Project 1 Report
Experimental Analysis of Sorting Algorithms

CS 5120
Design and Analysis of Algorithms

Nitin Chaurasia

Academic Supervisor:
Dr. Robert Dyer

Fall 2014
Department of Computer Science
Bowling Green State University, Ohio

Project 1

Nitin Chaurasia
Supervisor: Dr. Robert Dyer
Dept. of Computer Science
Bowling Green State University, Ohio
Reporting date: Oct/20/2014

1. Introduction

The Programs are implemented to compare the theoretical runtime with that of the experimental. Presented below is the Theoretical Performance of Sorting Algorithms for each of the 4 cases.

1. **Random Data**

1.1 Insertion Sort: $\Theta(n^2)$: The first for loop will iterate n times and then for each iteration, there is another nested for loop that iterates n times. So n*n or Θ(n2).
1.2 Quick Sort: $\Theta(n \log_2 n)$: Algorithm always chooses best pivot and keeps splitting sub-arrays in half at each recursion;
T(0) = T(1) = O(1) and
T(n) = 2T(n/2) + O(n)
T(n) = O(n log n)(By using 2nd case of master theorem)
1.3 Heap Sort: $\Theta(n \log_2 n)$: It takes $O(n*\log_2 n)$ time to build a heap. Heapification is applied roughly n/2 times (to each of the internal nodes). And it takes $O(n*\log_2 n)$ time to extract each of the maximum elements, since we need to extract roughly n elements and each extraction involves a constant amount of work and one heapify. Therefore the total running time of heap sort is $\Theta(n \log_2 n)$.
1.4 Merge Sort:$\Theta(n \log_2 n)$: Every iteration splits the problem into two halves until the halving comes down to one. Thus there are $\log_2 n$ steps involved. We need $\log_2(n)$ iterations, and each iteration takes exactly O(n) (each iteration is on all sub lists, total size is still n), so total $\Theta(n \log_2 n)$
1.5 3 Way merge Sort: $\Theta(n \log_3 n)$: At each iteration the array is split into three sub lists, and recursively calls the method. At best case the split it exactly in the factors of 3, and thus the reduction of the original problem(in each recursive call) . We need $\log_3(n)$ iterations, and each iteration takes exactly O(n) (each iteration is on all sub lists, total size is still n), so at total $\Theta(n \log_3 n)$.

2. **Sorted Data and Identical Data**

2.1 Insertion Sort: $\Theta(n)$: Same as above. The complexity is of order of theta(n).

2.2 Quick Sort: $\Theta(n^2)$: Because the pivot is always the last element and also quick sort would produce one region with n-1 elements and one with only 1 element at every recursion step. The recurrence for this running time would be (partitioning costs O(n) time):
$T(n) = T(n-1) + O(n) \Rightarrow \Theta(n^2)$(by using mathematical induction).
2.3 Heap Sort: $\Theta(n \log_2 n)$: The total running time of heap sort is $\Theta(n \log_2 n)$.
2.4 Merge Sort:$\Theta(n \log_2 n)$: Every iteration splits the problem into two halves until the halving comes down to one. Thus there are $\log_2 n$ steps involved. We need $\log_2(n)$ iterations, and each iteration takes exactly O(n) (each iteration is on all sub lists, total size is still n), so total $\Theta(n \log_2 n)$
2.5 3 Way merge Sort: $\Theta(n \log_3 n)$: At each iteration the array is split into three sub lists, and recursively calls the method. At best case the split it exactly in the factors of 3, and thus the reduction of the original problem(in each recursive call) . We need $\log_3(n)$ iterations, and each iteration takes exactly O(n) (each iteration is on all sub lists, total size is still n), so at total $\Theta(n \log_3 n)$.

3. Reverse Sorted Data

3.1 Insertion Sort: $\Theta(n^2)$: because the first for loop will iterate n times and then for each iteration, there is another nested for loop that iterated n times. So essentially n*n or $\Theta(n^2)$.
3.2 Quick Sort: $\Theta(n^2)$: Because the pivot is always the last element and also quick sort would produce one region with n-1 elements and one with only 1 element at every recursion step. The recurrence for this running time would be (partitioning costs O(n) time):
$T(n) = T(n-1) + O(n) \Rightarrow \Theta(n^2)$(by using mathematical induction).
3.3 Heap Sort: $\Theta(n \log_2 n)$: because (1) that it takes $O(n*\log_2 n)$ time to build a heap, because we need to apply heapify roughly n/2 times (to each of the internal nodes), and (2) that it takes $O(n*\log_2 n)$ time to extract each of the maximum elements, since we need to extract roughly n elements and each extraction involves a constant amount of work and one heapify. Therefore the total running time of heap sort is $\Theta(n \log_2 n)$.
3.4 Merge Sort:$\Theta(n \log_2 n)$: Every iteration splits the problem into two halves until the halving comes down to one. Thus there are $\log_2 n$ steps involved. We need $\log_2(n)$ iterations, and each iteration takes exactly O(n) (each iteration is on all sub lists, total size is still n), so total $\Theta(n \log_2 n)$
3.5 Merge3 Sort: 3 Way merge Sort: $\Theta(n \log_3 n)$: At each iteration the array is split into three sub lists, and recursively calls the method. At best case the split it exactly in the factors of 3, and thus the reduction of the original problem(in each recursive call) .

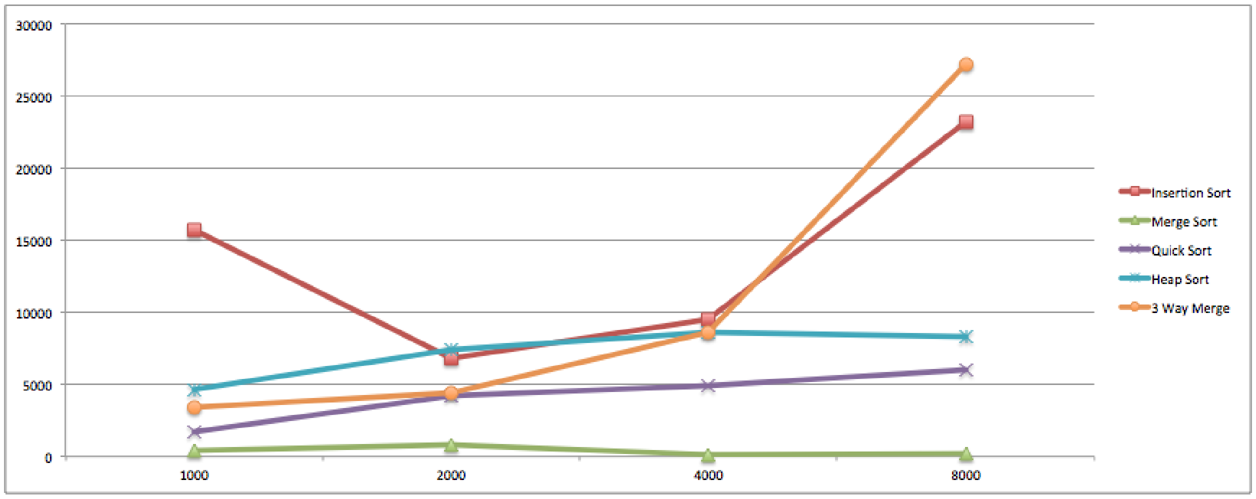
We need $\log_3(n)$ iterations, and each iteration takes exactly O(n) (each iteration is on all sub lists, total size is still n), so at total $\Theta(n \log_3 n)$.

Practical Performances(Θ) of Sorting Algorithms for each case(Time(in micro seconds)):

Execution Time with Random data

Random - Execution Times					
Length of Input	**Insertion Sort**	**Merge Sort**	**Quick Sort**	**Heap Sort**	**3 Way Merge**
1000	**15714**	**386**	**1693**	**4651**	**3413**
2000	**6775**	**812**	**4176**	**7384**	**4406**
4000	**9515**	**133**	**4935**	**8622**	**8593**
8000	**23187**	**200**	**6021**	**8326**	**27238**

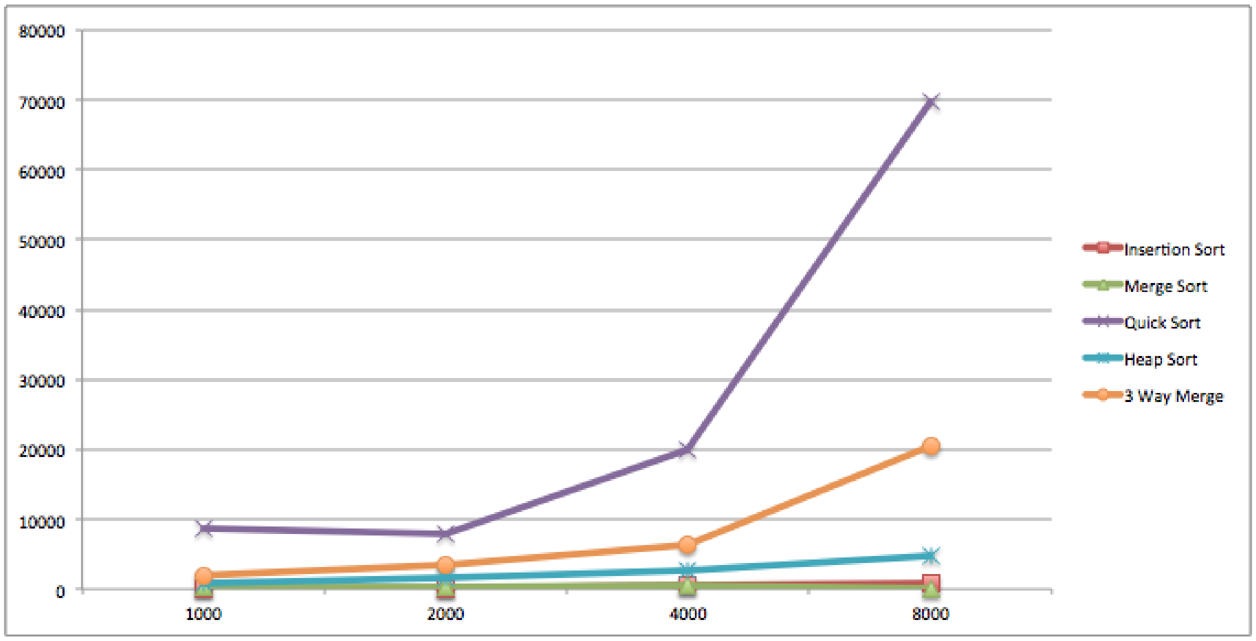
Graph based on the above table



Execution Time with Identical data

Identical - Execution Times					
Length of Input	**Insertion Sort**	**Merge Sort**	**Quick Sort**	**Heap Sort**	**3 Way Merge**
1000	**148**	**205**	**8703**	**701**	**1981**
2000	**163**	**421**	**7889**	**1693**	**3544**
4000	**445**	**543**	**20061**	**2759**	**6377**
8000	**794**	**94**	**69825**	**4700**	**20558**

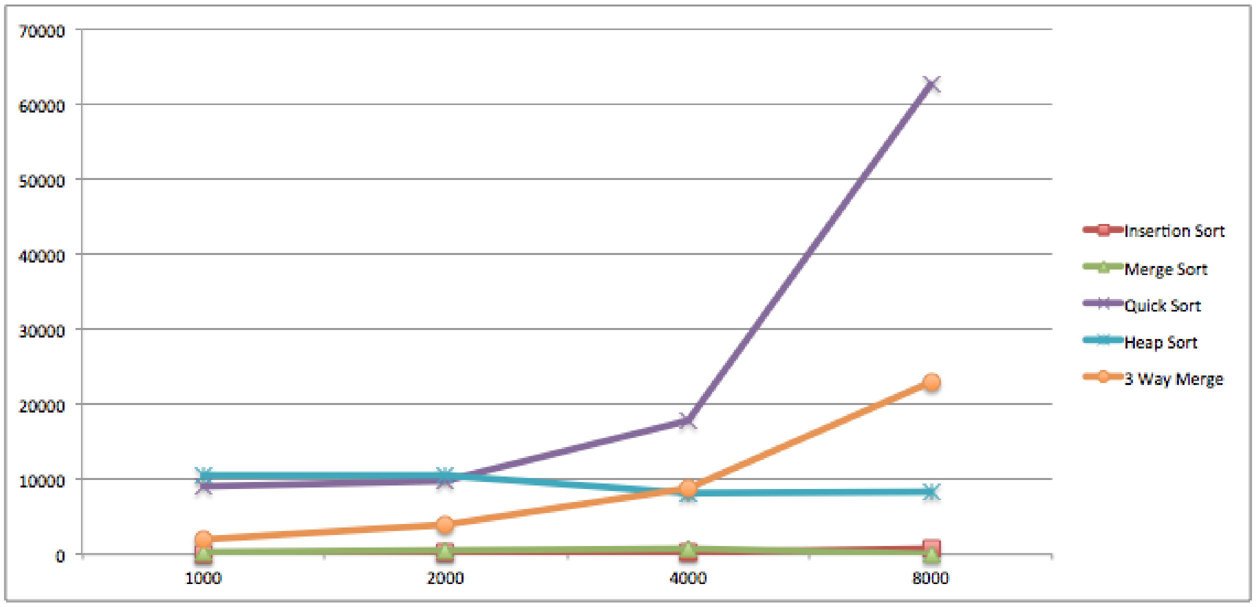
Graph based on the above table



Execution Time with Sorted data

Sorted - Execution Times					
Length of Input	**Insertion Sort**	**Merge Sort**	**Quick Sort**	**Heap Sort**	**3 Way Merge**
1000	**91**	**309**	**8970**	**10398**	**1972**
2000	**292**	**552**	**9755**	**10454**	**3881**
4000	**384**	**867**	**17930**	**8164**	**8747**
8000	**719**	**65**	**62681**	**8412**	**22939**

Graph based on the above table



Execution Time with Reversed Sorted data

Reversed Sorted - Execution Times					
Length of Input	**Insertion Sort**	**Merge Sort**	**Quick Sort**	**Heap Sort**	**3 Way Merge**
1000	**9380**	**291**	**7356**	**4615**	**2087**
2000	**8162**	**597**	**8850**	**9857**	**3718**
4000	**12898**	**54**	**24122**	**7468**	**9131**
8000	**35884**	**93**	**53378**	**8913**	**23337**

Graph based on the above table

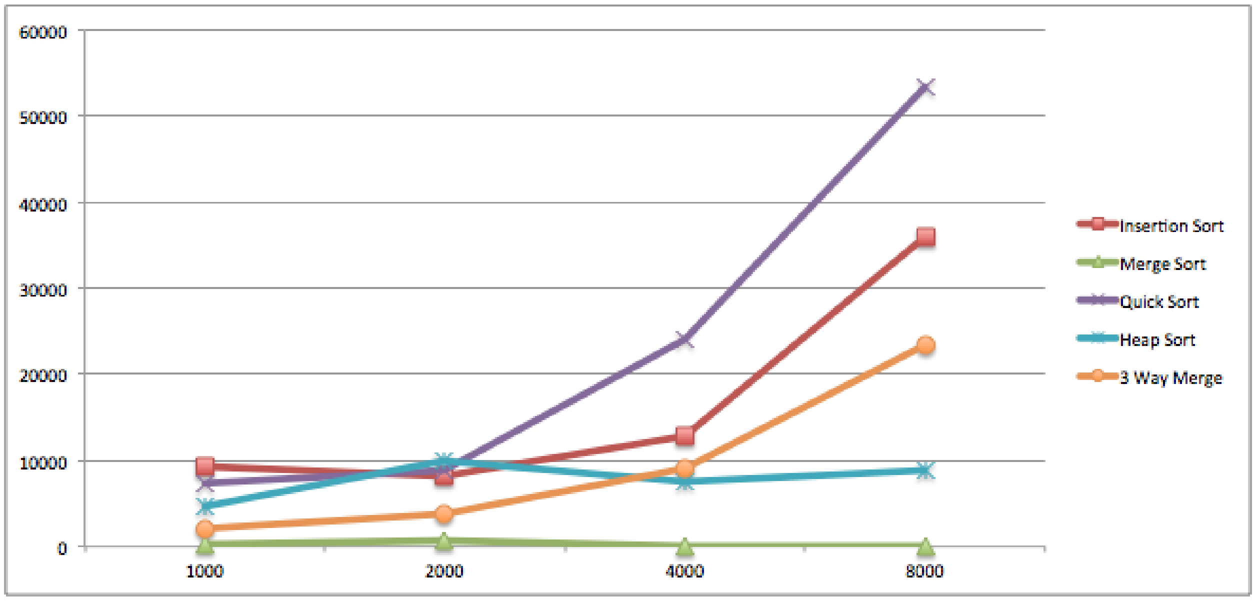


Table of Sorting Algorithms for each case(Number of Comparisons):

Presented below is the table giving the total number of comparisons

1. Total number of comparisons with Random Data

Random - Comparisions					
Length of Input	**Insertion Sort**	**Merge Sort**	**Quick Sort**	**Heap Sort**	**3 Way Merge**
1000	248667	31442	10770	18822	17321
2000	971960	68710	23545	41701	38519
4000	4008554	149642	50531	91474	85699
8000	16153993	323128	119035	198744	187835

2. Total number of comparisons with Identical Data

Identical - Comparisions					
Length of Input	**Insertion Sort**	**Merge Sort**	**Quick Sort**	**Heap Sort**	**3 Way Merge**
1000	999	24060	499500	2997	6345
2000	1999	52124	1999000	5997	13486
4000	3999	112252	7998000	11997	29619
8000	7999	240508	31996000	23997	65521

3. Total number of comparisons with Sorted Data

Sorted - Comparisions					
Length of Input	**Insertion Sort**	**Merge Sort**	**Quick Sort**	**Heap Sort**	**3 Way Merge**
1000	**999**	**24060**	**499500**	**19636**	**6345**
2000	**1999**	**52124**	**1999000**	**43276**	**13486**
4000	**3999**	**112252**	**7998000**	**94706**	**29619**
8000	**7999**	**240508**	**31996000**	**205453**	**65521**

4. Total number of comparisons with Reversed Sorted Data

Reversed Sorted - Comparisions					
Length of Input	**Insertion Sort**	**Merge Sort**	**Quick Sort**	**Heap Sort**	**3 Way Merge**
1000	**499500**	**23836**	**499500**	**17953**	**12084**
2000	**1999000**	**51678**	**1999000**	**39924**	**26513**
4000	**7998000**	**111362**	**7983406**	**87993**	**59858**
8000	**31996000**	**238760**	**31852038**	**192368**	**126042**

Discussion of Results:

Practical Performance and Theoretical Performance results are consistent.

Insertion Sort:

1.) For Random Data: It's theoretical performance(Worst Case) is $\Theta(n^2)$ and in the practical performance also we are getting a n^2 graph. This was the expectation also. At the time of testing, the run time for the random data file with 1000 entries took more time than that of 2000 for the reasons unknown.

2.) For Sorted Data and Identical data: As It's theoretical performance (Best Case) is $\Theta(n)$ and in the practical performance also we have a straight line graph.

3.) For Reverse Sorted Data: It's theoretical performance(worst case) is $\Theta(n^2)$ and in the practical performance also we are getting a n^2 graph.

Quick Sort/Merge Sort:

1.) For Random Data: It's theoretical performance(Best Case) is $\Theta(n \log_2 n)$ and in the practical performance also we are getting a n $\log_2$ n graph. As in graph, it is showing a constant line but in reality quick sort graph comes in this way in the best case. This was the expectation.

2.) For Sorted Data: It's theoretical performance(Worst Case) is $\Theta(n^2)$ and in the practical performance also we are getting a n^2 graph.

3.) For Reverse Sorted Data: It's theoretical performance(Worst Case) is $\Theta(n^2)$ and in the practical performance also we are getting a n^2 graph. This was the expectation.

Heap Sort and 3 Way Merge Sort:

1.) For Random Data: It's theoretical performance is Θ(n $\log_2$ n) and in the practical performance also we are getting a n $\log_2$ n graph..

2.) For Sorted Data: It's theoretical performance is Θ(n $\log_2$ n) and in the practical performance also we are getting a n $\log_2$ n graph. This was the expectation also.

3.) For Reverse Sorted Data: It's theoretical performance is Θ(n $\log_2$ n) and in the practical performance also we are getting a n $\log_2$ n graph..

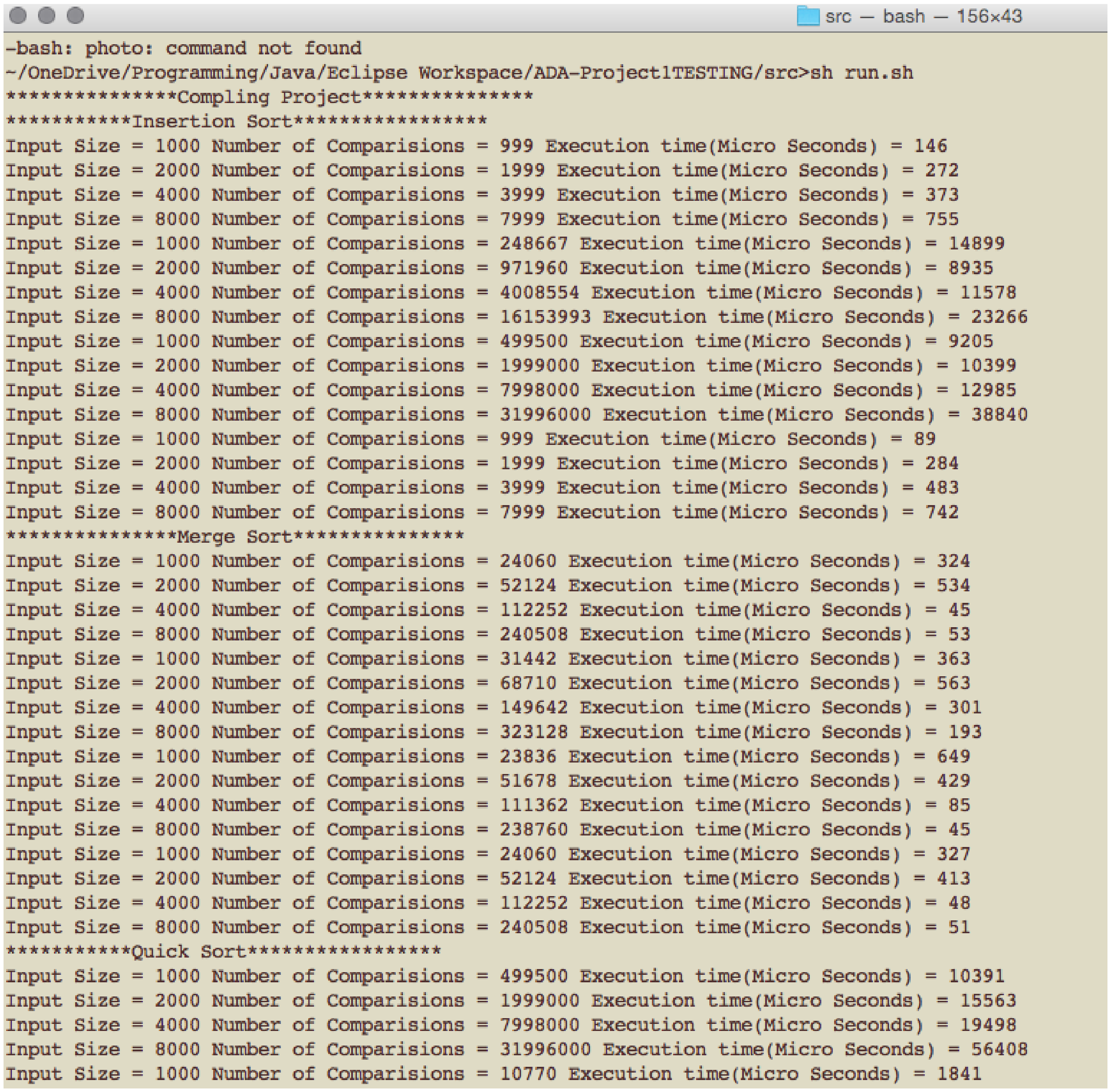
As we know that, The merge and heap sort complexity remains same for the best case and worst case means we can give any type of input whether random data, sorted data or reverse sorted data there will be no effect in complexity and same in case of drawing the graph(i.e. practical performance). In case of merge sort, the best or worst case time complexity remains same even if we are dividing the array into three sub-arrays or if we are dividing into k-sub arrays.

Analysis of Practical results:

- Insertion Sort is suitable for small files, but again it is an O(n^2) algorithm, but with a small constant. Also note that it works best when the data is already almost sorted as can be seen from graph. This algorithm is very good when the file/numbers are already sorted in increasing order.
- Quick Sort results in surprising results. It is an O(n*$\log_2$(n)) algorithm on an average and best case and an O(n^2) algorithm in the worst case. (Quick sort's worst case occurs when the numbers are already sorted!!) The graph speaks it all. This algorithm is very good when the data are not sorted in any order(i.e. random data).
- Heap Sort takes O(n log n) in average and worst case, no extra space. On random data somewhat slower than Quick Sort and Merge Sort. If you only need the first few items in a sorted list, it can be better since the initial "heapify" can be done in time O(n) as can be seen from graph. This sorting algorithm is very good for implementing priority queues. This algorithm is very good when the data are already sorted in any order.
- 3 way merge sort and Merge Sort takes O(n log n) in average and worst case, O(n) extra space. On random data they are slower than Quick Sort (graph). Theoretically performs well on external files where all data size is huge. This algorithm performs better when already sorted (in any order).

Output Results: Partial Screen shot is presented here

Snapshot of the executing program


src — bash — 156×43

```
-bash: photo: command not found
~/OneDrive/Programming/Java/Eclipse Workspace/ADA-Project1TESTING/src>sh run.sh
***************Compling Project***************
***********Insertion Sort*****************
Input Size = 1000 Number of Comparisions = 999 Execution time(Micro Seconds) = 146
Input Size = 2000 Number of Comparisions = 1999 Execution time(Micro Seconds) = 272
Input Size = 4000 Number of Comparisions = 3999 Execution time(Micro Seconds) = 373
Input Size = 8000 Number of Comparisions = 7999 Execution time(Micro Seconds) = 755
Input Size = 1000 Number of Comparisions = 248667 Execution time(Micro Seconds) = 14899
Input Size = 2000 Number of Comparisions = 971960 Execution time(Micro Seconds) = 8935
Input Size = 4000 Number of Comparisions = 4008554 Execution time(Micro Seconds) = 11578
Input Size = 8000 Number of Comparisions = 16153993 Execution time(Micro Seconds) = 23266
Input Size = 1000 Number of Comparisions = 499500 Execution time(Micro Seconds) = 9205
Input Size = 2000 Number of Comparisions = 1999000 Execution time(Micro Seconds) = 10399
Input Size = 4000 Number of Comparisions = 7998000 Execution time(Micro Seconds) = 12985
Input Size = 8000 Number of Comparisions = 31996000 Execution time(Micro Seconds) = 38840
Input Size = 1000 Number of Comparisions = 999 Execution time(Micro Seconds) = 89
Input Size = 2000 Number of Comparisions = 1999 Execution time(Micro Seconds) = 284
Input Size = 4000 Number of Comparisions = 3999 Execution time(Micro Seconds) = 483
Input Size = 8000 Number of Comparisions = 7999 Execution time(Micro Seconds) = 742
***************Merge Sort***************
Input Size = 1000 Number of Comparisions = 24060 Execution time(Micro Seconds) = 324
Input Size = 2000 Number of Comparisions = 52124 Execution time(Micro Seconds) = 534
Input Size = 4000 Number of Comparisions = 112252 Execution time(Micro Seconds) = 45
Input Size = 8000 Number of Comparisions = 240508 Execution time(Micro Seconds) = 53
Input Size = 1000 Number of Comparisions = 31442 Execution time(Micro Seconds) = 363
Input Size = 2000 Number of Comparisions = 68710 Execution time(Micro Seconds) = 563
Input Size = 4000 Number of Comparisions = 149642 Execution time(Micro Seconds) = 301
Input Size = 8000 Number of Comparisions = 323128 Execution time(Micro Seconds) = 193
Input Size = 1000 Number of Comparisions = 23836 Execution time(Micro Seconds) = 649
Input Size = 2000 Number of Comparisions = 51678 Execution time(Micro Seconds) = 429
Input Size = 4000 Number of Comparisions = 111362 Execution time(Micro Seconds) = 85
Input Size = 8000 Number of Comparisions = 238760 Execution time(Micro Seconds) = 45
Input Size = 1000 Number of Comparisions = 24060 Execution time(Micro Seconds) = 327
Input Size = 2000 Number of Comparisions = 52124 Execution time(Micro Seconds) = 413
Input Size = 4000 Number of Comparisions = 112252 Execution time(Micro Seconds) = 48
Input Size = 8000 Number of Comparisions = 240508 Execution time(Micro Seconds) = 51
***********Quick Sort*****************
Input Size = 1000 Number of Comparisions = 499500 Execution time(Micro Seconds) = 10391
Input Size = 2000 Number of Comparisions = 1999000 Execution time(Micro Seconds) = 15563
Input Size = 4000 Number of Comparisions = 7998000 Execution time(Micro Seconds) = 19498
Input Size = 8000 Number of Comparisions = 31996000 Execution time(Micro Seconds) = 56408
Input Size = 1000 Number of Comparisions = 10770 Execution time(Micro Seconds) = 1841
```