Gebze Technical University
Computer Engineering

CSE 222 - 2018 Spring

HOMEWORK 5 REPORT

AHMET MERT GÜLBAHÇE
141044015

1 Double Hashing Map

Double hashing is a collision resolving technique in Open Addressed Hash tables. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

For this part I used HashtableOpen.java which implements Map interface from course book. I implented only the methods which we saw in the class. Other methods body is empty.

Methods:

- `V get(Object key);`
- `V put(K key, V value);`
- `V remove(Object key);`
- `int size();`
- `boolean isEmpty();`

1.1 Pseudocode and Explanation

In course book, this class implemented by using linear probing. So I rewrote put and remove methods by using double hashing if there is collusion. To find the index where the item will be inserted I wrote two function which calculates the hashcode of given key.

- `calculateHashCode(Object key)` calculates the hashcode of given key
- `calculateStepSize(Object key)` calculates the step which will be added to hashcode index if there is collusion. If there is collusion we add the step to hashcode index and find hashcode index again. This process will continue untill there is no collusion.

For the table size I give a prime number `67`.
`LOAD_THRESHOLD` is `0.85`.

1.2 Test Cases

To test Double Hashing Map I put elements where

- `Key`: City in Turkey
- `Value`: Phone Code

Example: Key:"Adana", Value:322

Then test all map methods.

```java
public static void main(String[] args){

    HashtableOpen<String,Integer> hashtable = new HashtableOpen<>();

    hashtable.put("Adana",322);
    hashtable.put("Maras",344);
    hashtable.put("Kocaeli",262);
    hashtable.put("Konya",332);
    hashtable.put("Malatya",422);
    hashtable.put("Aksaray",382);
    hashtable.put("Manisa",236);
    hashtable.put("Antalya",242);
    hashtable.put("Mugla",252);
    hashtable.put("Artvin",466);
    hashtable.put("Aydin",256);
    hashtable.put("Batman",488);
    hashtable.put("Bayburt",458);
    hashtable.put("Bilecik",228);
    hashtable.put("Sakarya",264);
    hashtable.put("Sinop",368);

    System.out.println("size() => "+ hashtable.size());
    System.out.println("getting Sinop => " + hashtable.get("Sinop"));
    System.out.println("put Sinop 500 => " + hashtable.put("Sinop",500));
    System.out.println("getting Sinop => " + hashtable.get("Sinop"));

    System.out.println();
    System.out.println("isEmpty() => "+ hashtable.isEmpty());

    System.out.println("removing Adana => "+hashtable.remove( key: "Adana"));
    System.out.println("getting Adana => " + hashtable.get("Adana"));
    System.out.println("removing Maras => "+hashtable.remove( key: "Maras"));

    System.out.println("size() :  "+ hashtable.size());
}
```

Screenshot 1: *Screenshot of Main and Map Pairs*

```
size() => 16
getting Sinop => 368
put Sinop 500 => 368
getting Sinop => 500

isEmpty() => false
removing Adana => 322
getting Adana => null
removing Maras => 344
size() :  14
```

Screenshot 2: *Screenshot of result*

2 Recursive Hashing Set

This part is not done.

3 Sorting Algortihms

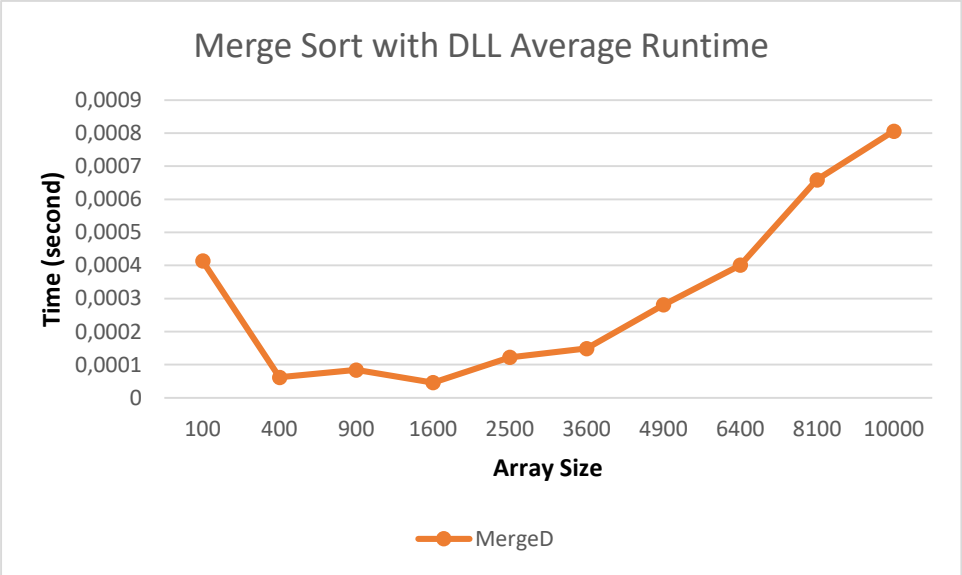
3.1 MergeSort with DoubleLinkedList

3.1.1 Pseudocode and Explanation

Merge sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.

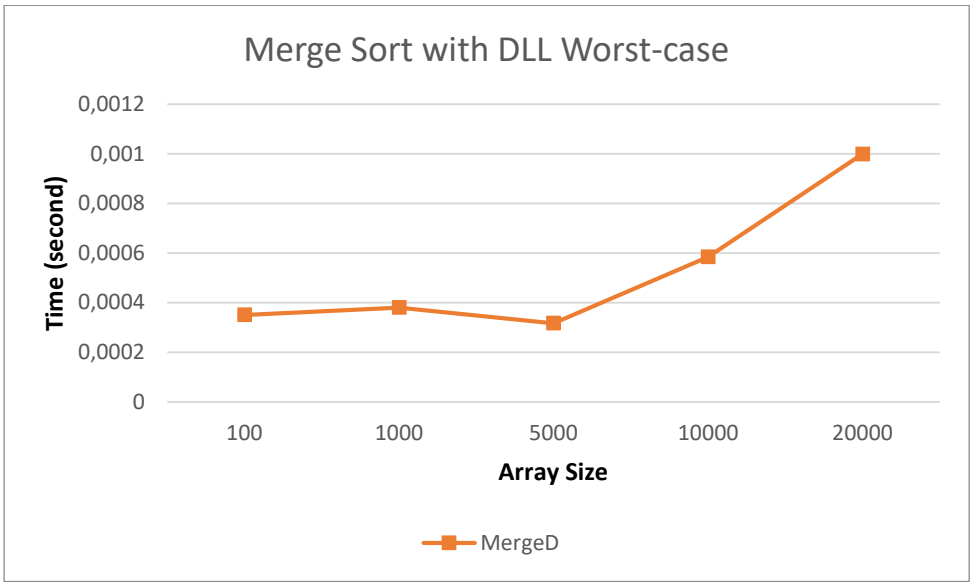
```
sort(DoubleLinkedList.Node node)
If r > l
     1. Find the middle point to split the array into two halves:
             second = split(node)
     2. Call sort for first half:
             first = sort(first)
     3. Call mergeSort for second half:
             second = sort(second)
     4. Merge the two halves sorted in step 2 and 3:
             merge(first,second)
```

3.1.2 Average Run Time Analysis



Graphic 1: *Merge Sort with DLL Average Runtime*

3.1.3 Worst-case Performance Analysis

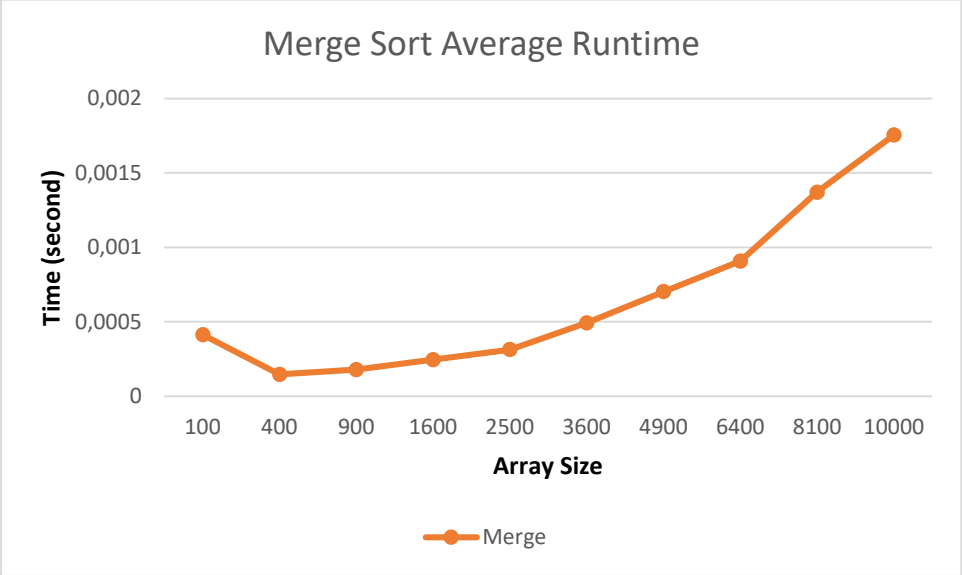


Graphic 2: *Merge Sort with DLL Worst-case*

3.2 MergeSort

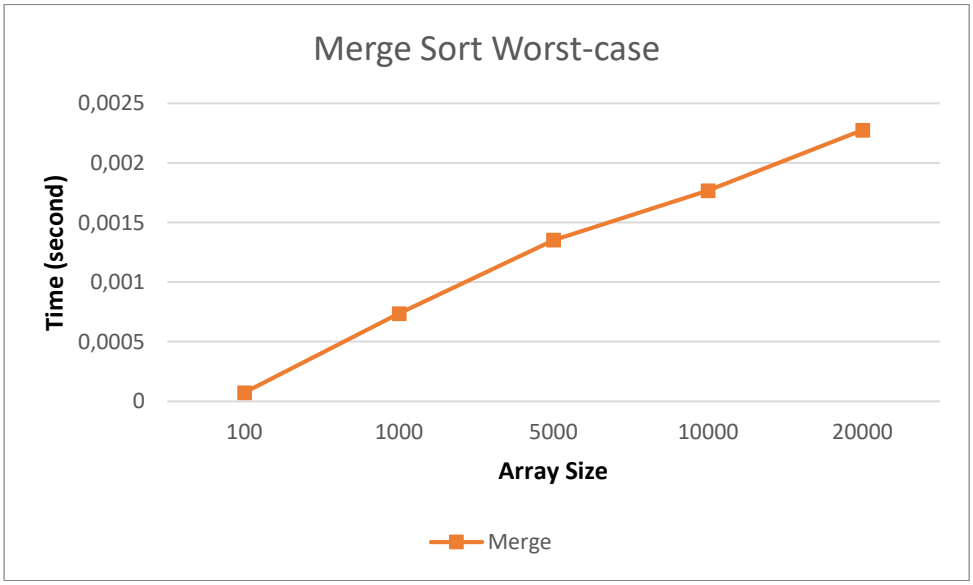
Merge sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves.

3.2.1 Average Run Time Analysis



Graphic 3: *Merge Sort Average Runtime*

3.2.2 Wort-case Performance Analysis

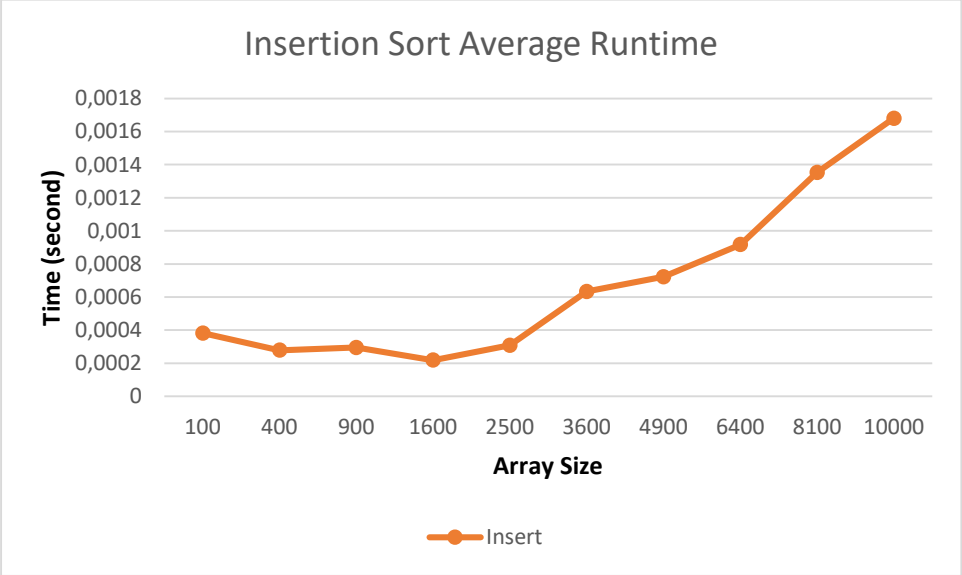


Graphic 4: *Merge Sort Worst-case*

3.3 Insertion Sort

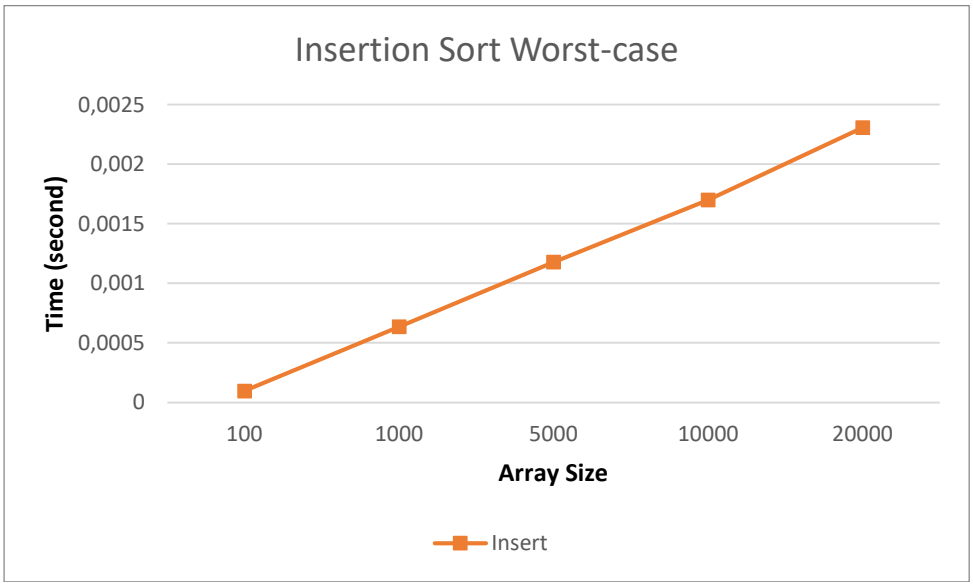
Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort.

3.3.1 Average Run Time Analysis



Graphic 5: *Insertion Sort Average Runtime*

3.3.2 Worst-case Performance Analysis

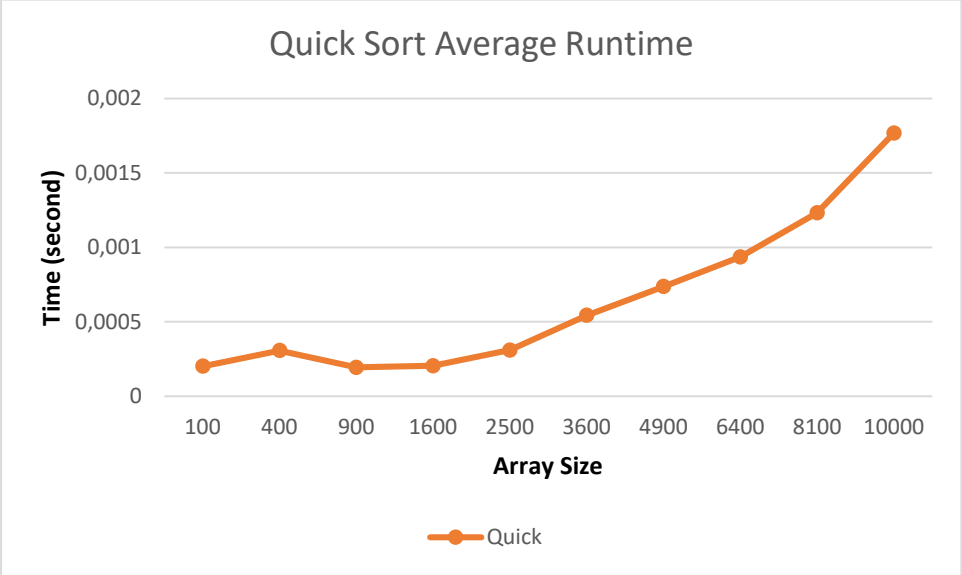


Graphic 6: *Insertion Sort Worst-case*

3.4 Quick Sort

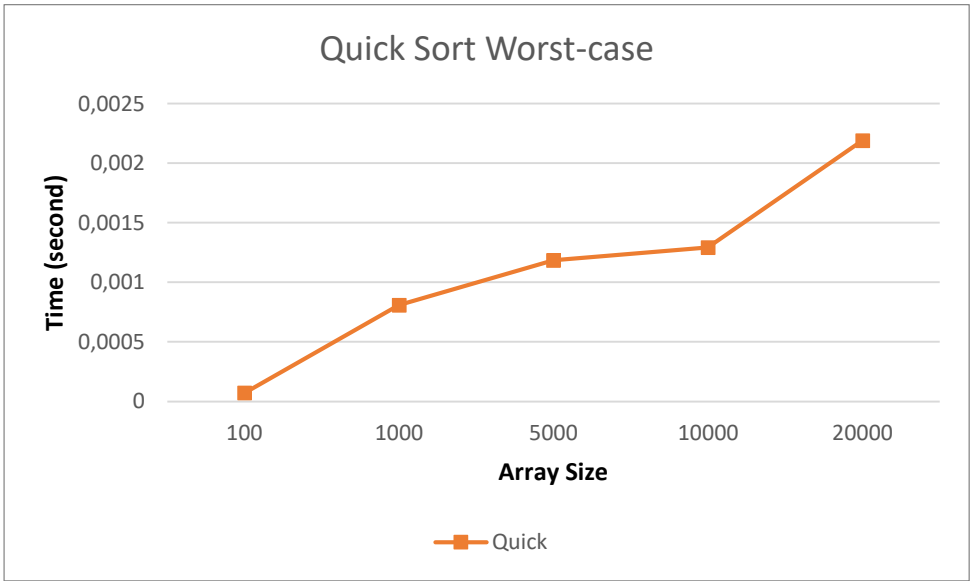
Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

3.4.1 Average Run Time Analysis



Graphic 7: *Quick Sort Average Runtime*

3.4.2 Worst-case Performance Analysis

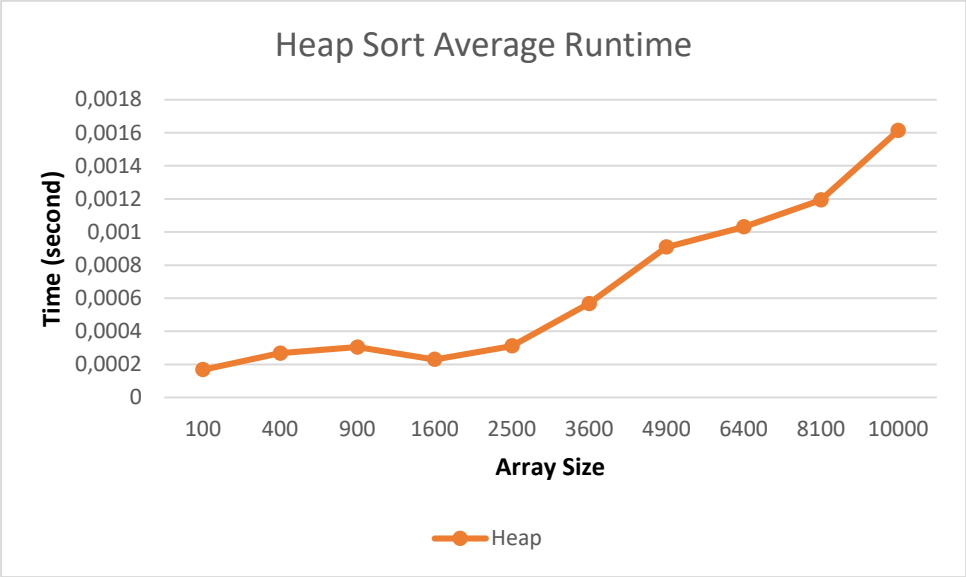


Graphic 8: *Quick Sort Worst-case*

3.5 Heap Sort

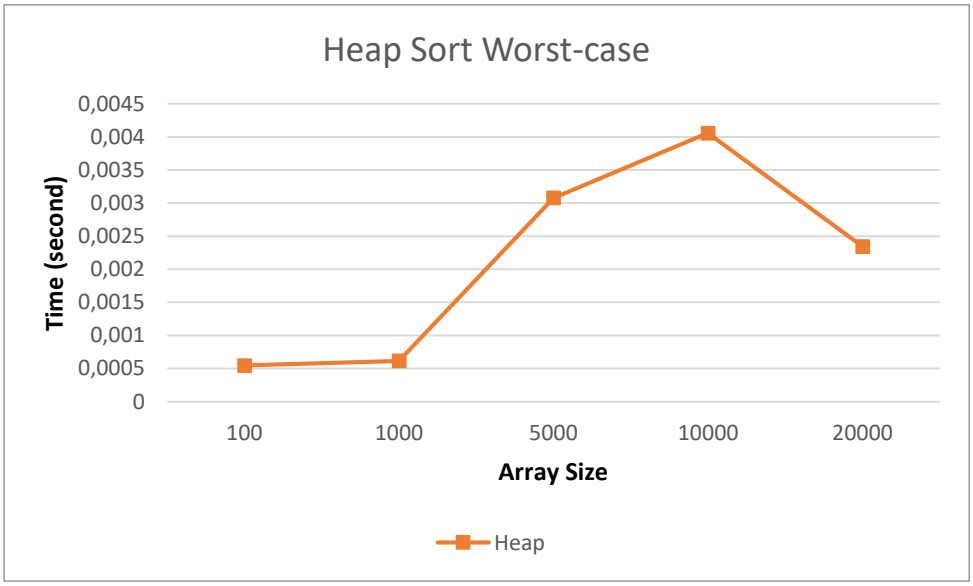
Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

3.5.1 Average Run Time Analysis



Graphic 9: *Heap Sort Average Runtime*

3.5.2 Worst-case Performance Analysis



Graphic 10: *Heap Sort Worst-case*

4 Comparison the Analysis Results

I generated 10 random array each sizes are 100, 400, 900, 1600, 2500, 3600, 4900, 6400, 8100, 10000. I run each array 10 times and average of runtimes. All average run time graphics in this document stay connected to data in Screenshot 1.

AVERAGE RUNTIME ANALYSIS

Size	100	400	900	1600	2500	3600	4900	6400	8100	10000
HEAP:	167962	267087	303982	229699	310790	567297	908591	1031059	1193360	1613091
INSERT:	381824	278227	294038	217738	307910	633413	722421	916742	1351694	1679734
MERGE:	413577	146672	179204	245976	313649	493036	702615	908241	1370074	1753414
MERGEDLL:	413553	61727	84204	45949	122228	148906	281226	401095	659107	805877
QUICK:	201346	306602	193725	204582	310381	543338	736760	935904	1231469	1766085

The unit of time is 'Nanoseconds'.

Screenshot 3: *Screenshot of Average Runtime Analysis*

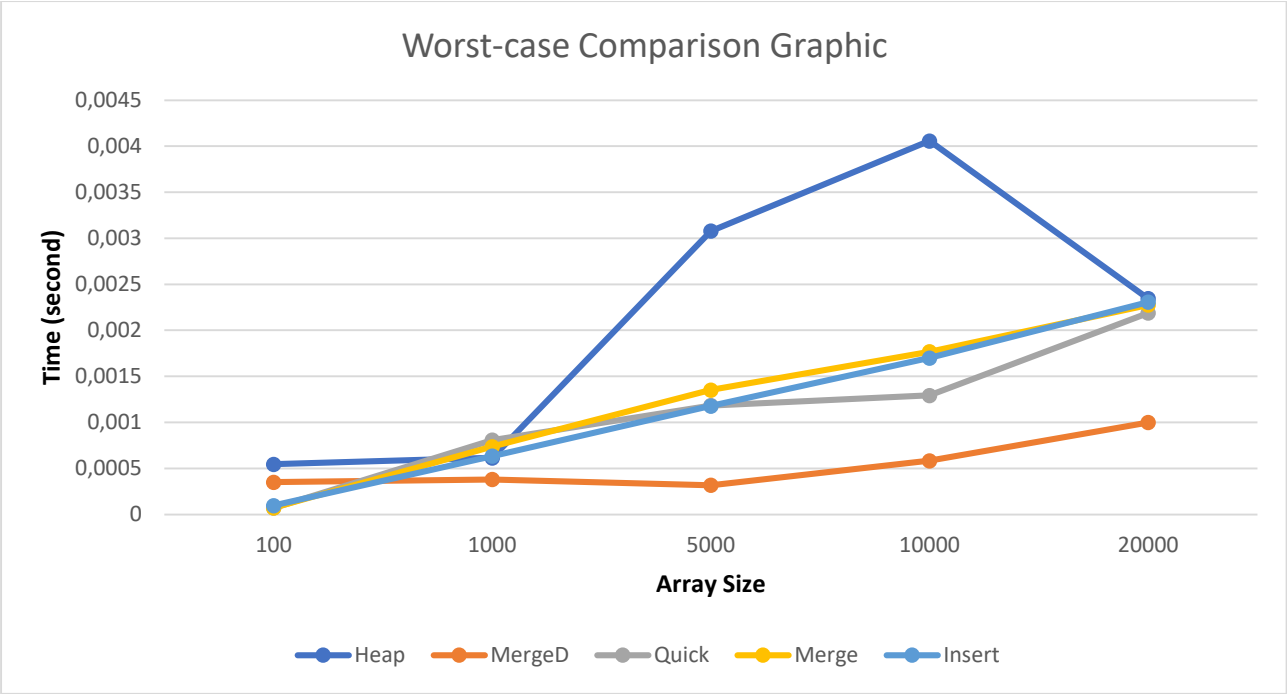
I generated 5 array and each sizes are 100, 1000, 5000, 10000, 20000. For worst case, I use reversed sorted numbers e.g., 5,4,3,2,1. This arrays is used 1 times and that result is the worst-case for the given size. All worst-case graphics in this document stay connected to data in Screenshot 2.

WORST-CASE ANALYSIS

Size	100	1000	5000	10000	20000
HEAP:	545053	615062	3078012	4058445	2344566
INSERT:	96151	634592	1177541	1699156	2307006
MERGE:	71812	736151	1358423	1767062	2276059
MERGEDLL:	350649	380394	316996	584114	998761
QUICK:	71211	808564	1184151	1292320	2189222

The unit of time is 'Nanoseconds'.

Screenshot 4: *Screenshot of Worst-case Performance Analysis*



Graphic 11: *Worst-case Comparison Graphic*