CO322: DATA STRUCTURES AND ALGORITHMS
LAB1-SORTING ALGORITHMS

K.G.I.S. NAWARATHNA
E/17/219
CO322
28/06/2021

CODE:

```java
 * Simple sorting algorithms and their performance 

import java.util.*; 

public class Sort {

    //create an array of given size and populate it with random data
    static int [] create_rand_data(int size_of_array) {
    int [] data = new int[size_of_array];
    int i;
    for(i=0; i < data.length; i++)
        data[i] = (int)(Math.random() * 100);
    return data;
    }

    //create an array of given size and populate it with worst data arrangement
    static int [] create_worst_data(int size_of_array) {
    int [] data = new int[size_of_array];
    int i;
    for(i=0; i < data.length; i++)
        data[i] = data.length - i;
    return data;
    }

    //create an array of given size and populate it with best data arrangement
    static int [] create_best_data(int size_of_array) {
    int [] data = new int[size_of_array];
    int i;
    for(i=0; i < data.length; i++)
        data[i] = i;
    return data;
    }

    // function to swap. Would be useful since all need this
    static void swap(int []d, int i, int j) {
    int tmp = d[i];
    d[i] = d[j];
    d[j] = tmp;
    }

    // check if the soring worked on the array
    static boolean isSorted(int [] data) {
    int i;
    for(i=1; i < data.length; i++)
        if(data[i] < data[i-1]) break;
    return (i == data.length);
    }

    // If you want just display the array as well :)
    static void display(int []data) {
    System.out.println("=======");
    for(int i=0; i < data.length; i++)
        System.out.print(data[i] + "  ");
    System.out.println("\n=======");
    }


    /***********************************************************
     *     Implementation of sorting algorithms               *
     ***********************************************************/
    static void buble_sort(int [] data)  {
    // Implement

        boolean quit=false;
        for(int i=0;i<data.length && !quit;i++) {
            quit = true;
            for(int j = data.length-1;j>i;j--) {
                if(data[j]<data[j-1]) {
                    swap(data,j,j-1);
                    quit = false;
                }
            }
        }
    }
```

```java

    static void selection_sort(int [] data) {
    // Implement
        for(int i=0;i<data.length;i++) {
            int minIndex=i;
            for(int j=data.length-1;j>i;j--) {
                if(data[minIndex]>data[j]) {
                    minIndex = j;
                }
            }
            swap(data,i,minIndex);
        }
    }

    static void insertion_sort(int [] array) {
    // Implement
        int n = array.length;
        for (int j = 1; j < n; j++) {
            int key = array[j];
            int i = j-1;
            while ( (i > -1) && ( array [i] > key ) ) {
                array [i+1] = array [i];
                i--;
            }
            array[i+1] = key;
        }
    }
```

```java

    static void timer(int inputSize,String case_,String alg) {

        int myArray[] = null;

        if(case_=="best") {
            myArray = create_best_data(inputSize);
        }
        else if(case_=="average") {
            myArray = create_rand_data(inputSize);
        }
        else if(case_=="worst") {
            myArray = create_worst_data(inputSize);
        }

        long start = 0,finish = 0;

        if(alg=="bubble") {
            start = System.nanoTime();
            buble_sort(myArray);
            finish = System.nanoTime();
        }
        else if(alg=="selection") {
            start = System.nanoTime();
            selection_sort(myArray);
            finish = System.nanoTime();
        }
        else if(alg=="insertion"){
            start = System.nanoTime();
            insertion_sort(myArray);
            finish = System.nanoTime();
        }

        System.out.print("Time Taken to "+alg+" sorting for the "+case_+" case when input size is "+inputSize+":" );
        System.out.println((finish-start)/1000000);
    }

```

```java
    public static void main(String [] args) {
    // create arrays of different size populate with data
    // measure the time taken by different algorithms to
    // sort the array.
    // Think about effects of caches, other apps running etc.

        while(true) {
            Scanner sc= new Scanner(System.in);    //System.in is a standard input stream
            System.out.print("Enter the input size(enter 0 to terminate): ");
            int n= sc.nextInt();

            if(n==0) {
                break;
            }

            timer(n,"worst","bubble");
            timer(n,"best","bubble");
            timer(2*n,"worst","bubble");
            timer(2*n,"best","bubble");
            timer(3*n,"worst","bubble");
            timer(3*n,"best","bubble");
            timer(4*n,"worst","bubble");
            timer(4*n,"best","bubble");
            timer(5*n,"worst","bubble");
            timer(5*n,"best","bubble");
            timer(6*n,"worst","bubble");
            timer(6*n,"best","bubble");

            timer(n,"worst","selection");
            timer(n,"best","selection");
            timer(2*n,"worst","selection");
            timer(2*n,"best","selection");
            timer(3*n,"worst","selection");
            timer(3*n,"best","selection");
            timer(4*n,"worst","selection");
            timer(4*n,"best","selection");
            timer(5*n,"worst","selection");
            timer(5*n,"best","selection");
            timer(6*n,"worst","selection");
            timer(6*n,"best","selection");

            timer(n,"worst","insertion");
            timer(n,"best","insertion");
            timer(2*n,"worst","insertion");
            timer(2*n,"best","insertion");
            timer(3*n,"worst","insertion");
            timer(3*n,"best","insertion");
            timer(4*n,"worst","insertion");
            timer(4*n,"best","insertion");
            timer(5*n,"worst","insertion");
            timer(5*n,"best","insertion");
            timer(6*n,"worst","insertion");
            timer(6*n,"best","insertion");


        }
    }
}
```

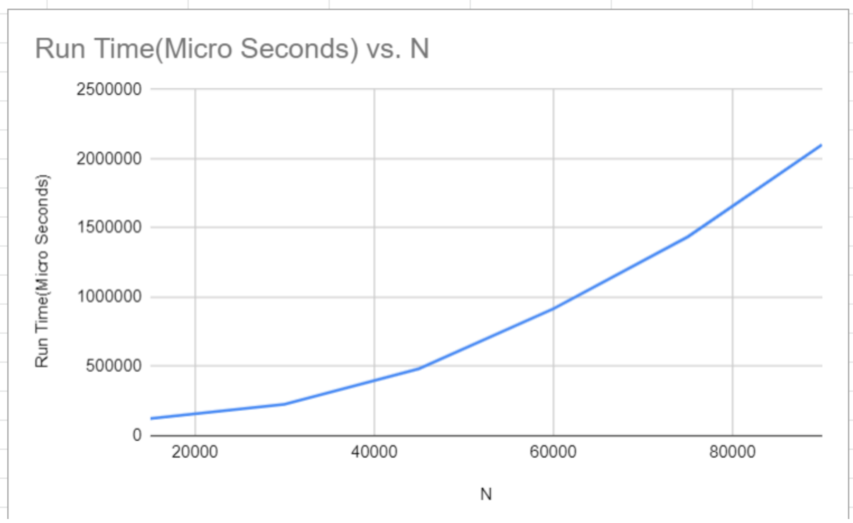
DISCUSSION

1.How does the performance vary with the input size?

-When measuring the running time against different input sizes the following were the results.

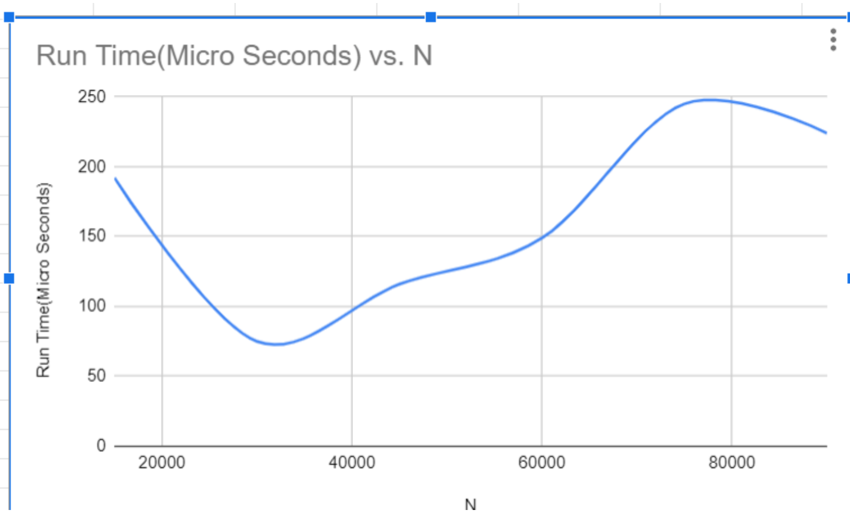
For bubble sort worst case:

Bubble Sort Worst Case	
N	Run Time(Micro Seconds)
15000	124482
30000	226930
45000	482663
60000	915616
75000	1435640
90000	2101151



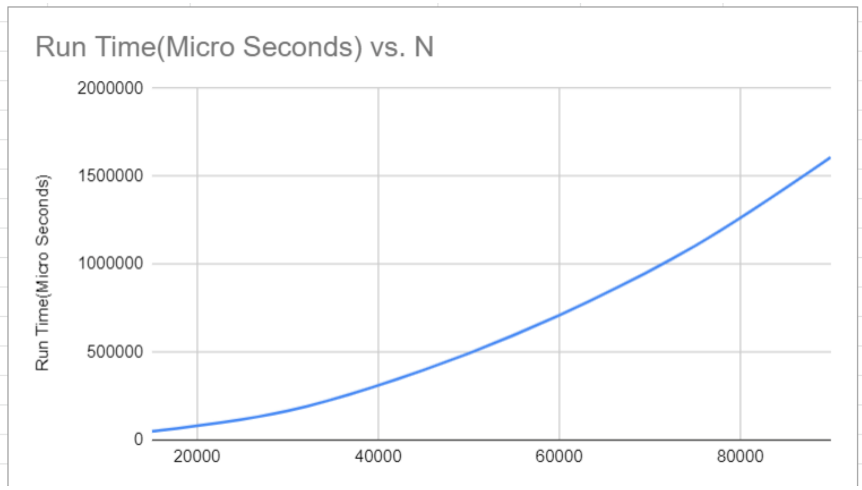
For bubble sort best case:

Bubble Sort Best Case	
N	Run Time(Micro Seconds)
15000	192
30000	75
45000	116
60000	149
75000	245
90000	224



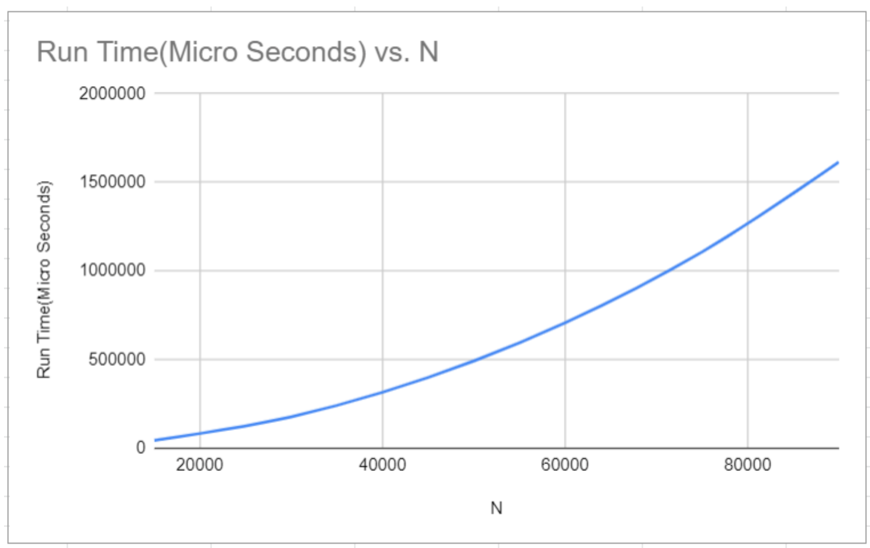
For selection sort worst case:

Selection Sort Worst Case	
N	Run Time(Micro Seconds)
15000	48964
30000	165130
45000	396865
60000	709530
75000	1103978
90000	1608166



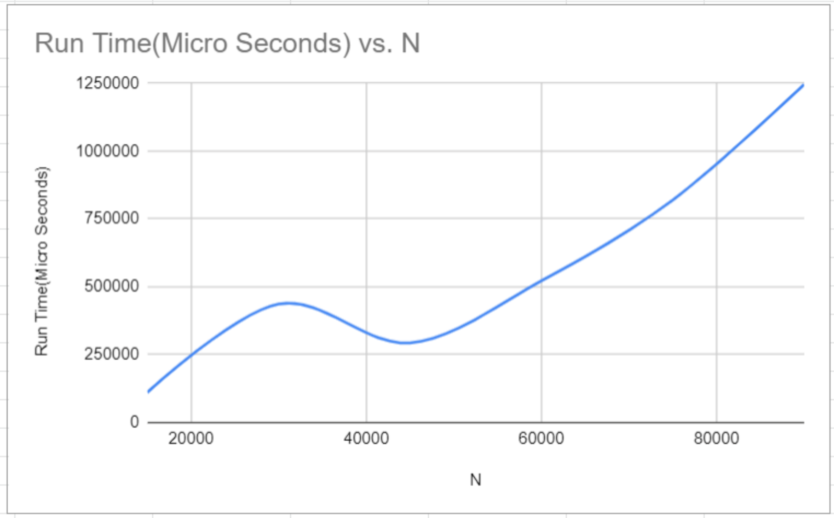
For selection sort best case:

Selection Sort Best Case	
N	Run Time(Micro Seconds)
15000	44422
30000	177013
45000	398778
60000	707430
75000	1107088
90000	1614585



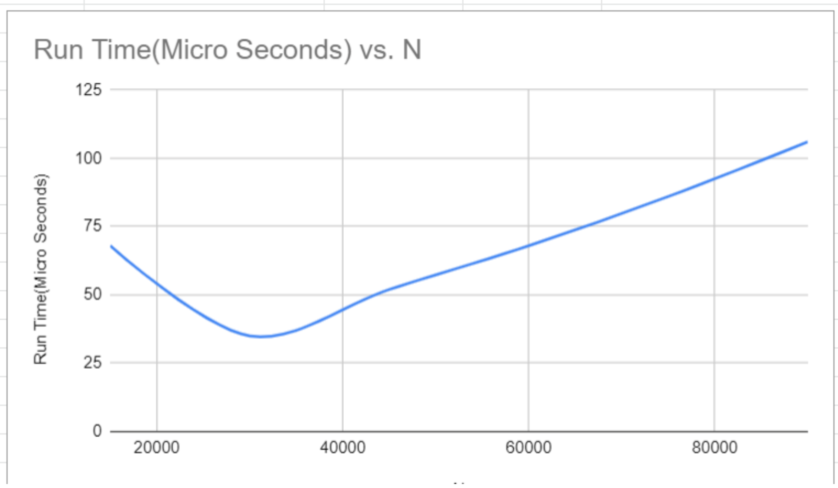
For insertion sort worst case:

Insertion Sort Worst Case	
N	Run Time(Micro Seconds)
15000	111294
30000	436764
45000	293435
60000	521820
75000	819483
90000	1245565



For insertion sort best case:

Insertion Sort Best Case	
N	Run Time(Micro Seconds)
15000	68
30000	35
45000	52
60000	68
75000	86
90000	106



OBSERVATIONS:

-Time was measured using micro seconds.
-For the best cases, most sorting algorithms produce running time most equal to 0 micro second except for selection sort.
-Selection sort produces exponential running time vs input size relationship for both cases.
-Bubble sort algorithm for best case produces a result somewhat similar to exponential behaviour. For the best case it produces a result which is mostly equal to 0 micro seconds.
- Insertion sort algorithms in the worst cases : The shape is similar but not exponential.At the start(for lower input size), then it starts to grow exponentially and gets flatten in the medium range of the graph.But at the last part the graph again enter to the exponential behaviour.
-Insertion sort algorithms in the best cases: The shape is almost similar in both algorithms. The running time is almost 0.At the start, they start with a high value for lower inputs and then gradually increase the running time.
-If the running time for randomly generated data were measured , it could see that the running time for sorting that data is higher than sorting for the worst case for the same input size.(This observation is only valid for bubble and selection sorting algorithms)
-When the input size is greater than 50,000 the code will take a lot of time to produce results for all the algorithms.
-Selection sort: This algorithm produces exponential behaviour for both the cases. Another significant characteristic is that both worst and best cases produce high running time whereas other algorithms produce very low running time for best case.

2.Does the empirical results you get agree with theoretical analysis?

Most of the time the practical performance of the algorithm is different from the theoretical performance.

For the bubble sort algorithm, worst case have a time complexity of big O of N^2 theoretically .The practical results are not exactly that , but somewhat similar.In theory , when N gets doubled , running time should be 4 times the previous.But this is not the exact result but very similar for that.For the best case bubble sort has N time complexity but in the practical scenario this is not the case.In practical scenario, the running time is high for smaller inputs but as the number of inputs increases the running time tends to be reduced and then try to be constant around 0 micro seconds.

If the selection sort is considered, it can see that the practical results almost agree with the theoretical results. As the theory suggests , the practical graph takes a N^2 behaviour for both the best and worst case scenarios.A very interesting fact to note here is that , selection sort is almost the same as for both the cases which are not shown from other algorithms.

For the insertion sort algorithm, worst case have a time complexity of big O of N^2.The theoretical results are not exactly that , but somewhat similar.The only difference with the theoretical result is that the graph tends to be constant for medium input sizes.For the best case bubble sort has N time complexity but in the practical scenario this is not the case.In practical scenario, the running time is very high for smaller inputs but as the number of inputs increases the running time tends to be reduced and then try to be constant.

There can be few reasons for these differences for some of these algorithms.The algorithms were analysed in a very abstract way in theoretical parts. But the actual representation of the running time is not very clearly explained in theory but just a rough idea. When we take an algorithm into consideration , there are more operations than swapping and comparisons. Once the algorithm is converted into assembly code there are numerous instructions that the algorithm consists of.

Running time or the performance of an algorithm in practice depends on a lot of facts. Hardware of the system, programming language(which is similar in this lab:Java), compiler and many more factors. But in the theoretical analysis these were ignored in order to analyze only the performance of the algorithm and not depending on other factors. But in the real situation all of those other factors count as well. Therefore this kind of behaviour of the algorithm can be expected.

3.How did/should you test your code? Explain the test cases you used and the rationale for using them.

The main goal of this lab is to find how the algorithms perform in the real world when the input size increases. Therefore a mechanism to measure the running time of an algorithm should be implemented in the code. This is done by using the System.nanotime() method. Best case and worst cases were considered for all algorithms. Then the input size is increased such that the input size is N,2N,3N,.. to see how the running time varies with those input sizes. The input size was increased to a certain degree that after the maximum input size , the algorithm takes a lot of time to produce results.

It is important to reduce the time measuring errors. In order to do that , the time was measured for the same input size and with the same algorithm under the same case and calculated the average value. By doing this it can reduce the caching time and few timing errors.

All of the timing for all the algorithms were measured in one run.That means in one execution run times for all the algorithms were measured. This is done because if we check algorithms one by one it cannot be considered as the same state of the system. Since the system changes its behaviour very often, one execution can produce results very different to the other execution. Therefore it needs to measure running time for all the algorithms in one execution.