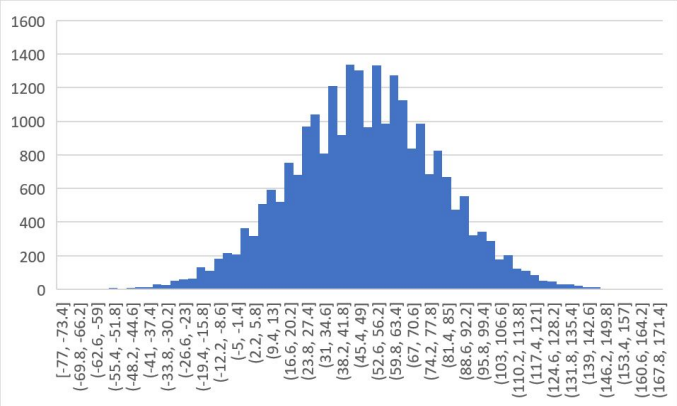
Michael Josten
Danielle Lambion
TCSS 598: Masters Seminar
Simulation Of Sorting Algorithms

a) Brief description of each sorting techniques's algorithm

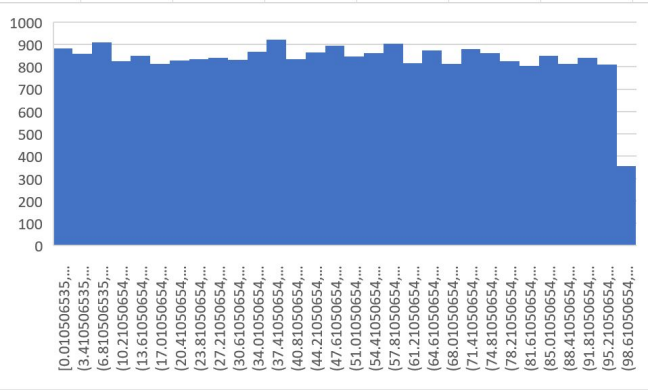
1. Bubble Sort:
   Bubble sort works by repeatedly swapping adjacent elements in the list. An element at index i will be swapped with i+1 if the element at i+1 is less than the element at i. Bubble sort will keep iterating through the list until there are no swaps to be made, then the list is sorted.
2. Selection Sort:
   Selection sort works by iterating through the list, finding the smallest element, then swapping that element with the first element in the unsorted subarray, then increasing the index of the sorted subarray by 1.
3. Insertion Sort:
   Insertion sort works by selecting the first element in the unsorted subarray, then finding the correct location for that element in the sorted subarray, and increasing the index of all elements greater than the chosen element in the sorted subarray by 1 and placing the chosen element in the space made. This will increase the sorted subarray index by 1, which we then choose the next element in the unsorted array to place in the sorted subarray.
4. Merge Sort:
   Merge sort works by recursively splitting the list in half, and then merging each half by comparing the left array element with the right array element, then choosing the smaller of the elements to be placed in the merged array, then increment the index of the pointer to the array of the element chosen by 1.
5. Quick Sort:
   Quicksort works by selecting a pivot element, then placing all elements smaller than pivot on left of pivot and all elements greater than pivot on right of pivot. Then recursively call quicksort on the elements smaller than the pivot and on the elements greater than the pivot.

b) Description of data sets
1. Random Normal Distribution Dataset:



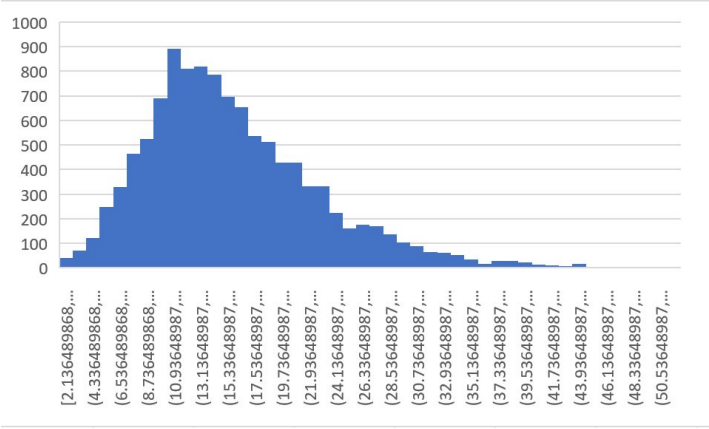
The dataset has 25,000 numbers. Each number is a random number with a mean of 50.0 and a standard deviation of 30.0. The distribution is shown above.

2. Random Uniform Distribution Dataset:



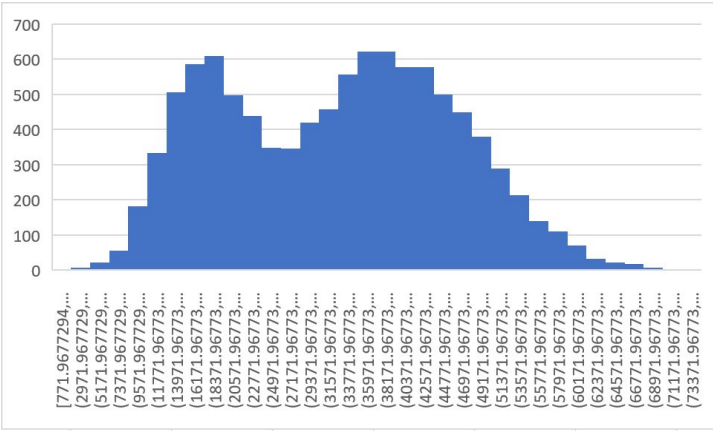
The dataset has 25,000 numbers. Each number is a random decimal number from 0 to 100. The distribution is shown above.

3. Average Hourly Income Dataset:



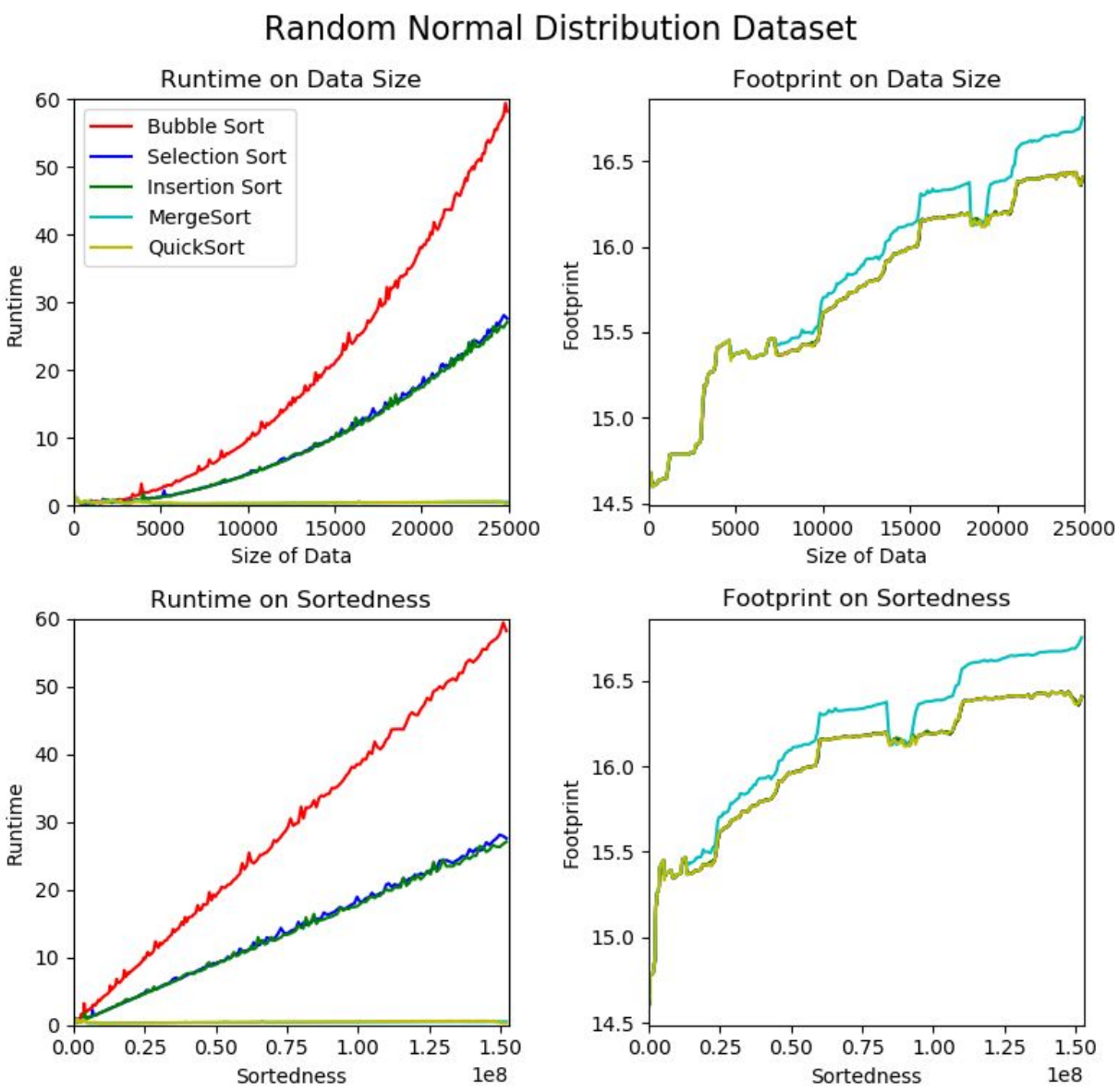
The dataset contains 11,130 numbers and has a distribution as seen above. It contains the average hourly earnings of 11,130 individuals from 1998 and contains both male's and female's earnings. Sourced from the Bureau of labor statistics, U.S. Department of Labor.

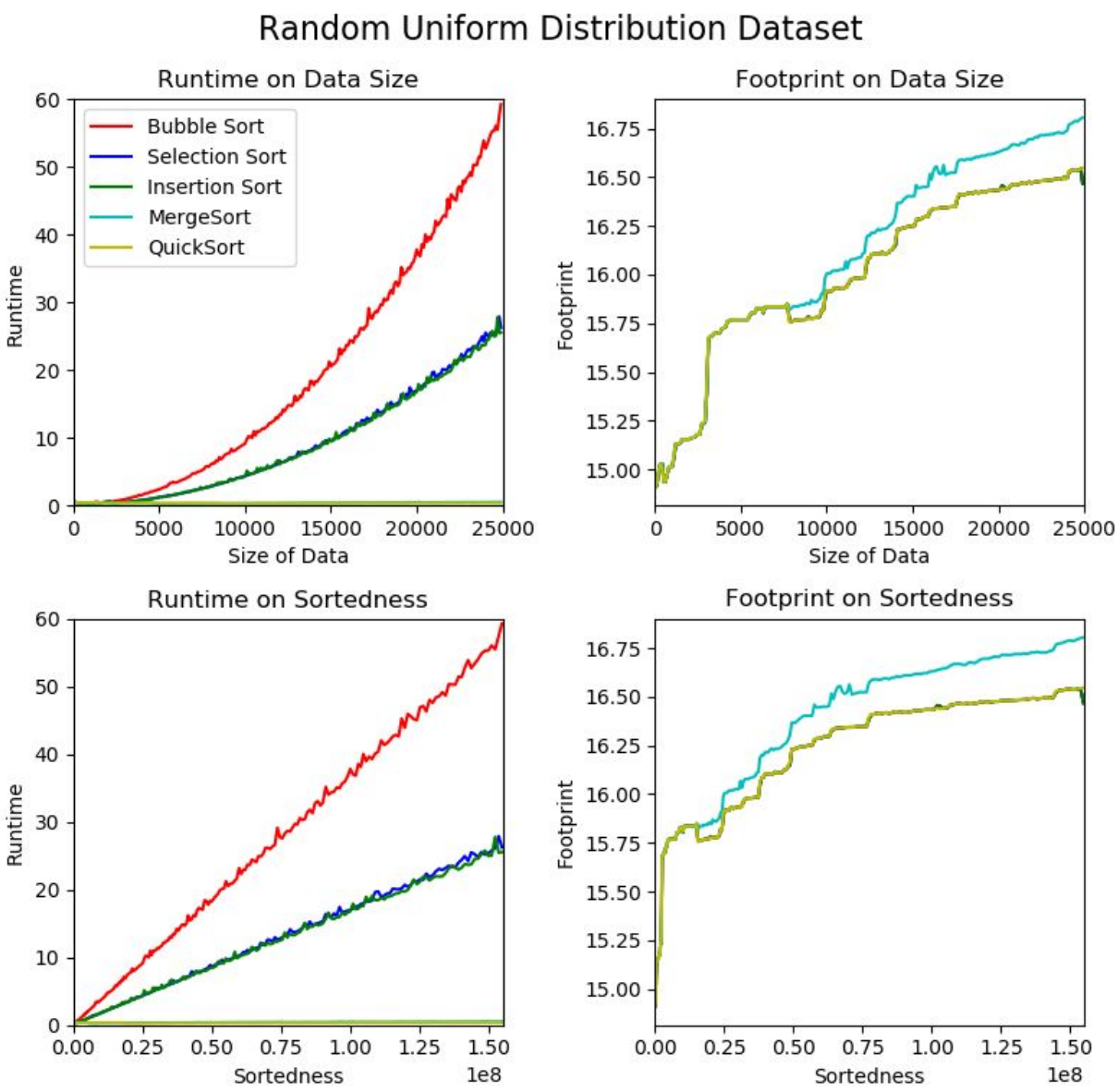
4. Individual Income When Credit Card Default Dataset:



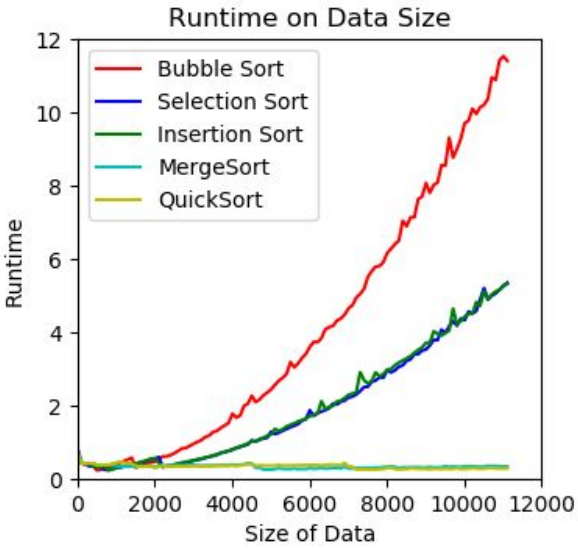
The dataset contains 10,000 numbers with a distribution curve as seen above. It is a dataset of 10 thousand customers for a credit card service which has their income and if they defaulted on a credit card payment or not. The authors goal of this dataset is to build a prediction model for which customers are most likely to default on credit card payment based on income, student status, and the balance remaining on credit card after payment. This is a simulated data set from James, G., Witten, D., Hastie, T., and Tibshirani, R. (2013) *An Introduction to Statistical Learning with applications in R*, www.StatLearning.com, Springer-Verlag, New York.

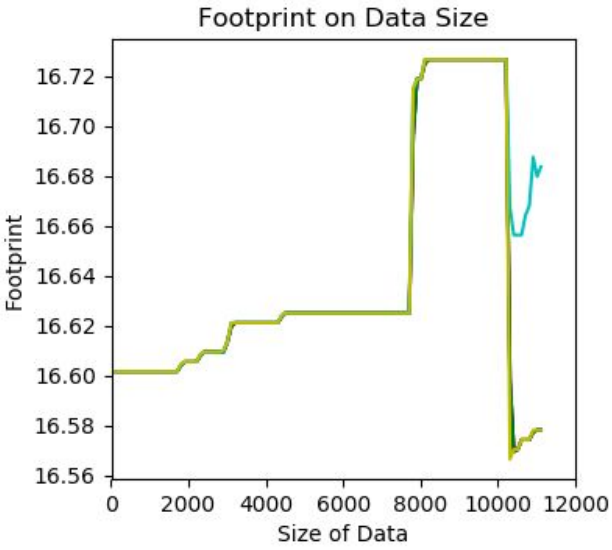
c) 16 Performance Curves

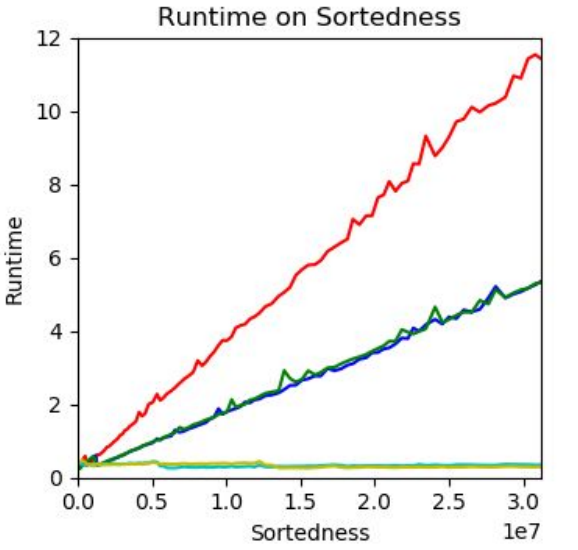

Random Normal Distribution Dataset
Runtime on Data Size
Bubble Sort
Selection Sort
Insertion Sort
MergeSort
QuickSort
Runtime
Size of Data
Footprint on Data Size
Footprint
Size of Data
Runtime on Sortedness
Runtime
Sortedness
1e8
Footprint on Sortedness
Footprint
Sortedness
1e8

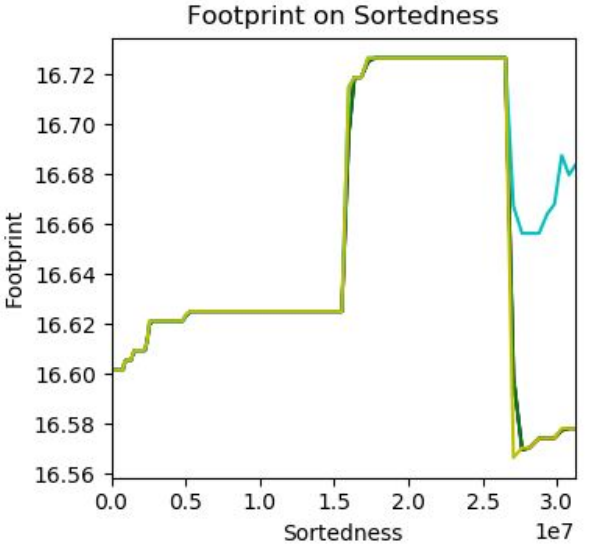

Random Uniform Distribution Dataset
Runtime on Data Size
Bubble Sort
Selection Sort
Insertion Sort
MergeSort
QuickSort
Runtime
Size of Data
Footprint on Data Size
Footprint
Size of Data
Runtime on Sortedness
Runtime
Sortedness
1e8
Footprint on Sortedness
Footprint
Sortedness
1e8

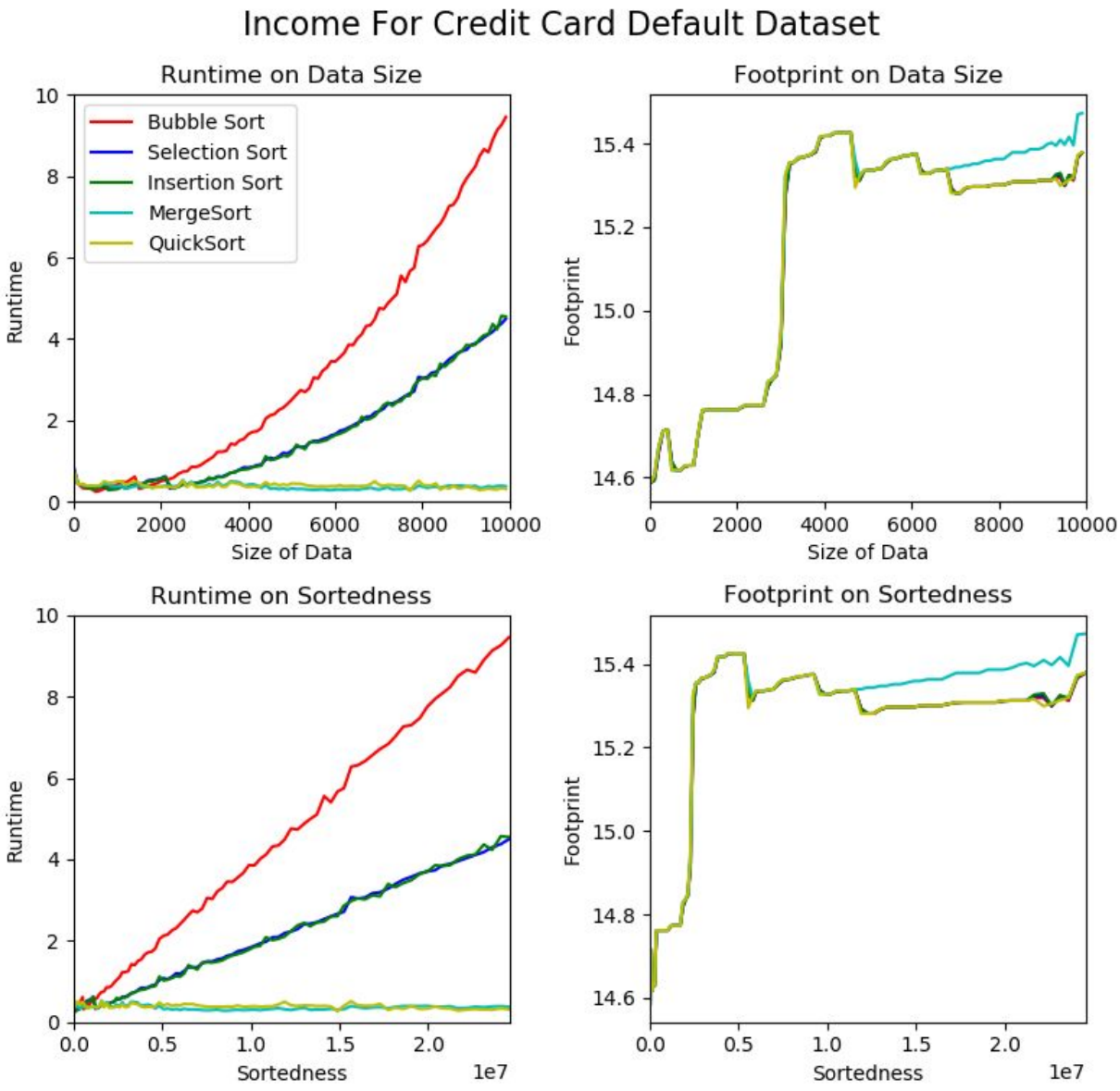
Average Hourly Earning Dataset











d) discussion on the performance curves that interprets and describes the behavior of each sorting technique, its complexity analysis and when to use each technique:

With the bubble sort algorithm, the time complexity is $O(n^2)$. The bubble sort runtime is clearly shown in each of the runtime graphs as n^2 with the curve as shown. Bubble sort is also affected by the degree of sortedness where bubble sort is at its worst case with a list in reverse order which has a high degree of sortedness. We can see that bubble sort runtime grows as the degree of sortedness increases. It is also the case that the degree of sortedness increases with the size of the data, which can contribute to the increase of the bubble sort runtime. Bubble sort does not use any extra memory space to sort the list of numbers. The extra storage complexity

of bubble sort is O(1). The footprint measure is recorded by running a python function called memory_profiler which records the memory usage of a python function. So the footprint being recorded by the memory_profiler increases as the data size increases because the array being sorted is the size of the data. There is some fluctuation in the footprint which might be some other processes that python is running which contributes to the footprint when the sorting algorithm is running. The simulation shows that bubble sort is best used when the size of the data is very low and the degree of sortedness is also low.

The selection sort algorithm has a time complexity of $O(n^2)$. The time complexity is shown in the runtime graphs, but has a lower runtime than bubble sort when the size of the data increases on each of the runtime graphs. Selection sort also has the same time complexity no matter the degree of sortedness. The reason the runtime on the selection sort algorithm increases when the degree of sortedness increases is that the degree of sortedness increases with the size of the data increase. Except for the array being sorted, selection sort uses a constant extra amount of space, so the footprint increases when the size of the data increases from the array to be sorted. The selection sort algorithm is best used when there is a moderate amount of unsorted numbers, about 2500 numbers, with a high degree of sortedness, and when extra memory is limited.

The insertion sort algorithm has a time complexity of $O(n^2)$. The runtime of insertion sort is similar to the runtime of selection sort. Insertion sort has a better runtime than bubble sort in the simulation when the size of the data increases. Insertion sort is affected by the degree of sortedness where the worst case of insertion sort is when the list is in reverse sorted order. Insertion sort uses an extra constant amount of space so the space complexity is O(1). Based on this simulation, we would recommend using selection sort when there is a moderate amount of numbers, about 2500 numbers, with a low degree of sortedness, and when extra memory is limited.

The merge sort algorithm has a time complexity of O(n logn). The runtime of merge sort is clearly shown in each runtime graph to have a better runtime than bubble, selection, and insertion sort. Merge sort has a similar runtime to quick sort. Merge sort is not affected by the degree of sortedness. Merge sort has a space complexity of O(n) so in the worst case, it can use double the amount of memory bubble, selection, or insertion sort would use. We can see this extra space requirement in the footprint graphs. Based on this simulation, we would recommend using merge sort when there is a moderate to high amount of data with a high degree of sortedness and when extra memory is abundant.

The quicksort algorithm has an average time complexity of Θ(n logn). The runtime of quicksort is shown in each runtime graph to be better than bubble, selection, and insertion sort. In some cases, quicksort has a faster runtime than merge sort, although not by much. Quicksort is affected by the degree of sortedness with a worst case runtime of $O(n^2)$ when the list is in reverse sorted order. Quicksort uses O(n logn) extra memory which is better than merge sort, but worse than bubble, selection, and insertion sort. Based on this simulation, we would recommend using quicksort when there is a moderate to high amount of data to sort with a low degree of sortedness and when extra memory is available to use.

e) The measure of sortedness is the number of inversions required to sort the list of numbers.

An issue with the degree of sortedness graphs is that we were calculating the degree of sortedness as the size of the data increased when we should have been using the same data size and then increasing the degree of sortedness which would have made graphs that better represent how each sorting algorithm is affected by the degree of sortedness.

Another issue is with how we collected the footprint statistics. We used the memory_profiler function for python which gave us some fluctuations on the memory used in a function and it was not a steady increase in memory used as the size of the data increases. Although we are able to see the difference between merge sort and the other sorting algorithms with the memory footprint size, the footprint value does not smoothly increase.