versione modificata della ricerca e questi sono stati i risultati:



Abbiamo confrontato i valori delle due versioni e di seguito i risultati:



Si può evidenziare in rosso il grafico dei tempi di esecuzione della versione modificata della funzione di ricerca della skip list, in blu invece la versione standard. Possiamo quindi concludere che nonostante la nostra idea potesse risultare più efficiente, nella sua attuazione pratica risulta seppur dello stesso grado di complessità di qualche costante più inefficiente. In particolare la differenza media tra le due versioni è di circa 0,02%.

3 - Heap

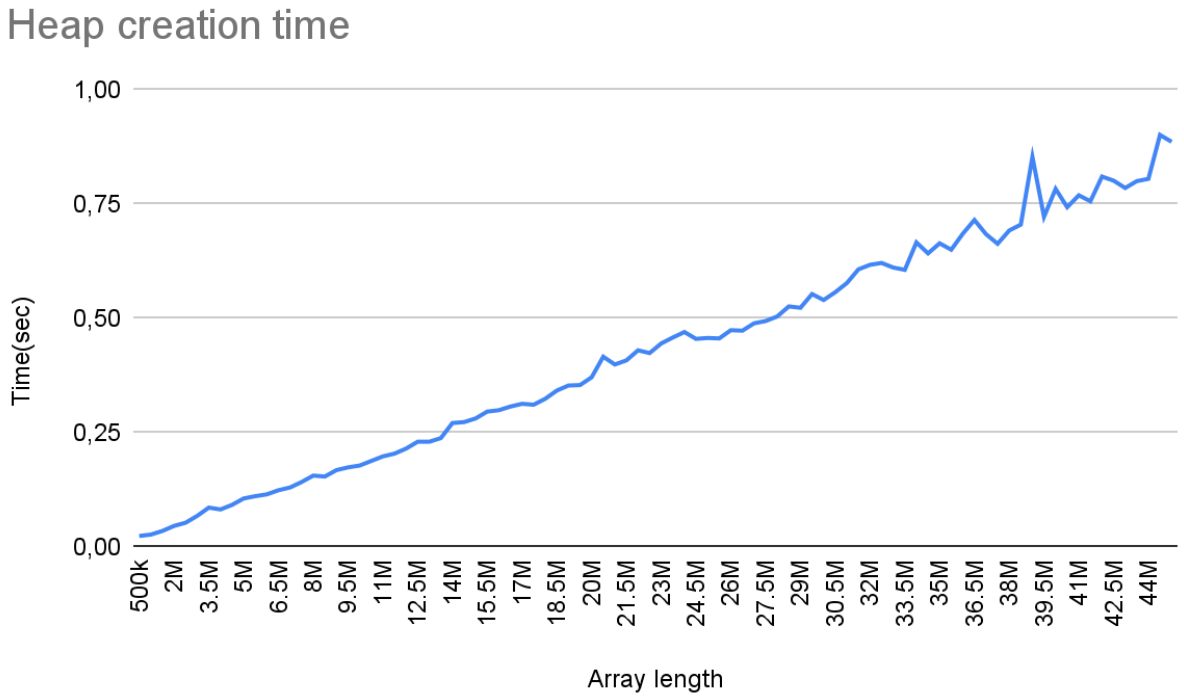
3.1 - Metodologia

Si implementi una libreria che realizza la struttura dati Heap Minimo. La struttura dati deve:

- **Rappresentare internamente lo heap tramite un vettore:** è stata implementata all'interno del file "MinHeap.java" una classa che implementa lo Heap Minimo la quale contiene i seguenti parametri:
    - "length:int" : rappresenta la lunghezza dell'array;
    - "heap_size:int" : rappresenta la grandezza dello heap;
    - "array:ArrayList<T>" : rappresenta l'array con cui viene rappresentato lo heap;
    - "comparator:Comparator<?>" : rappresenta il criterio di comparazione degli elementi nello heap
- **consentire un numero qualunque e non noto a priori di elementi dello heap:** avendo utilizzato una struttura dati dinamica è possibile inserire un qualsiasi numero di elementi.

Sono stati analizzate le prestazioni temporali su alcune delle operazioni offerte dalla libreria, nello specifico sono stati effettuati 135 test su un insieme di record che variano da 500k a 67.5M di elementi ed i risultati sono i seguenti:

- **inserimento di un elemento - O(log n):** l'analisi di prestazione di questa funzione è inestimabile in termini di tempo. È stato infatti effettuato un test fino ad un array di 160M di elementi ed è stato restituito un errore di: *Exception in thread "main" java.lang.OutOfMemoryError: Java heap space.*
- **costruzione dello heap minimo partendo da un array esistente:**



- **estrazione dell'elemento con valore minimo:**

4 - Grafo

Si implementi una libreria che realizza la struttura dati Grafo in modo che sia **ottimale** per dati sparsi. La struttura deve consentire di rappresentare sia grafi diretti che grafi non diretti.

4.1 - Metodologia

La struttura dati implementata, chiamata Graph.java offre le seguenti operazioni:

- Creazione di un grafo vuoto:

```
public Graph()
```

- Aggiunta di un nodo:

```
public void addNode(T node, W distance)
public void addNode(T node)
```

- Aggiunta di un arco:

```
public void addEdge(Edge<T , W> edge
```

- Verificare se il grafo è diretto:

```
public boolean isOriented()
```

- Verificare se il grafo contiene un dato nodo:

```
public boolean containNode(T node)
```

- Verificare se il grafo contiene un dato arco:

```
public boolean containEdge(Edge<T , W> edge)
```

- Cancellazione di un nodo:

```
public boolean removeNode(T node)
```

- Cancellazione di un arco:

```
public boolean removeEdge(Edge<T , W> edge)
```

- Determinazione del numero di nodi:

```
public int getNodesNum()
```

- Determinazione del numero di archi:

```
public int getEdgesNum()
  getEdgesNum(T node)
```

- Recupero dei nodi del grafo:

```
public Set<T> getNodes()
```

- Recupero degli archi del grafo:

```
public Collection<LinkedList<Edge<T , W>>> get Edges()
```

- Recupero nodi adiacenti di un dato nodo:

```
public ArrayList<T> adj(T node)
```

- Recupero etichetta associata a una coppia di nodi:

```
public Edge<T , W> getEdge(T source, T destination)
```

Con una serie di operazioni aggiuntive come quelle di stampa etc…

Si sono inoltre implementate le classi:

Edge → rappresenta un arco. Quest'ultimo può essere creato con o senza un valore di "peso". La funziona al suo interno presenta inoltre tutte le funzioni necessarie per ricavare il nodo sorgente, destinazione il peso etc…

ValueKey → permette di associare alla chiave della nostra HashMap più valore. Nel dettaglio:

```
private W distance;
   private T predecessor;
   private boolean flag;
   private LinkedList<Edge<T,W>> edges;
```