Yuri Castro
626003324
CSCE 221-200
Programing Assignment 4
Sorting Algorithms

Introduction

Sorting algorithms are a key component to computer science and in finding the best algorithm to work with various data types. This programming assignment implements four different kinds of sorting algorithms. These four algorithms are bubble sort, heap sort, merge sort, and quicksort. The objective of this assignment it to see how each of these sorting algorithms run in testing them with a variety of input files.

Theoretical Analysis

The running of the times of each of these sorting algorithms depend on how the program runs through the given data and sorts through it. The best and worst case of an algorithm can vary greatly too depending on the amount of data needed to be sorted. This section will go further into detail on the analysis and expected time complexity for each of the implemented sorting algorithms in this programming assignment.

Bubble Sort:

In this bubble sorting algorithm, the computer steps through the data to be sorted repeatedly until it is in order. The algorithm 'bubbles' two elements in the list and swaps them if needed to be in order. The comparison between a pair and swapping passes through the list until the list is completely in order. Although this type of sorting is very simple to understand and implement, it is very slow and impractical for large data. For the best case, the list needed to be sorted is already in order. With n elements, the running time will be $O(n)$. The worst case occurs when the list is in reverse order. This will result in a running time of $O(n*n) = O(n^2)$ which is not the best run time possible.

Heap Sort:

Heap sort is another comparison-based sorting algorithm. The algorithm mainly uses a structure of a complete binary tree. In the binary tree, heapsort takes place in placing the largest or smallest element to the root of the tree. With each removal, the binary tree is updates to maintain the property of desired type of binary tree. And in removing all of the objects from the heap, the result will be a sorted array. To build the heap with all the elements is $O(n)$. To heapify and maintain the heap property if O ($log\ n$). So, the overall time complexity of a heap sort if O ($n\ log\ n$). Since heap sort is a binary tree, the best and worst case for the heap sort algorithm is O ($n\ log\ n$). However, the downside of this algorithm is that it may not always be stable.

Merge Sort

Merge sort is a dived and conquer type of algorithm. The algorithm recursively divides the array in half until each sub array contains one element only. Then, it merges arrays into pairs and sorts them each time they merge together. This continues until they are in one list again which is sorted. The running time of merge sort is $T(n) = 2\ T(n/2) + O(n)$ for each divided sub array. Merging the two halves also takes linear time. As a recurrence algorithm, the overall performance of merge sort if O ($n\ log\ n$) for the best and worst case.

Quick Sort

Similar to merge sort, quick sort is a divide and conquer algorithm. What this algorithm does is uses a pivot in the array and partitions the array around the chosen pivot point. There are several ways to choose a pivot point. To partition the array is to run through the elements around the pivot point and to move the element to the left (if its less than the element) or to move it to the right (if it is greater than the pivot). The pivot point can (1) always be the first element, (2) always the last element, (3) a random element, or (4) the middle element. Partitions on an array take linear time. The worst case occurs when the pivot chose happen to be the greatest or the smallest element. This would happen if the last element is chosen all the time and the given array is in either decreasing or increasing order. The running time for this is $T(n) = T(0) + T(n-1) + O(n) = T(n-1) + O(n)$. Overall time of $O(n^2)$. The best case for the quick sort occurs when the pivot point chose is the middle element which results in a running time of $T(n) = 2T(n/2) + O(n)$. $O(n \log n)$.

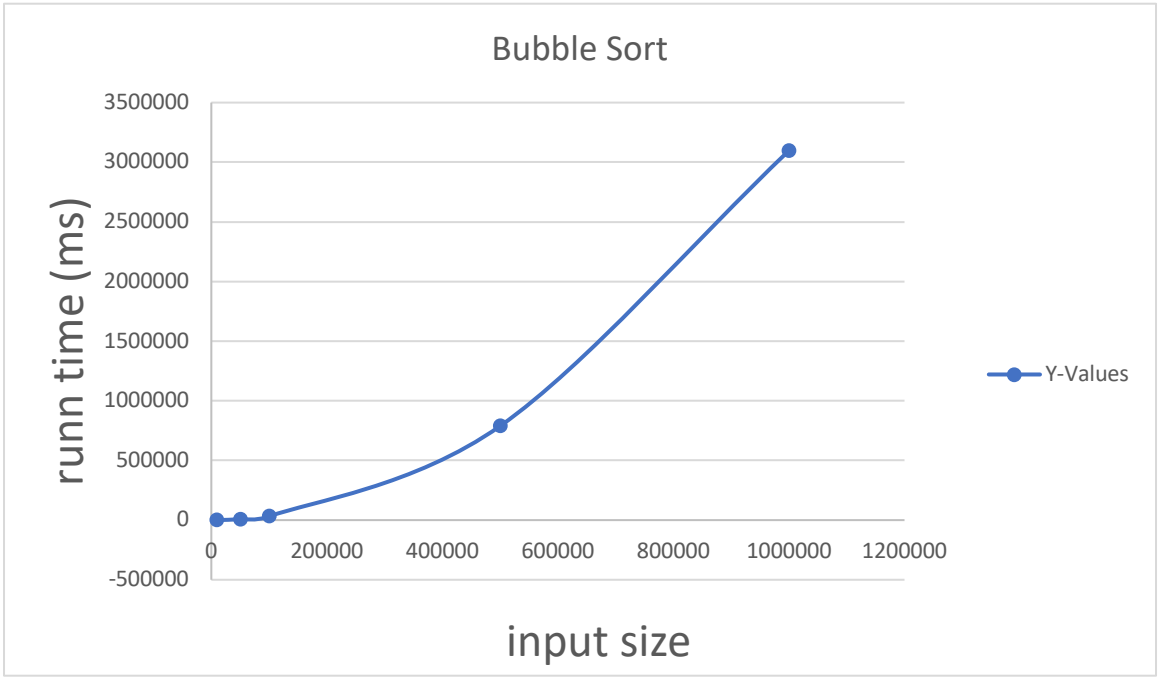
Experimental Setup

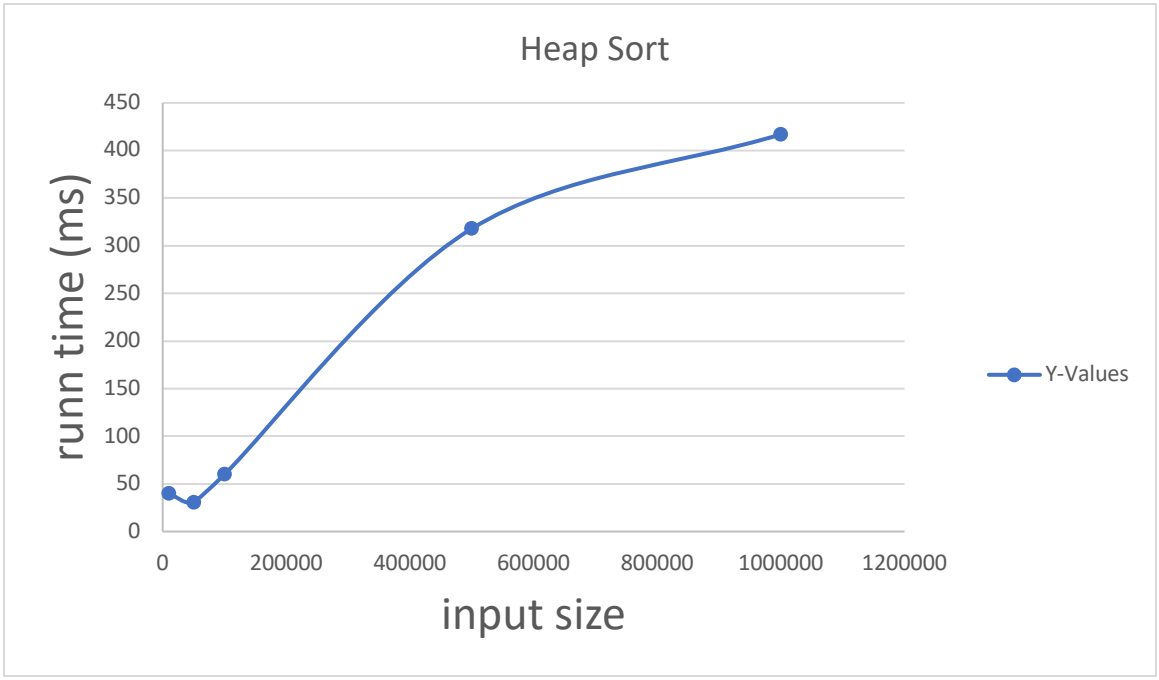
Machine Specification: macOS 2015 with 2.9 GHz Intel Core i5 processor and 8 GB 1867 MHz DDR3 memory.

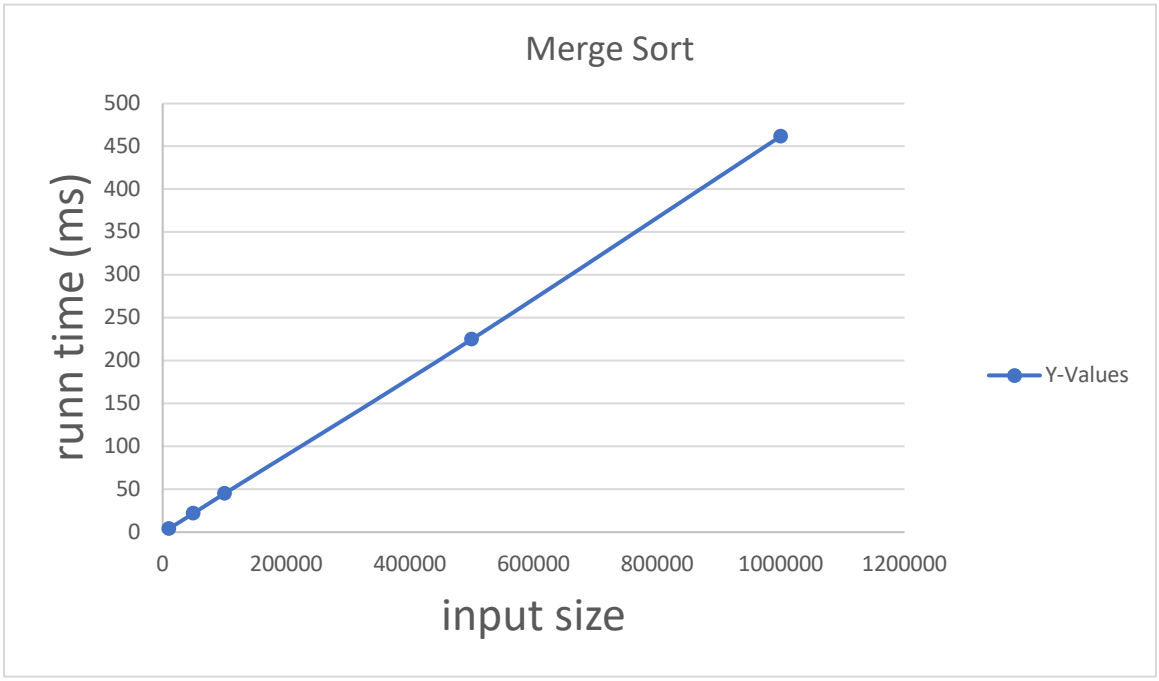
In testing the different sorting algorithms implementations, three different types of inputs were placed. I have a sorted input, reverse of the ordered, and a random ordered input. Additionally, I tested the algorithms with five different sizes of 10000, 50000, 100000, 500000, and 1000000 integers. With the different sizes of inputs, it will hopefully a good overview of how the different implementations work with large and small input sizes. All of the numbers were streamed in from a text file and saved into an array within the class data structure. Each experiment was tested at least three times.

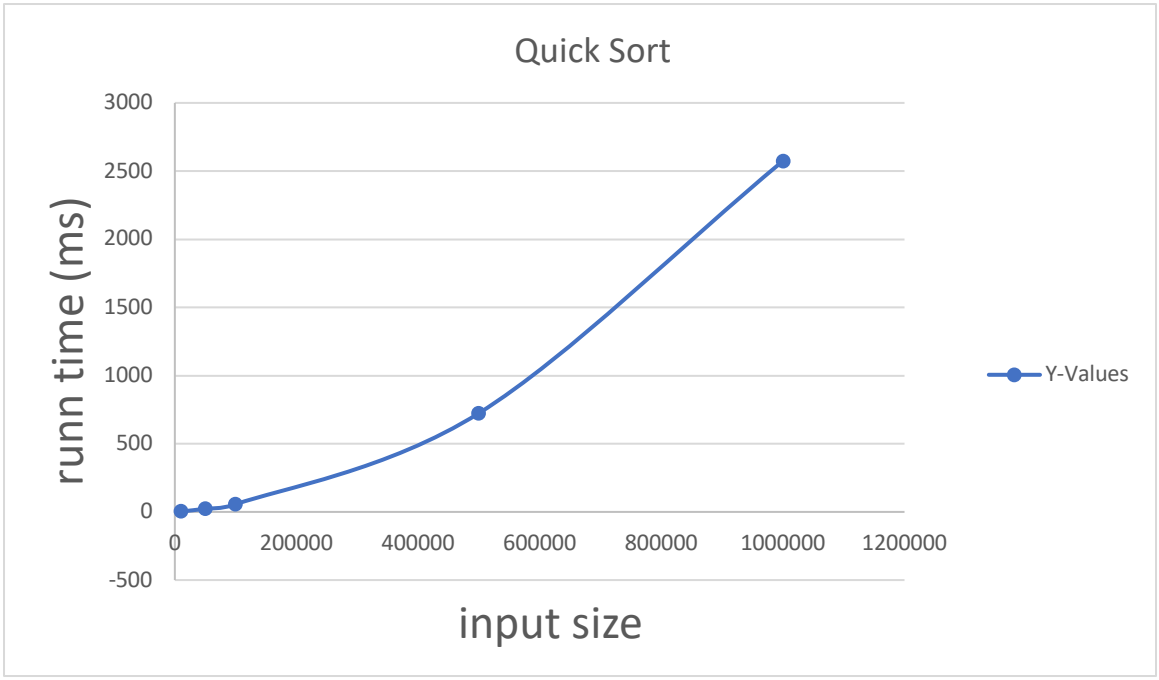
Experimental Results

RANDOM

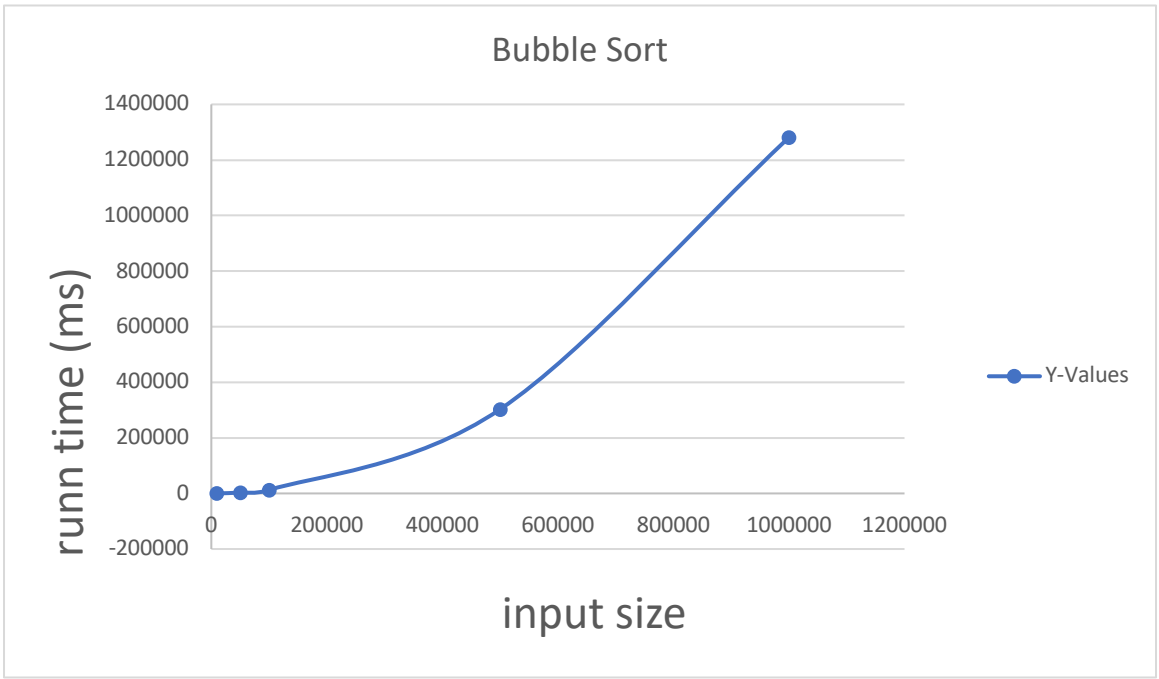


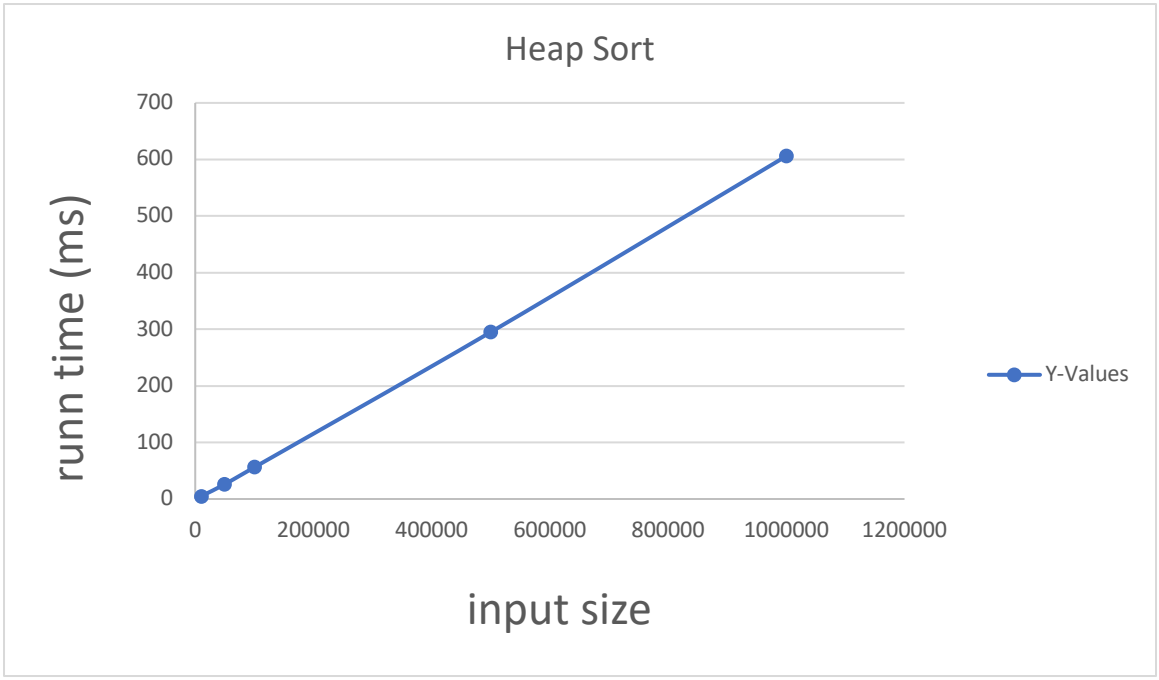

Heap Sort
runn time (ms)
450
400
350
300
250
200
150
100
50
0
0
200000
400000
600000
800000
1000000
1200000
input size
Y-Values

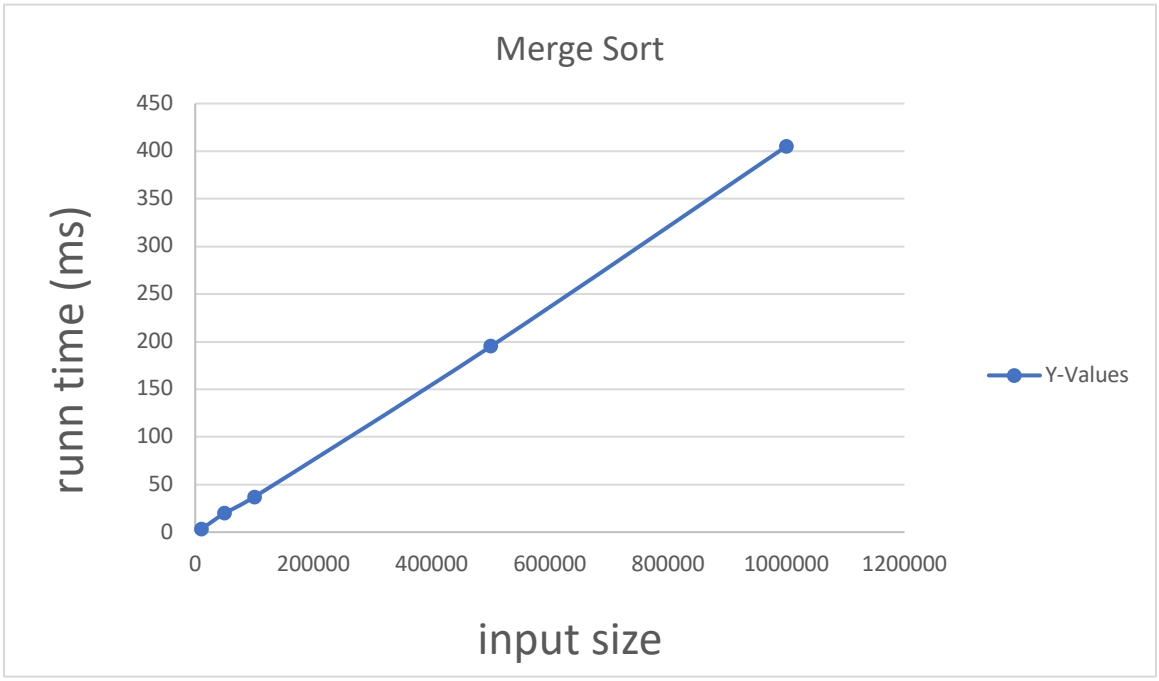

Merge Sort
runn time (ms)
500
450
400
350
300
250
200
150
100
50
0
0
200000
400000
600000
800000
1000000
1200000
input size
Y-Values

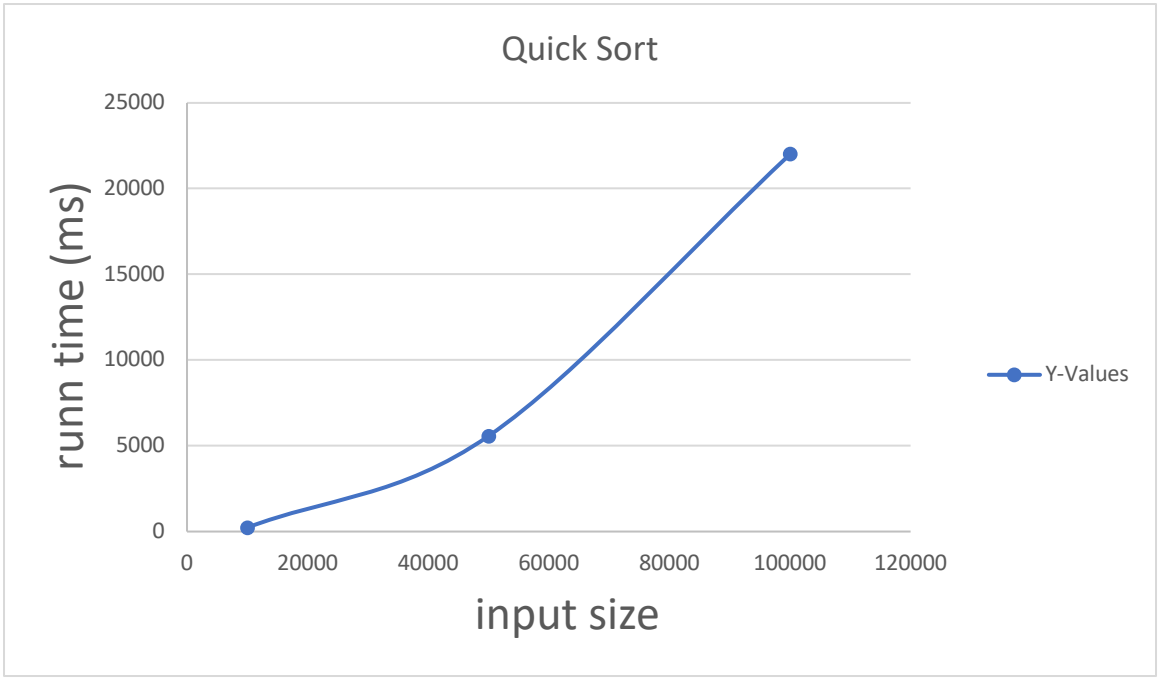

Quick Sort
runn time (ms)
3000
2500
2000
1500
1000
500
0
-500
0
200000
400000
600000
800000
1000000
1200000
input size
Y-Values

SORTED

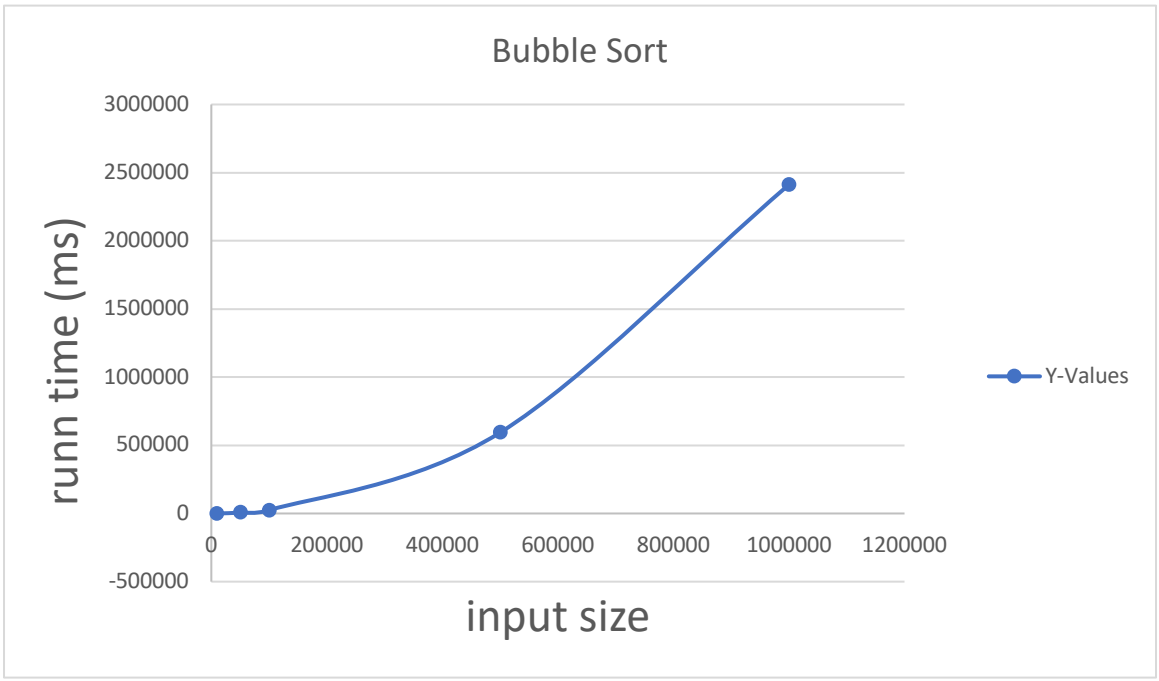

Bubble Sort
runn time (ms)
1400000
1200000
1000000
800000
600000
400000
200000
0
-200000
0
200000
400000
600000
800000
1000000
1200000
input size
Y-Values

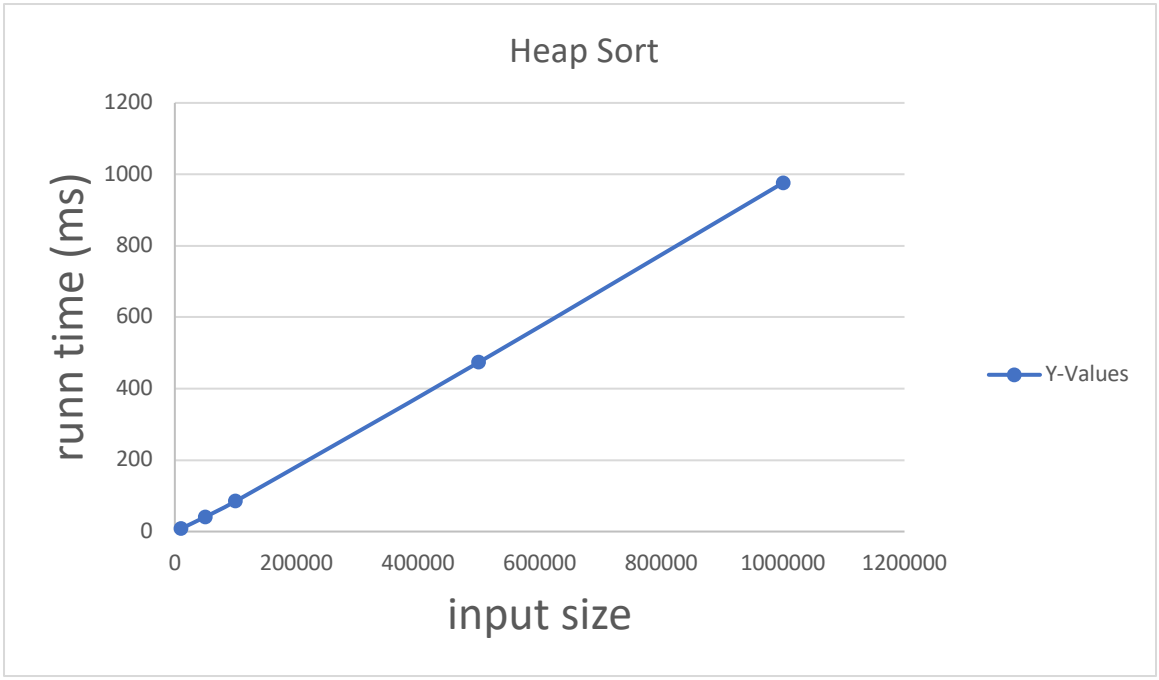

Heap Sort
runn time (ms)
700
600
500
400
300
200
100
0
0
200000
400000
600000
800000
1000000
1200000
input size
Y-Values

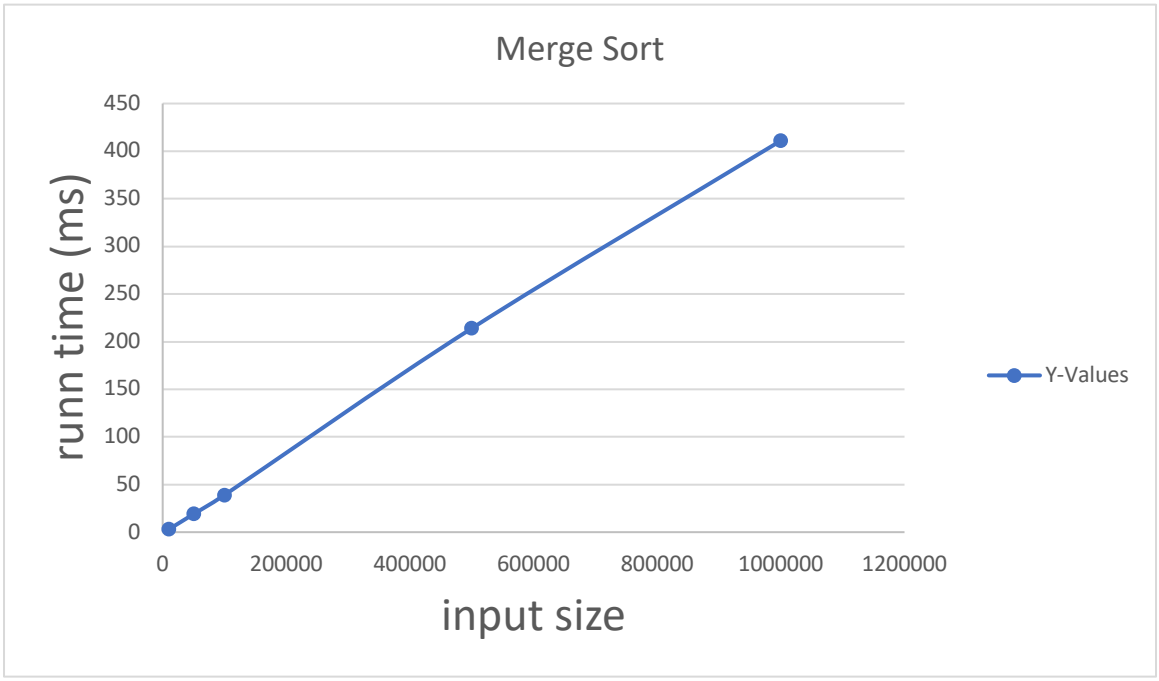

Merge Sort
runn time (ms)
450
400
350
300
250
200
150
100
50
0
0
200000
400000
600000
800000
1000000
1200000
input size
Y-Values

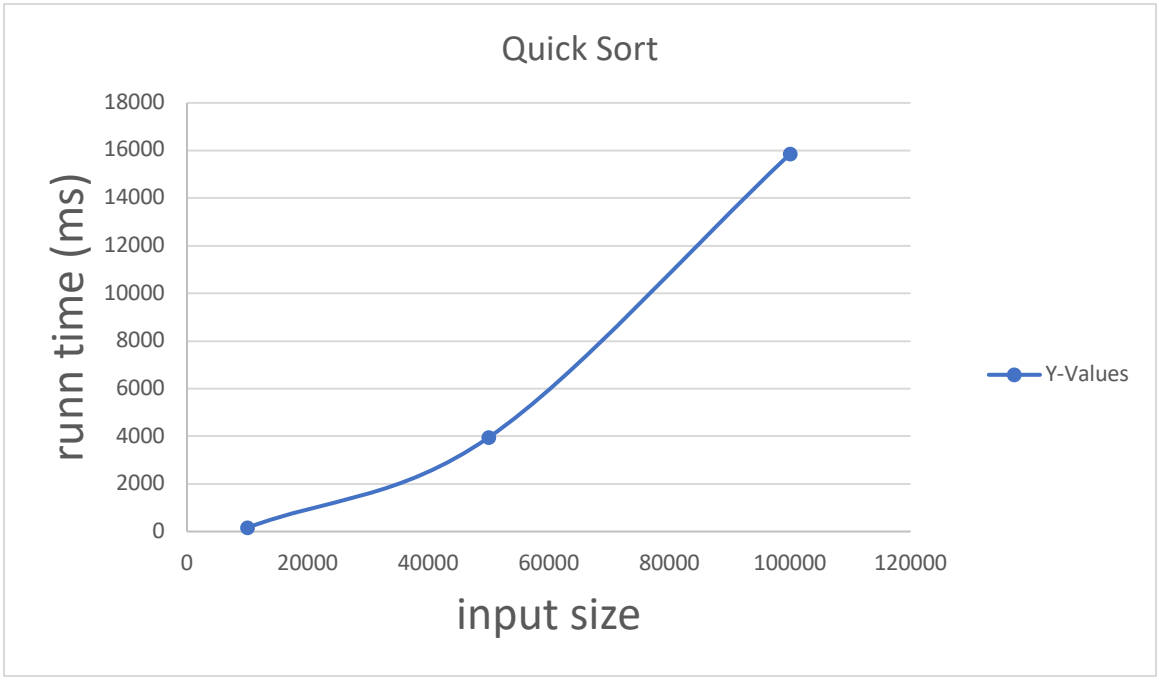

Quick Sort
25000
20000
15000
10000
5000
0
runn time (ms)
0
20000
40000
60000
80000
100000
120000
input size
Y-Values

REVERSED


Bubble Sort
3000000
2500000
2000000
1500000
1000000
500000
0
-500000
runn time (ms)
0
200000
400000
600000
800000
1000000
1200000
input size
Y-Values


Heap Sort
runn time (ms)
1200
1000
800
600
400
200
0
0
200000
400000
600000
800000
1000000
1200000
input size
Y-Values


Merge Sort
runn time (ms)
450
400
350
300
250
200
150
100
50
0
0
200000
400000
600000
800000
1000000
1200000
input size
Y-Values



As seen in the results of the graphs, most of the algorithms grow to have much larger run times with larger inputs. Quick sort shows to run the fastest and most efficient whenever the input data come in random order. But grows to be slower whenever the input is sorted or reversed. Bubble sort was the not the efficient at all whether the input be random, sorted, or reversed. Overall, heap sort seemed to work the best. It had pretty low run times compared to the others. The theoretical analysis closely lines up to the experimental results. Some discrepancies I noted was that the quick sort tended run for a really long time with the larger inputs of 500000 and 1000000 that my computer ran out of space and had to stop. Running times on the different sorting algorithms really depend on the size of the input and the order of the numbers to be sorted in the data files.