CO322 - LAB 01

SORTING ALGORITHMS

WIJERATHNE E.S.G.

E/18/397

SEMESTER 05

13/10/2022

Coding Implementation

```java
// Bubble sort
    static void buble_sort(int [] data)  {
        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;
                }
            }
        }

    }

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

    }

    // insertion sort
    static void insertion_sort(int [] data) {
    for(int i =1; i< data.length; i++){
        int val = data[i];
        int j =i -1;
        while(j>=0 && data[j]>val){
            data[j+1]=data[j];
            j = j- 1;
        }
        data[j+1] = val;
    }
    }
```

Figure 01: Implementations of sorting algorithms.

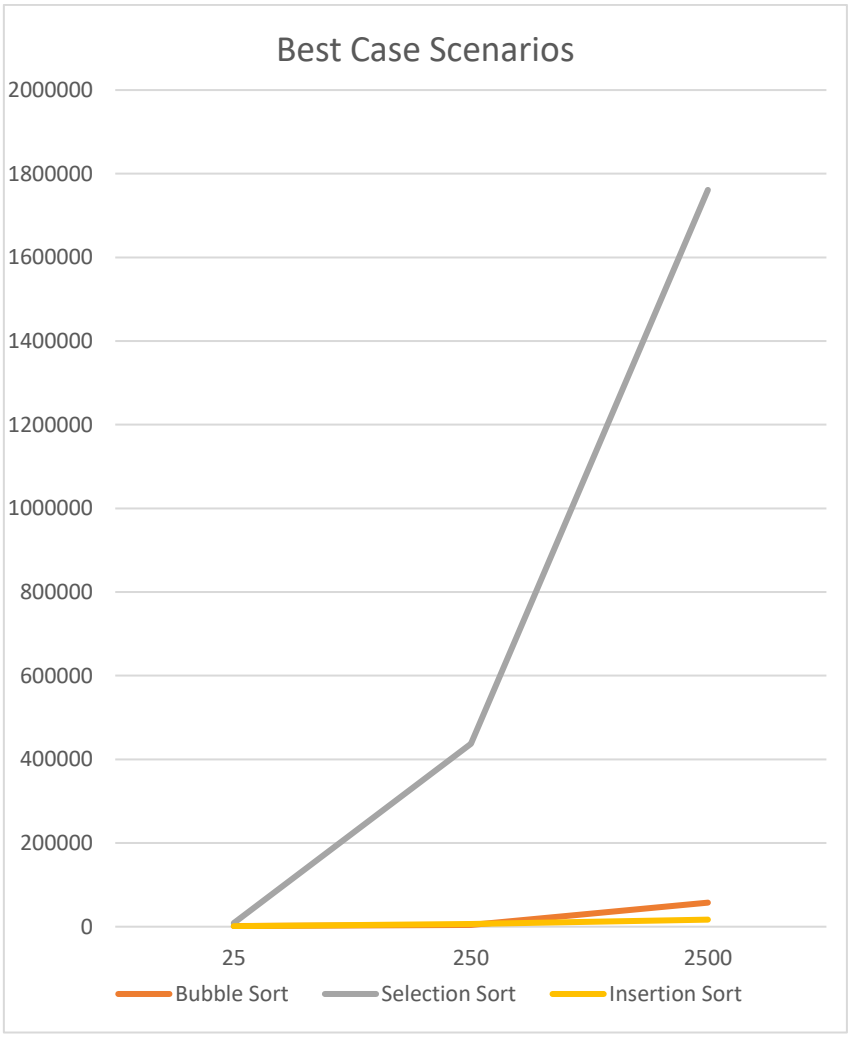
DISCUSSION

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

Case 01: Best case scenarios

Size of Array	Algorithm times in nanoseconds		
	Bubble Sort	Selection Sort	Insertion Sort
25	1600	8700	1300
250	4400	437700	6800
2500	57500	1761300	16900
25000	85300	174193500	46700
250000	221000	16517955700	440400

Table 01: variation of time with the array size and used algorithms

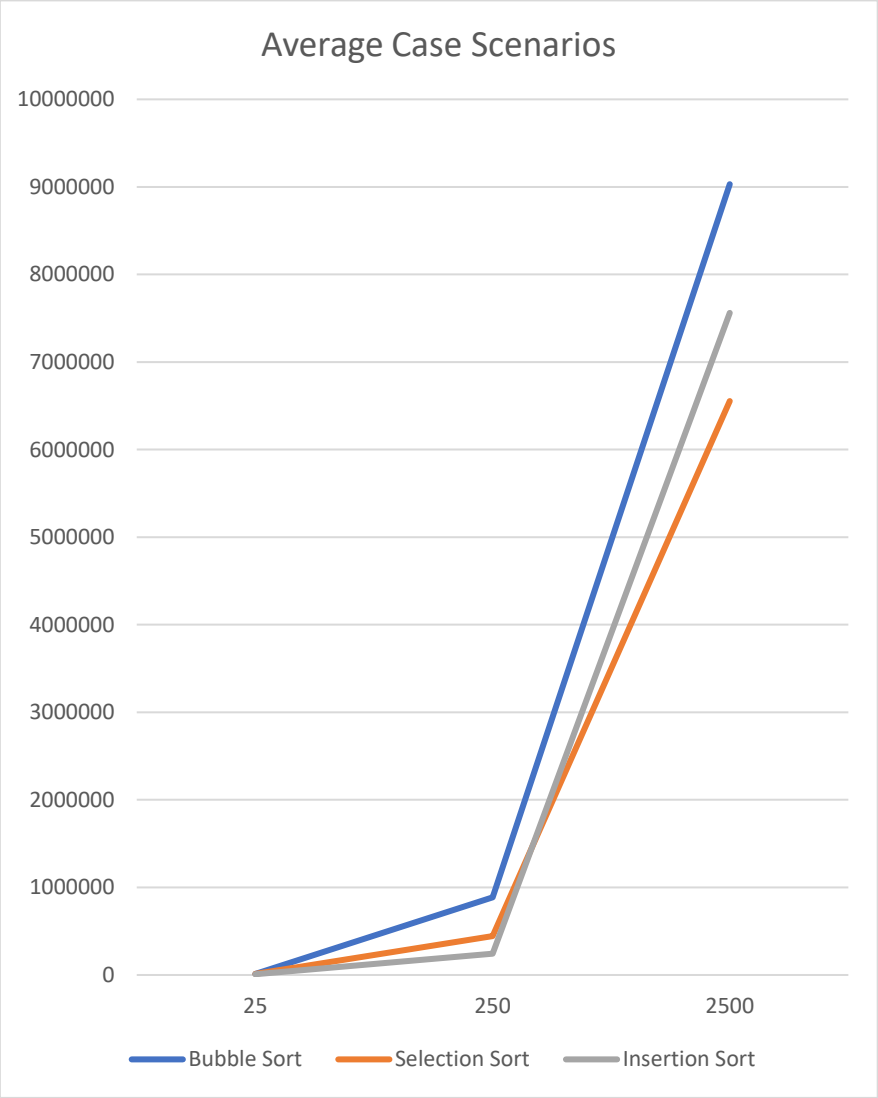


Graph 01: Time exceeds vs algorithm type and array size

Case 02: Average case scenarios

Size of Array	Algorithm times in nanoseconds		
	Bubble Sort	Selection Sort	Insertion Sort
25	12400	12700	9000
250	888300	443400	243000
2500	9031000	6554500	7561400
25000	815166800	183390100	57101700
250000	84344018500	16509377800	5306677100

Table 02: variation of time with the array size and used algorithms

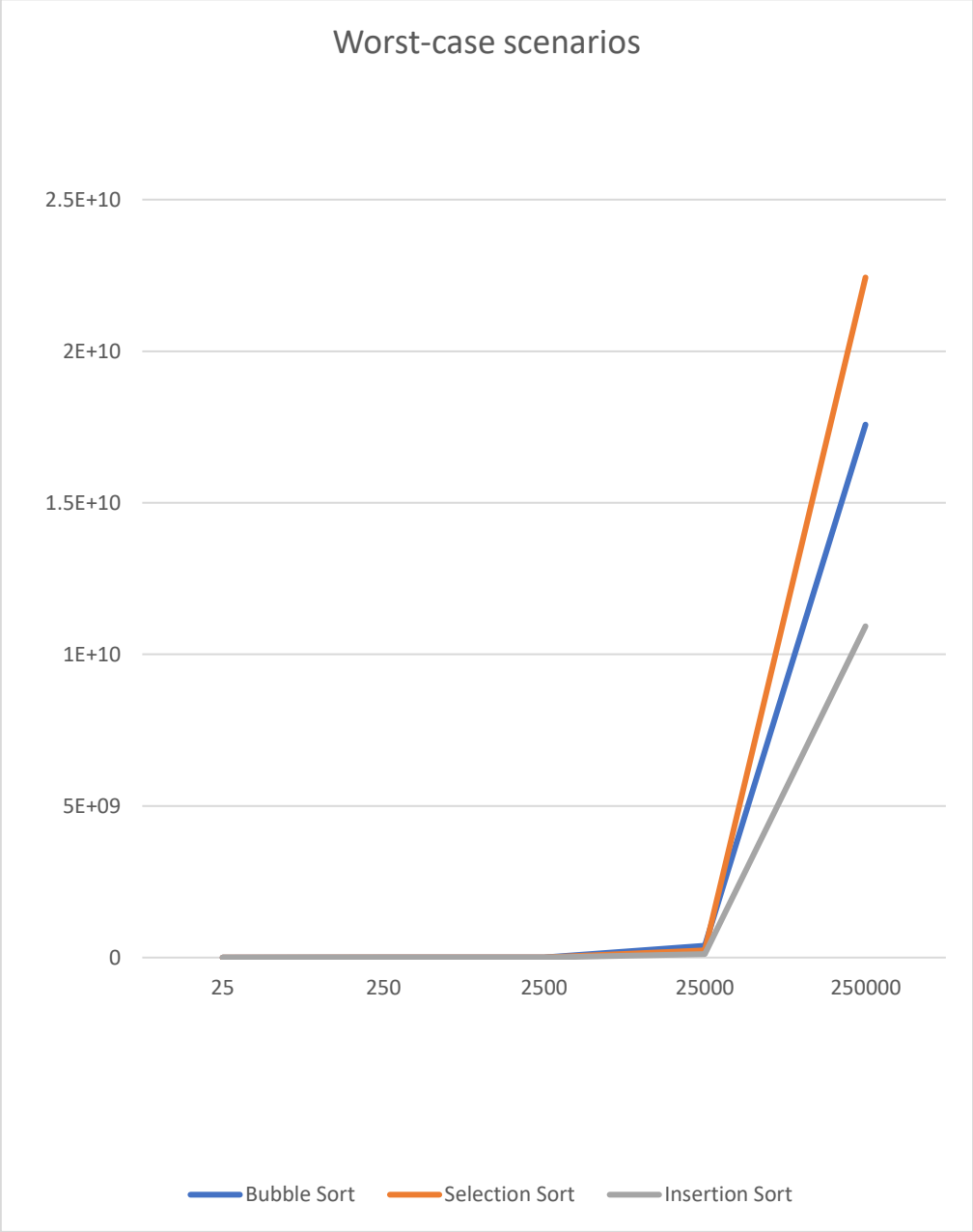


Graph 02: Time exceeds vs algorithm type and array size

Case 03: Worst-case scenarios

Size of Array	Algorithm times in nanoseconds		
	Bubble Sort	Selection Sort	Insertion Sort
25	28200	6300	6500
250	1113200	152200	527000
2500	4058500	2472100	6130500
25000	392484000	237258900	113984000
250000	17578901000	22432207900	10927040400

Table 03: variation of time with the array size and used algorithms



Graph 03: Time exceeds vs algorithm type and array size

Observations: -

- Time was measured using nanoseconds.
- For Best case scenarios both Bubble sort and insertion sort performed well except for selection sort for all array sizes.
- For average case scenarios for the small size of arrays all 3 algorithms resulted in nearly the same times.
- When array size increases for average scenarios bubble sorting took much time.
- For worst-case scenarios selection and insertion sort is given the best results for small array sizes.
- When array size increases in worst cases all three sorting algorithms took much time.

02. Do the empirical results you get to agree with the theoretical analysis?

- Big Oh notation was used to compare the performance of 3 sorting algorithms theoretically
- The run times of those sorting algorithms are shown below. (N-number of elements in the array)

Sorting Algorithm	Time Complexity		
	Best Case	Average Case	Worst Case
Bubble Sort	O(N)	$O(N^2)$	$O(N^2)$
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$
Insertion Sort	O(N)	$O(N^2)$	$O(N^2)$

Table 04: Theoretical data for sorting algorithms

- According to the table above, we can see that all three sorting algorithms have the same running time for the worst-case scenario.
- If we double the array size, the run time would be 4 times bigger than the previous one in the worst-case scenario.

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

For the bubble sort algorithm, the worst case has 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 a practical scenario, the running time is high for smaller inputs.

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 N^2 behavior for both the best and worst-case scenarios.

For the insertion sort algorithm, the worst case has 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.

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

03. 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 cases and worst cases were considered for all algorithms. Then the input size is increased such that the input size is 25, 250, 2500, 25000, and 250000 to see how the running time varies with those input sizes. The input size was increased to a certain degree and after the maximum input size, the algorithm takes a lot of time to produce results.

All the timing for all the algorithms was 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, they cannot be considered as the same state of the system. Since the system changes its behavior very often, one execution can produce results very different from the other execution. Therefore, it needs to measure the running time for all the algorithms in one execution.