CSE 331 Data Structures and Algorithms

Sorting Algorithms Program

PROJECT DOCUMENTATION

Submitted to:

Prof. Dr. Hesham Mahomoud Ahmed Farag

Eng. Fady Faragallah

Submitted by:

Youssef George Fouad	19P9824
Mostafa Nasrat Metwally	19P4619
Kerollos Wageeh Youssef	19P3468
Anthony Amgad Fayek	19P9880

0. Introduction

The program is written in Java using the Maven JavaFX library. It is compilled and built using jdk 17. An artifact is then built using the Intellij Compliler which generates a Jar executable which was then converted into an exe file using the Launch4j tool.

The Jar file should be runnable on any machine that has a Java Runtime Enviroment of 17 or higher installed. However a few features that will be mentioned later won't work on any machine other than a Windows machine.

The exe file also requires a Java Runtime Enviroment of 17 or higher, or a bundled JRE that is placed in the folder which the exe exists.

To install Java Runtime Enviroment open Java Downloads | Oracle and download the installer for jdk 17.0.1 and install Java. Another option if you are on windows could be to install the zip and extract its content into the folder of the exe so that there is a folder named "jdk-17.0.1" next to the exe.

The project code can be found on Anthony-Amgad/SortingAlgorithmDSProject (github.com).

You'll find that the GUI Scene is written using FXML and is built using Gluon JavaFX scene builder.

The demonstration video is on OneDrive

Table of Contents

Table of Figures

1. User Guide

Figure 1: Program GUI

This is the window that you are first met with when the program opens. As we can see on the left is all the options provided for the user to choose and mix between. On the right is the generated results of the steps taken to sort as well as the sorted arrays of numbers. Finally, in the middle is the line chart which all the steps of the different sorting algorithms are then compared.

Figure 2: Selecting the algorithm(s)

The provided check boxes are to allow the user between the desired sorting algorithms to use and compute.

Figure 3: Asymptotic Function(s) button

The “Add Asymptotic Function(s)” button is a toggle button to allow the user to compare the selected algorithm(s) with their respective asymptotic notation. For example, if the user picks the insertion sort and toggles the button, both the step data of the insertion sort as well as the n^2 function will be drawn on the line chart.

Figure 4: Text Boxes

The shown text boxes are to allow the user to pick the largest possible element (integer) of the generated random file, the maximum array size to compute the sorting steps at, as well as the steps taken between the different array sizes (n) to compute the sorting steps at.

Figure 5: Line Chart

This is the line chart of which the results will appear to allow comparison.

Figure 6: Tables

Those are 2 tables shown in different tabs to view the generated results numerically aside from the line chart.

Figure 8: Credits Button

This is the credits button which open an alert when clicked to view the project team members as shown in the next figure.

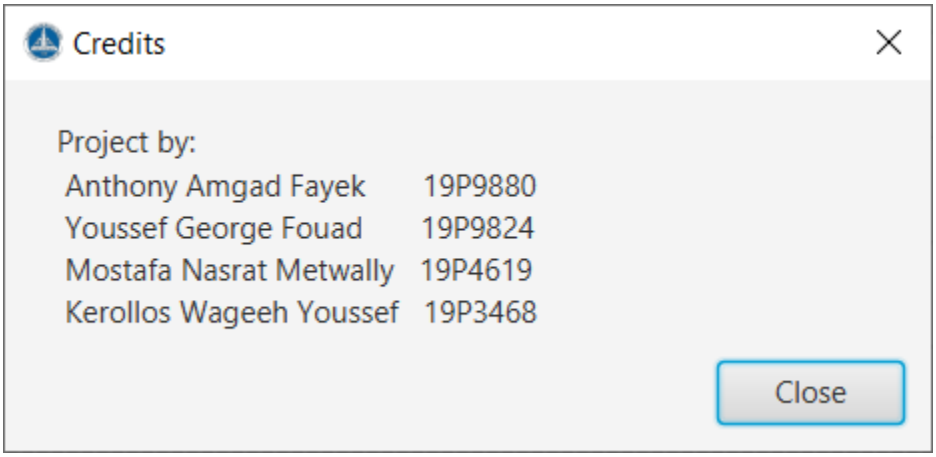

Figure 7: Credits Alert

Figure 10: Draw Button

This is the main program button that generates the random file as well as sort it while computing the steps to compare it on the line chart. It can be used in many different ways which will be explained in detail.

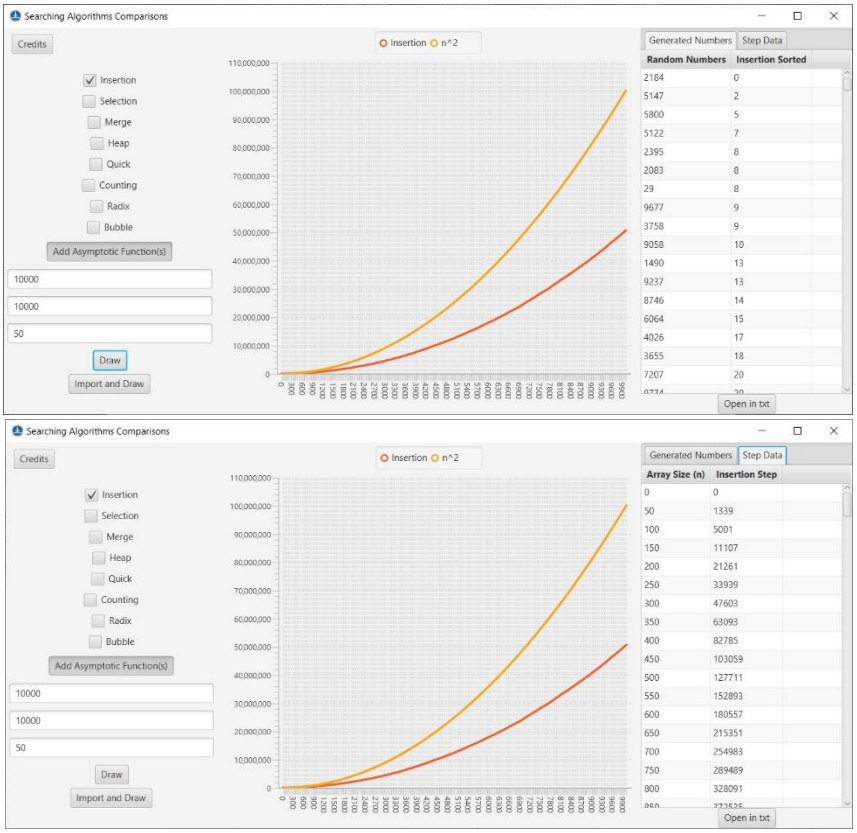

Figure 9: Insertion Sort Asymptotic Graph Demo

When pressed it shows the selected charts to appear as well as the generated data in the tables as in the previous figures.

Figure 11: Open in txt Button

When the “Open in txt” button shwon is pressed it views the numbers of the currently viewed table in separate text files on notepad; one for each algorithm. These text files can be found in the folder where the exe/jar is located.

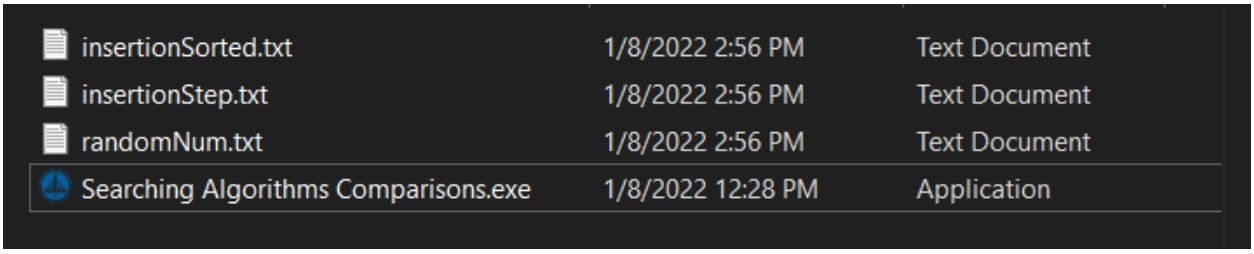

Figure 12: Text Files in Directory

Note: The mentioned feature which works only on Windows machines is for the text files to open with a click of a button from within the app, as it uses the Windows’ notepad app for having the same absolute path on all Windows machines.

Figure 13: Import and Draw Button

This button allows the user to import external test data and sort them to acquire the comparative results between the algorithms selected and the step size that is written.

2. Insertion Sort

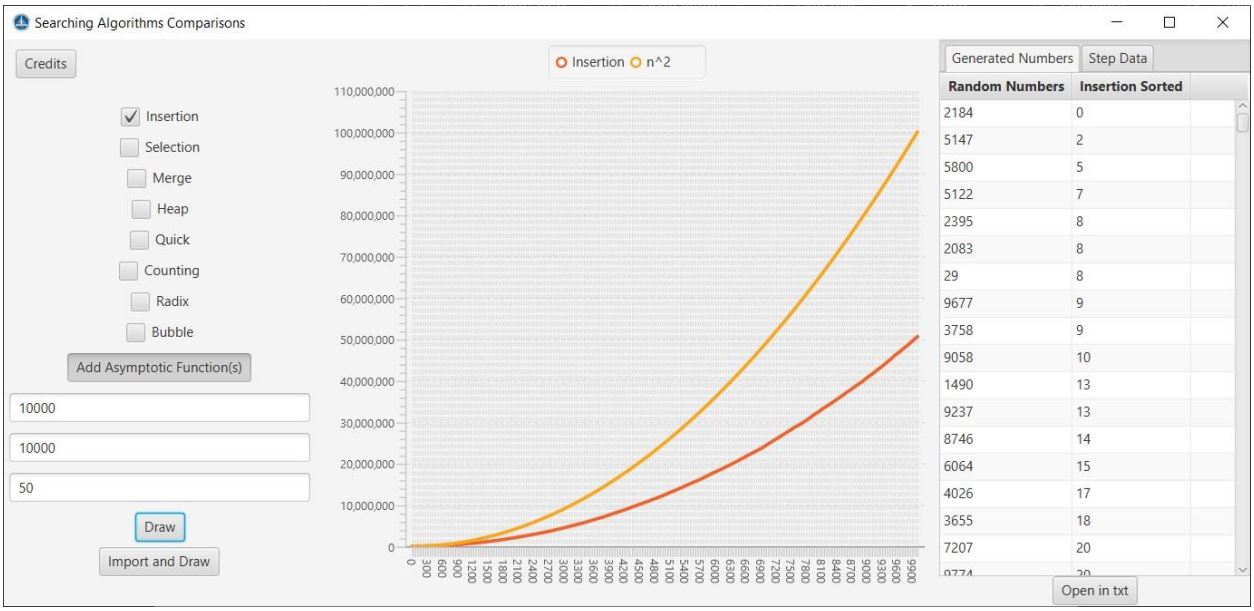

Figure 14: Insertion Sort Asymptotic Result

The Insertion sort is implemented using the following function:

```
void insertionSort(ArrayList<Integer> arr, int len){
    for(int i =1;i<len;i++){
        int key = arr.get(i);
        int j = i-1;
        while(j >= 0 && arr.get(j)>key){
            arr.set(j+1,arr.get(j));
            j--;
            step+=2;
        }
        arr.set(j+1,key);
        step += 3;
    }
}
```

The step variable is a global variable that is set to 0 before entering the sorting function. The sorting function is driven by a main function that runs the sorting algorithm at different array sized to then plot the graph.

3. Selection Sort

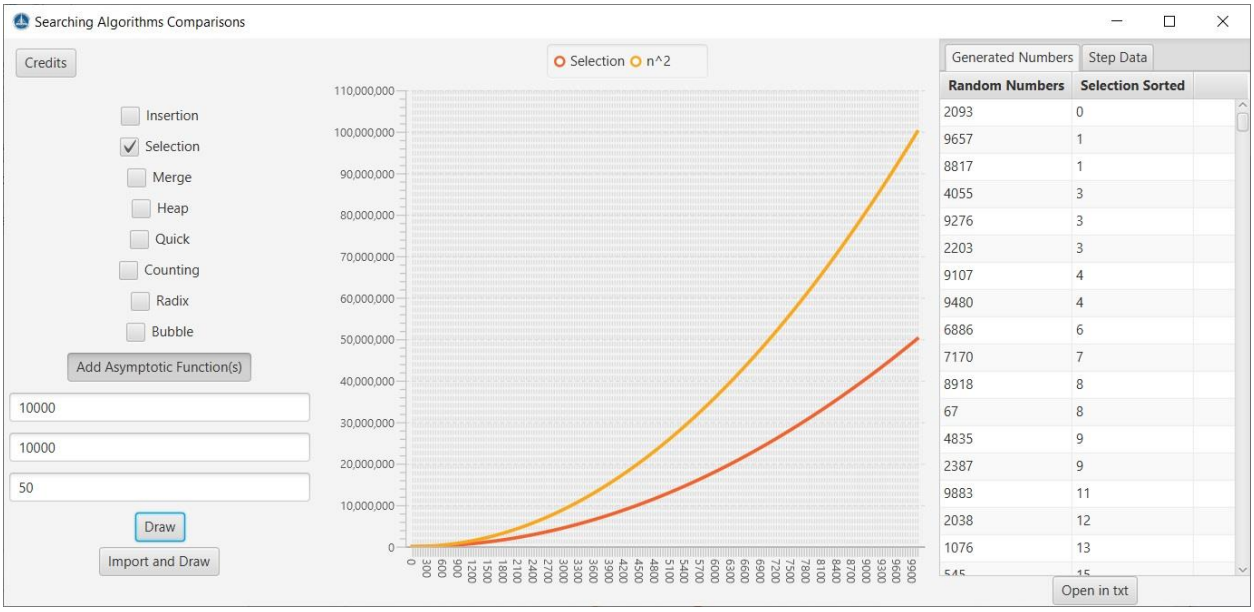

Figure 15: Selection Sort Asymptotic Result

The Selection sort is implemented using the following function:

```
void selectionSort(ArrayList<Integer> arr, int len){
    int min,temp;
    for(int i =0;i<len-1;i++){
        min = i;
        for(int j =i+1;j<len;j++){
            if(arr.get(j)<arr.get(min)){
                min =j;
                step++;
            }
            step++;
        }
        temp = arr.get(i);
        arr.set(i,arr.get(min));
        arr.set(min,temp);
        step+=4;
    }
}
```

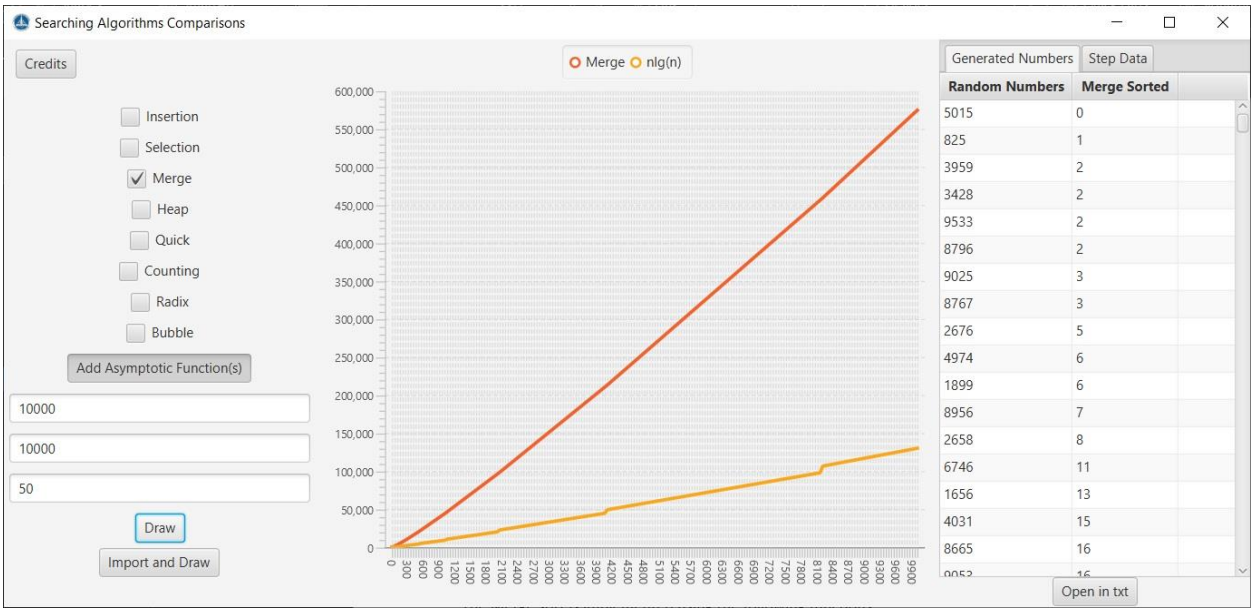

Figure 16: Merge Sort Asymptotic Result

4. Merge Sort

The Merge sort is implemented using the following functions:

```
void mergeSort(ArrayList<Integer> arr, int s, int l){
    if(s>=l){
        return;
    }
    int mid = (s+l)/2;
    step++;
    mergeSort(arr,s,mid);
    mergeSort(arr,mid+1,l);
    merge(arr,s,mid,l);
}

void merge(ArrayList<Integer> arr, int s, int m, int l){
    step += 3;
    ArrayList<Integer> firstarr = new ArrayList<Integer>();
    ArrayList<Integer> secondarr = new ArrayList<Integer>();
    for(int i=0;i<m-s+1;i++){
        firstarr.add(arr.get(s+i));
        step++;
    }
    for(int i=0;i<l-m;i++){
        secondarr.add(arr.get(m+i+1));
        step++;
    }
    int i=0, j=0, k=s;

    while(i<m-s+1 && j<l-m){
        if(firstarr.get(i)<secondarr.get(j)){
            arr.set(k, firstarr.get(i));
            i++;
```

```
        }else{
            arr.set(k,secondarr.get(j));
            j++;
        }
        k++;
        step+=3;
    }
    while(i<m-s+1){
        arr.set(k,firstarr.get(i));
        i++; k++; step+=3;
    }
    while(j<l-m){
        arr.set(k,secondarr.get(j));
        j++; k++; step+=3;
    }
}
```

5. Heap Sort

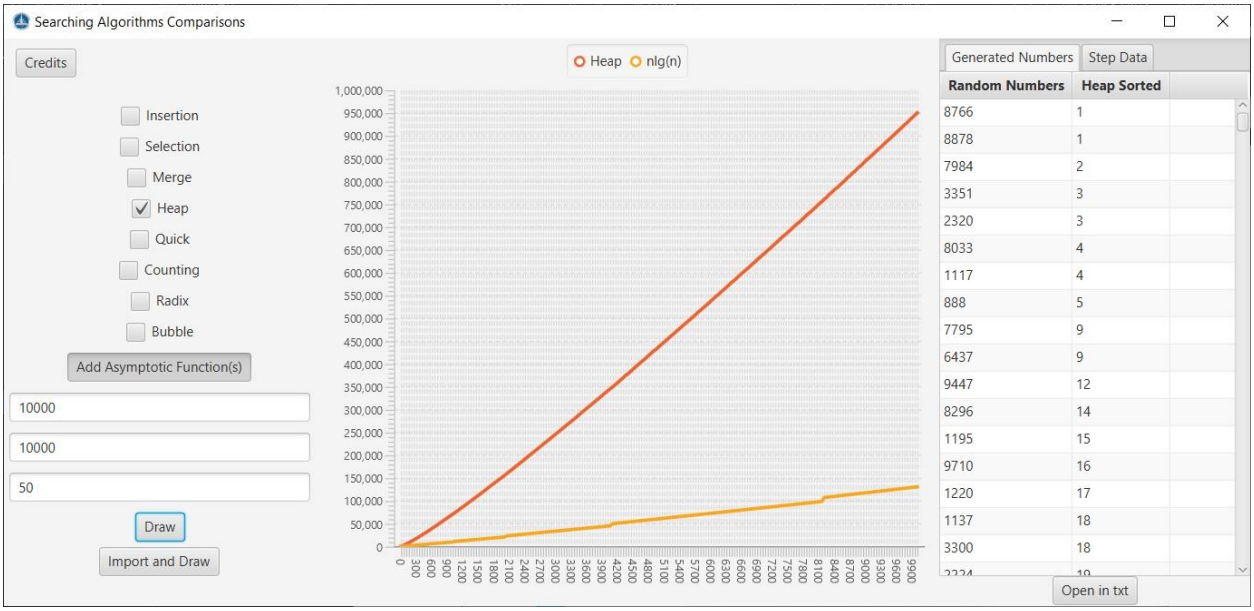

Figure 17: Heap Sort Asymptotic Result

The Heap sort is implemented using the following functions:

```
void heapSort(ArrayList<Integer> arr, int len){
    int temp;
    buildMaxHeap(arr,len);
    for(int i = len-1;i>0;i--){
        temp = arr.get(0);
        arr.set(0,arr.get(i));
        arr.set(i,temp);
        maxHeapify(arr, 0, i);
        step+=3;
    }
}

void buildMaxHeap(ArrayList<Integer> arr, int len){
    for(int i = len/2;i>=0;i--){
        maxHeapify(arr,i,len);
        step++;
    }
}

void maxHeapify(ArrayList<Integer> arr, int i, int len){
    step += 4;
    int l = (2*i)+1;
    int r = (2*i)+2;
    int largest;
    if(l<len && arr.get(l)>arr.get(i)){
        largest = l;
    }else{
        largest = i;
    }
    if(r<len && arr.get(r)>arr.get(largest)){
```

```
        largest = r;
        step++;
    }
    if(largest != i){
        int temp = arr.get(i);
        arr.set(i,arr.get(largest));
        arr.set(largest, temp);
        step+=3;
        maxHeapify(arr, largest, len);
    }
}
```

6. Quick Sort

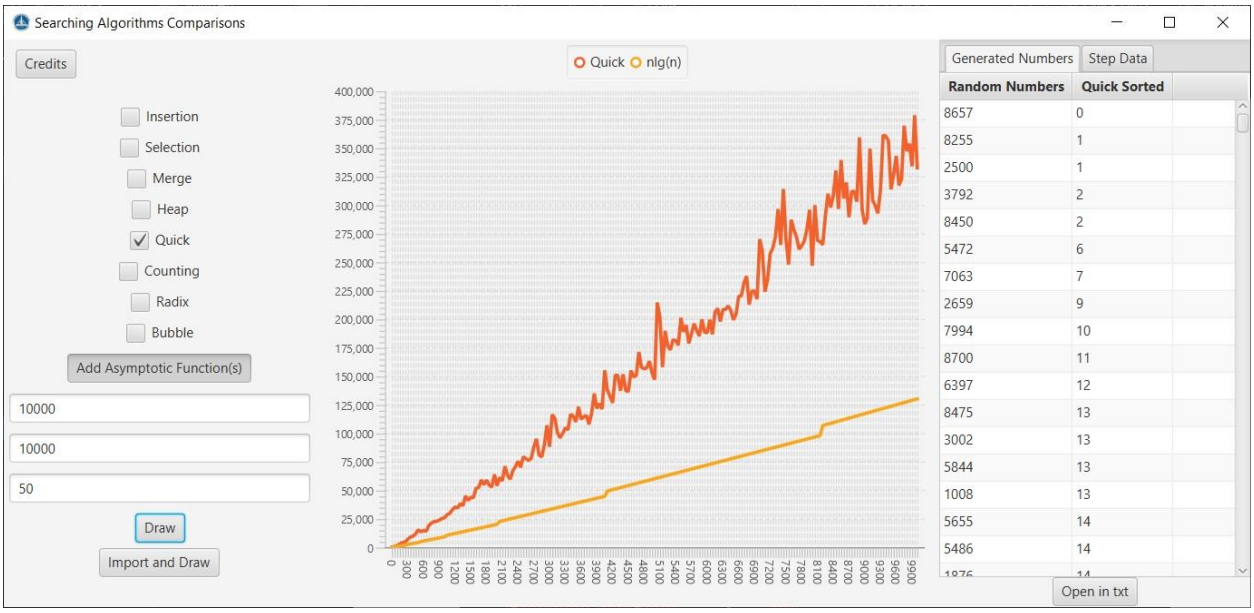

Figure 18: Quick Sort Asymptotic Result

The Quick sort is implemented using the following functions:

```
void quickSort(ArrayList<Integer> arr, int s, int l){
    if(s<l){
        int pi = partition(arr, s, l);
        quickSort(arr, s, pi-1);
        quickSort(arr, pi+1, l);
        step+= 3 ;
    }
}

int partition(ArrayList<Integer> arr, int s, int l){
    step += 5;
    int temp;
    int pivot = arr.get(l);
    int i = s-1;
    for(int j = s; j<=l-1;j++){
        if(arr.get(j)<pivot){
            i++;
            temp = arr.get(j);
            arr.set(j,arr.get(i));
            arr.set(i, temp);
            step+=4;
        }
    }
    temp = arr.get(l);
    arr.set(l,arr.get(i+1));
    arr.set(i+1,temp);
    return i+1;
}
```

7. Counting Sort

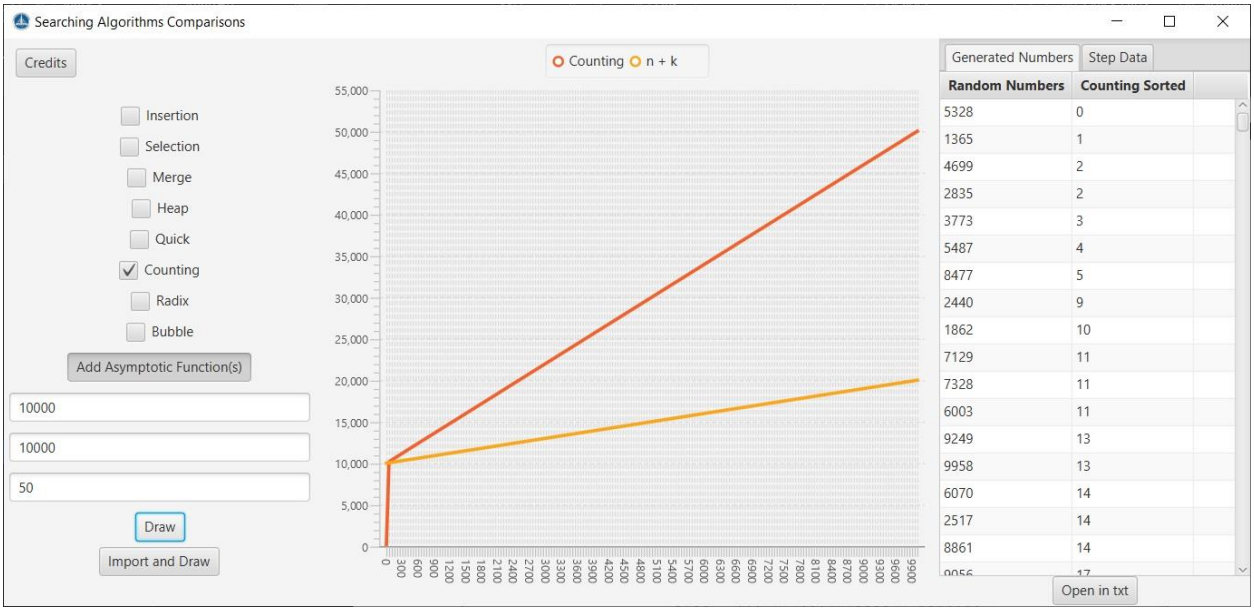

Figure 19: Counting Sort Asymptotic Result

The Counting sort is implemented using the following function:

```
void countingSort(ArrayList<Integer> arr, int len){
    step += 3;
    ArrayList<Integer> outarr = new ArrayList<Integer>();
    for(int j=0;j<len;j++){
        outarr.add(0);
        //step++;
    }
    if(len != 0){
        int max = arr.get(0);
        for(int i = 1; i < len; i++){
            if(arr.get(i)>max){
                max = arr.get(i);
                step++;
            }
        }
        ArrayList<Integer> count = new ArrayList<Integer>();
        for(int j=0;j<=max;j++){
            count.add(0);
            // step++;
        }
        for(int i = 0;i<len;i++){
            count.set(arr.get(i),count.get(arr.get(i))+1);
            step++;
        }
        for(int i = 1;i <= max;i++){
            count.set(i,count.get(i)+count.get(i-1));
            step++;
        }
        for(int i = len-1;i>=0;i--){
            outarr.set(count.get(arr.get(i))-1,arr.get(i));
```

```
            count.set(arr.get(i),count.get(arr.get(i))-1);
            step+=2;
        }
        for(int i = 0; i<len;i++){
            arr.set(i,outarr.get(i));
            step++;
        }
    }


}
```

The asymptotic notation of the counting sort is n+k whereas k is the largest element in the array.

8. Radix Sort

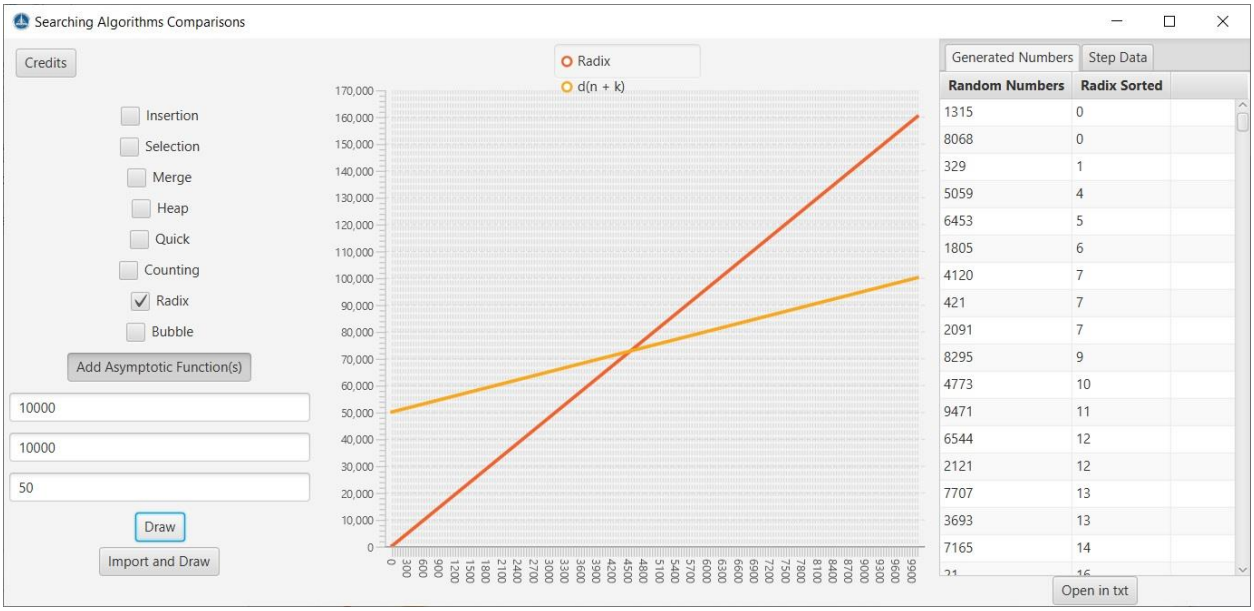

Figure 20: Radix Sort Asymptotic Result

The Radix sort is implemented using the following functions:

```
void radixSort(ArrayList<Integer> arr, int len){
    int max = arr.get(0);
    step++;
    for(int i = 1; i < len; i++){
        if(max < arr.get(i)){
            max = arr.get(i);
            step++;
        }
    }
    for(int d = 1; max / d > 0; d *= 10){
        countingRadix(arr, d, len);
    }
}

void countingRadix(ArrayList<Integer> arr, int d, int len){
    int[] count = {0,0,0,0,0,0,0,0,0,0};
    for(int i = 0; i<len; i++){
        count[(arr.get(i) / d) % 10]++;
        step++;
    }
    for(int i = 1; i<10; i++){
        count[i]+=count[i-1];
        step++;
    }
    ArrayList<Integer> outarr = new ArrayList<Integer>();
    step += 2;
    for(int j=0;j<len;j++){
        outarr.add(0);
        step++;
```

```
    }
    for(int i = len-1;i>=0; i--){
        outarr.set(--count[(arr.get(i)/d) % 10],arr.get(i));
        step++;
    }
    for(int i = 0; i<len; i++){
        arr.set(i,outarr.get(i));
        step++;

    }
}
```

The radix sort uses an alternate version of the counting sort as it is a stable method of sorting. This alternate method sorts based on a specific digit within the digits if the numbers.

The radix sort asymptotic notation in d(n+k) whereas d is the number of digits in the largest element in the array.

9. Bubble Sort

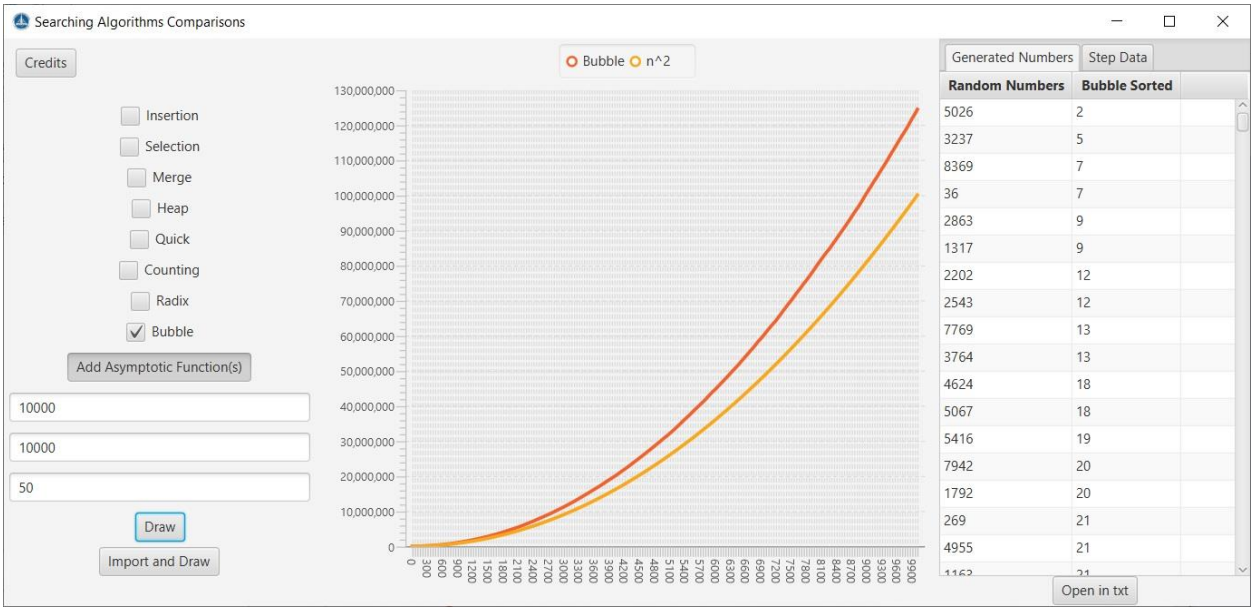

Figure 21: Bubble Sort Asymptotic Result

The Bubble sort is implemented using the following function:

```
void bubbleSort(ArrayList<Integer> arr, int len){
    for(int i =0;i<len-1;i++){
        for(int j=0;j<len-i-1;j++){
            step++;
            if(arr.get(j)> arr.get(j+1)){
                step+=3;
                int temp = arr.get(j);
                arr.set(j,arr.get(j+1));
                arr.set(j+1,temp);
            }
        }
    }
}
```

10. Different Comparing Results

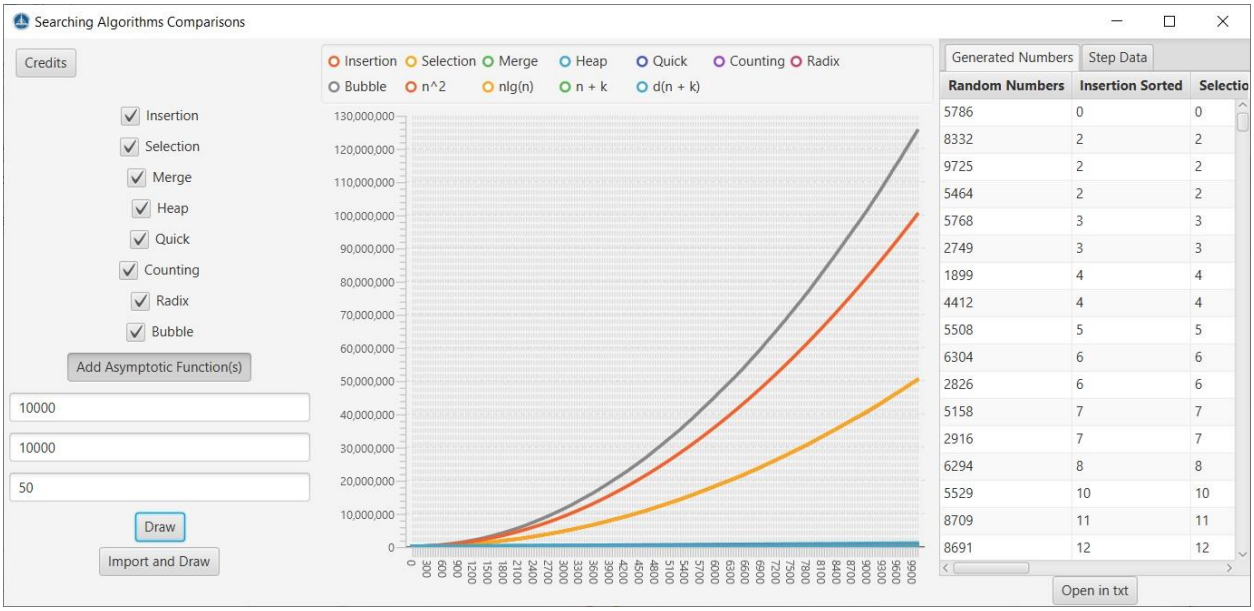

Figure 22: All Algorithms with Asymptotic Graph

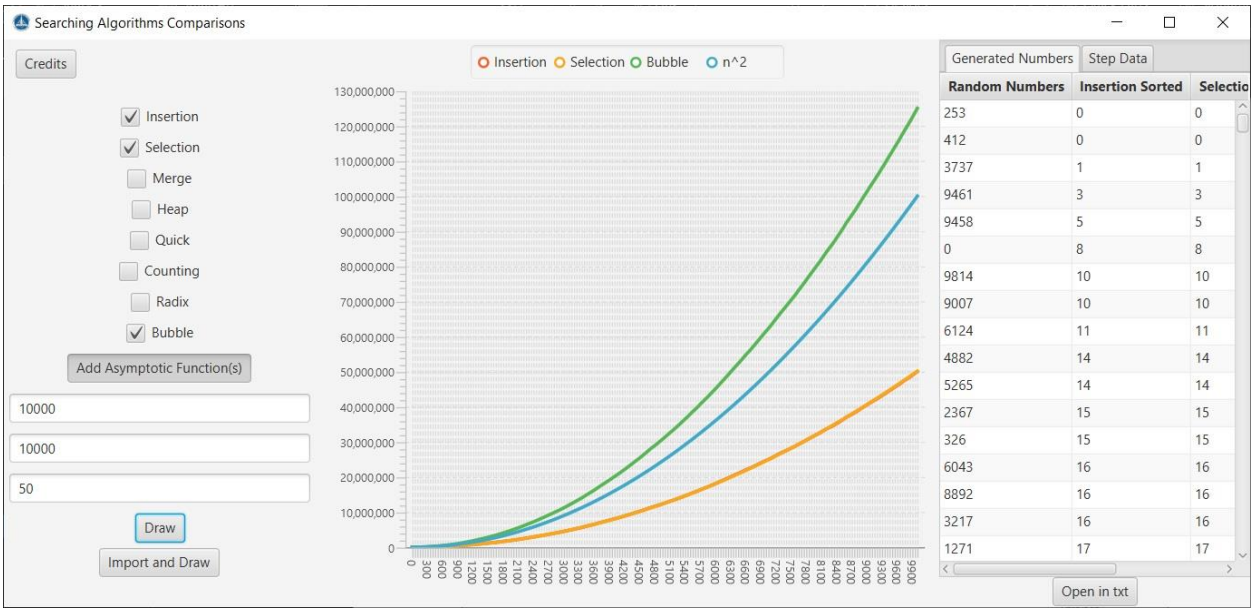

Figure 23: Insertion, Selection and Bubble with Asymptotic Graph

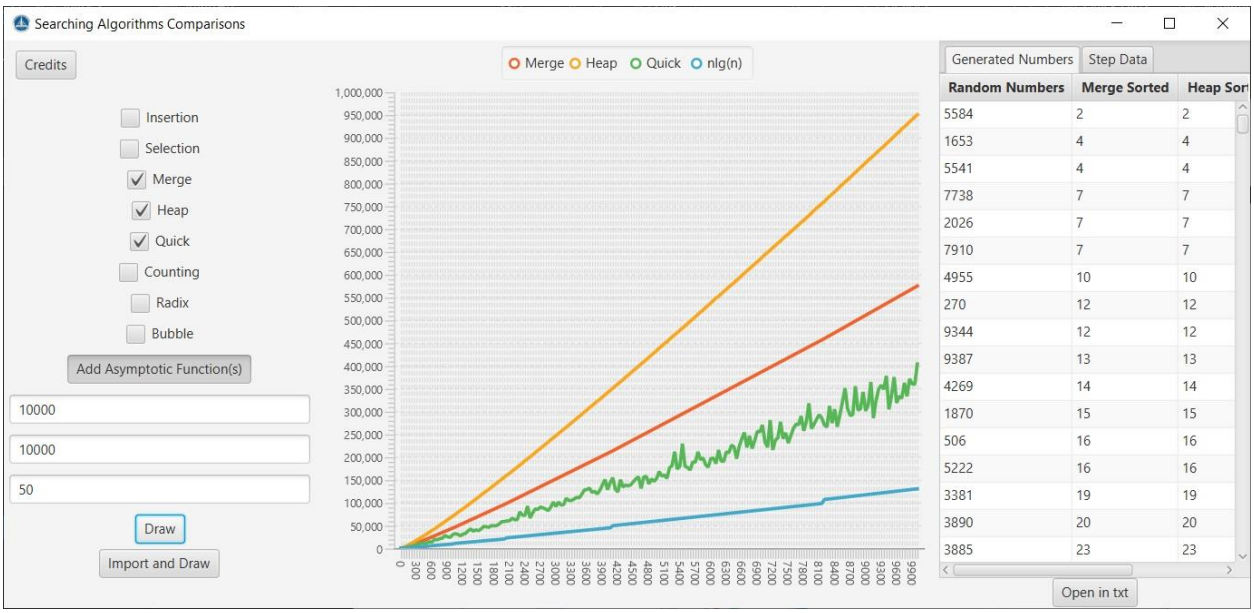

Figure 24: Merge, Heap and Quick with Asymptotic Graph

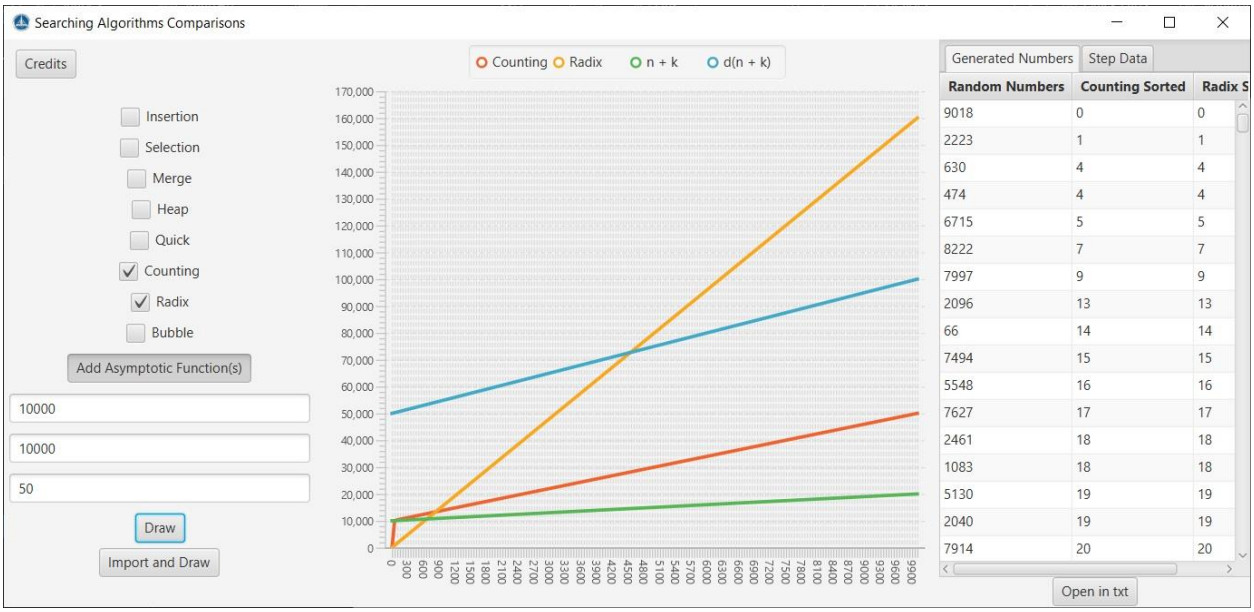

Figure 25: Counting and Radix with Asymptotic Graph

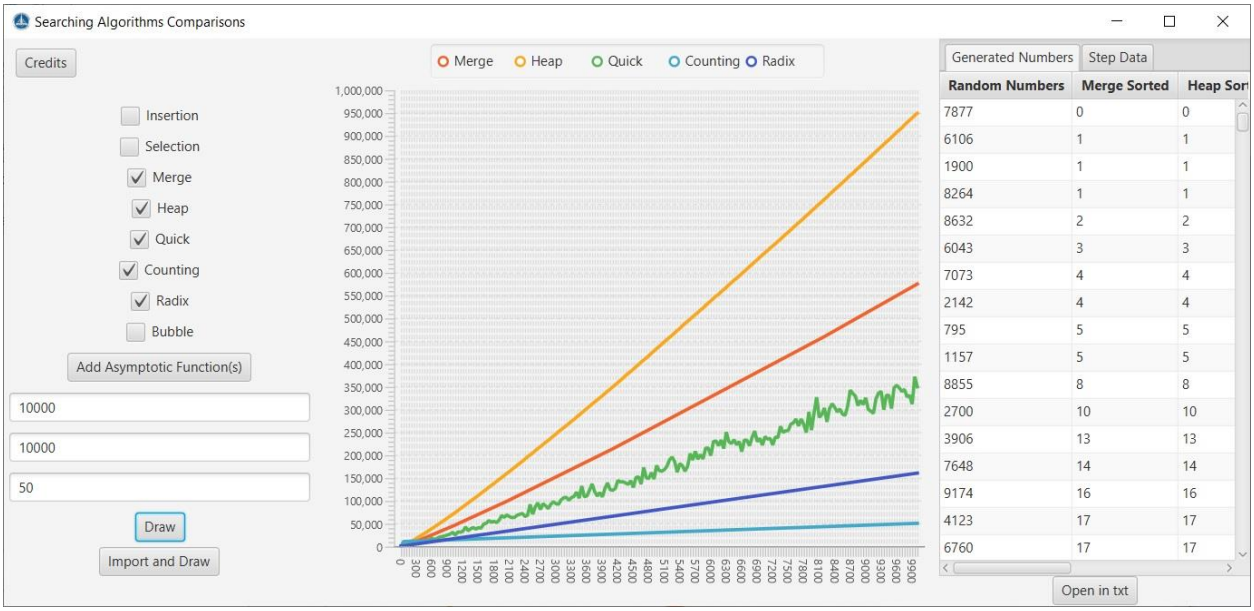

Figure 26: Merge, Heap, Quick, Counting and Radix Graph