Project 1: Sorting Techniques

By: Group 4

Habib Kalia

Janet Eames

Emmett Rasmussen

Introduction

The following document accounts the multitude of iterations, approaches, and analysis of multiple sorting techniques used to sort lists of different types. Each sort uses a generated list that adheres to a certain property (In Order, Reverse Order, Almost Order, and Random Order), and is measured based on the amount of comparisons, movements, and total time it takes to successfully sort a given list. The size of each list is changeable through a sliding tool within a custom-built GUI, and the winning algorithm among the sorts is determined based on whatever sort manages to complete their list in the shortest amount of comparisons. The sizes of the lists used for data points are 200, 2000, 10000, 20000, and 30000. Along with the data presented is an excel file attached within the zip file which contains the data points referenced in the graphs.

Approach

When beginning the initial planning for the project, our team worked to split the work into manageable parts among each other, as well as utilize GitHub to share work and streamline the development process. From here, the team took apart the project step by step. It was apparent that the implementation of the project would be simple as the code for the different sorts was given. The difficulty came from creating arrays to sort that would fulfill the categories required, as well as making counters for each individual type of sort. The GUI also posed some issue. With the troubles identified, our group split the tasks among one another and completed each one in a timely manner.

Goal

The goal our group determined was to find the Winning Algorithm amongst the sorting algorithms based on the amount of comparisons done, with the algorithm using the least comparisons being the desired result. As time can vary due to the way our group implemented the comparison and movement counting, we decided to keep comparisons as the number to watch over time.

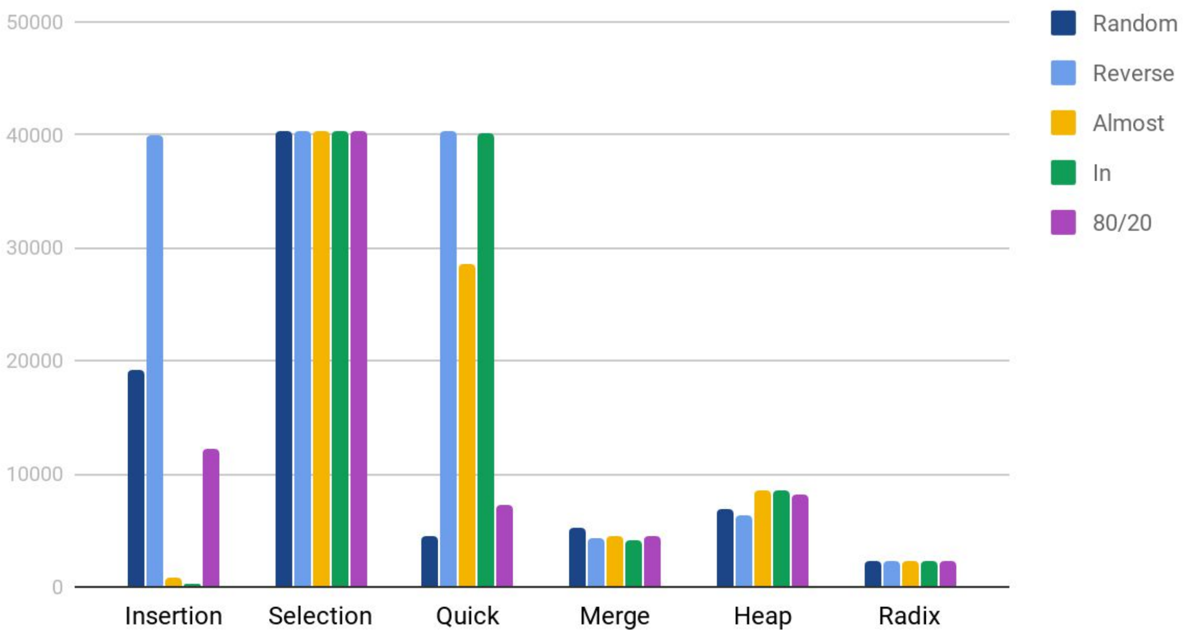
Assessment

When looking at the data presented within the graphs below at the different sizes of lists, discernable patterns begin to arise. Among different sorting algorithms, lists of reverse order often took the longest to sort at all levels with exception to those made through Radix and Selection sort, who both had consistent results among different types of lists as well as list sizes. The amount of comparisons is measured on the Y-axis, with the type of list shown in color and the sorting algorithm used being written along the X-axis.

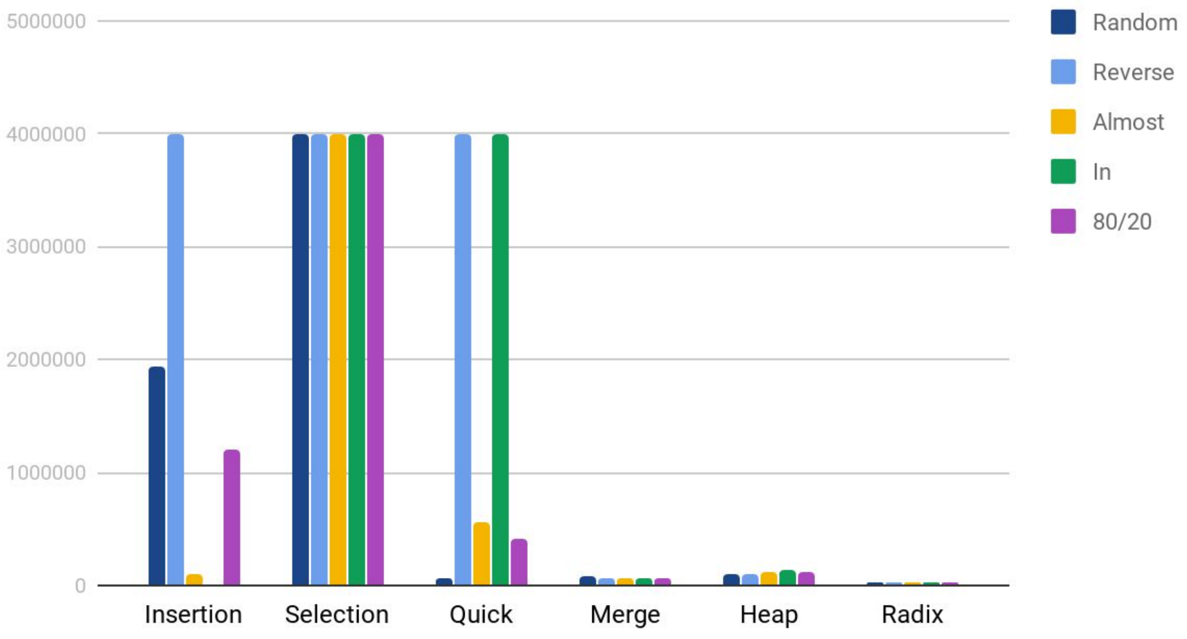
While the data present may seem strange in its display, it was found that as the size of the list grew, so did the number of comparisons, and for some sorts this grew to a very large degree. Many times the data would overlap within a graph, rendering it unreadable. The way shown below was used as it allows for the most amount of data to be clearly seen, while at the same time allowing anyone viewing to compare each graph to one another to view visible trends. This also allows one to observe which types of sort cause comparisons to grow the fastest, as sorts that grow at a rate of larger than O(n) remain seen while those which grow at rates of O(n) and less tend to disappear.

When determining which of the sorts was often the best, while discrepancy among which sort performed best at any given time tended to happen, Radix sort consistently remained very low with its amounts of comparisons even as the size of the lists grew. Selection sort performed the worst among each of the sorts, having consistently the most amount of comparisons. Heap and merge sorts both tended to be low as well, however there was often displacement among each of the types of lists used. As they both got into the upper levels of list size, they remained consistent against the amount of comparisons they used. And in the middle of each of the sorts were quick and insertion sort who both had the largest and the smallest amounts of comparisons depending on what type of list they were given.

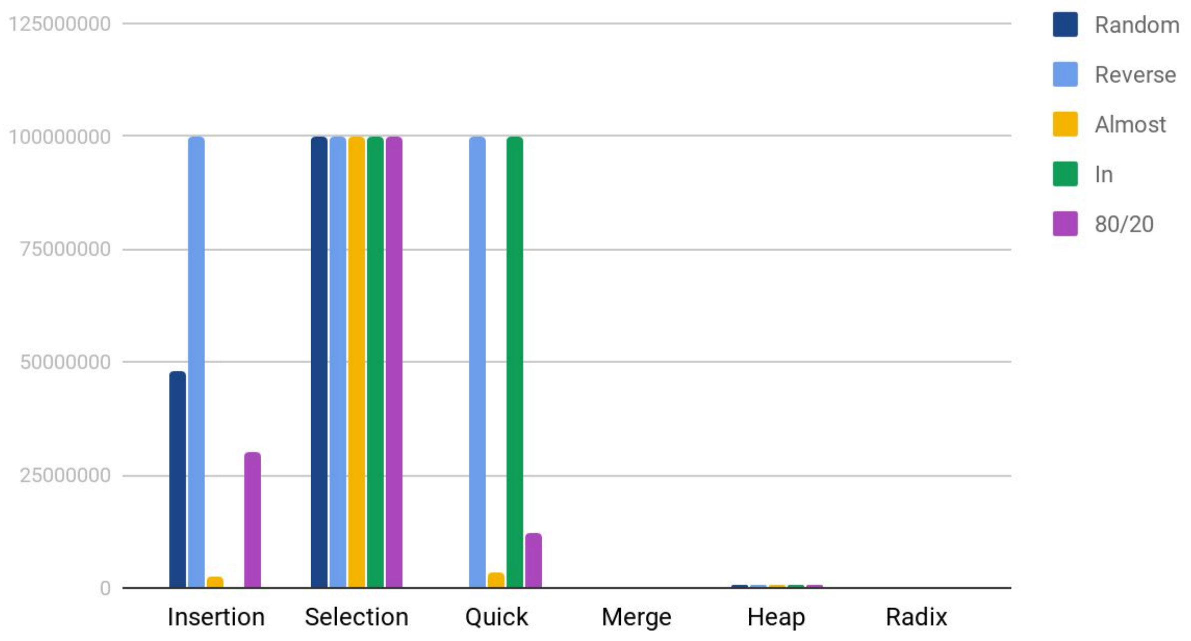
200


50000
40000
30000
20000
10000
0
Insertion
Selection
Quick
Merge
Heap
Radix
Random
Reverse
Almost
In
80/20

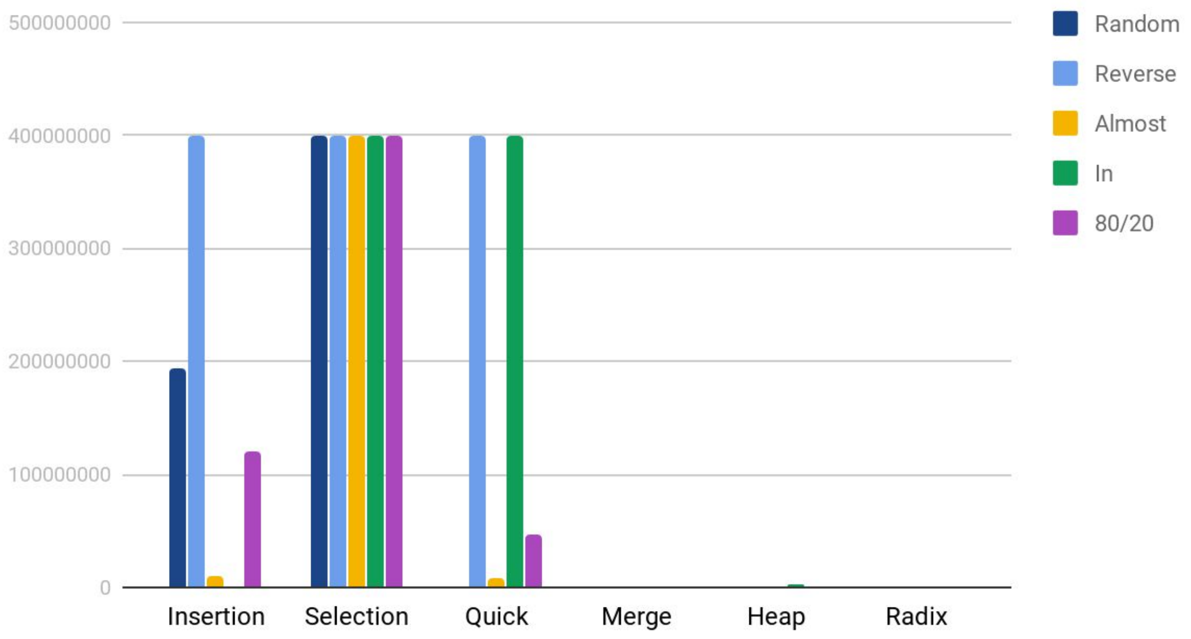
2000

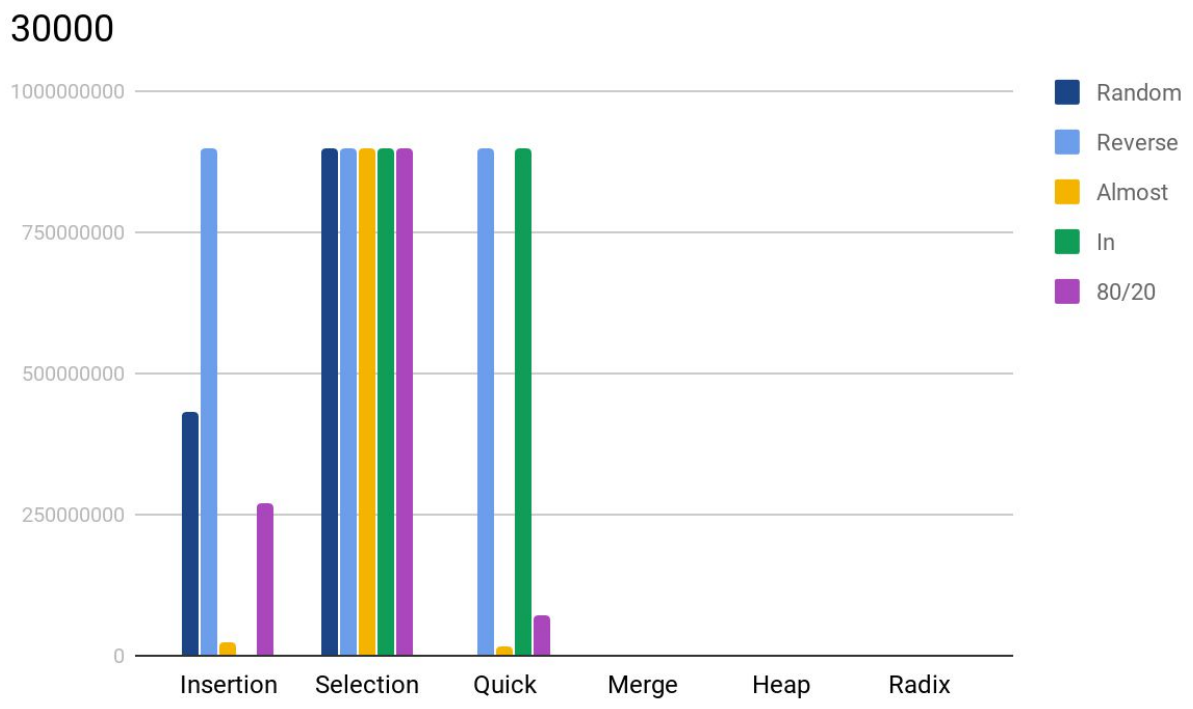

5000000
4000000
3000000
2000000
1000000
0
Insertion
Selection
Quick
Merge
Heap
Radix
Random
Reverse
Almost
In
80/20

10000


125000000
100000000
75000000
50000000
25000000
0
Insertion
Selection
Quick
Merge
Heap
Radix
Random
Reverse
Almost
In
80/20

20000


500000000
400000000
300000000
200000000
100000000
0
Insertion
Selection
Quick
Merge
Heap
Radix
Random
Reverse
Almost
In
80/20



Conclusion

Overall, the best type of sorting algorithm was Radix. If given any type of list, Radix tended to be both consistent and averaged the least amount of comparisons at all levels. Contrary to this was Selection Sort which consistently required the most amount of comparisons.