Carla Fernández González. UO244965

Algorithmics: Sorting algorithms

LAB 3

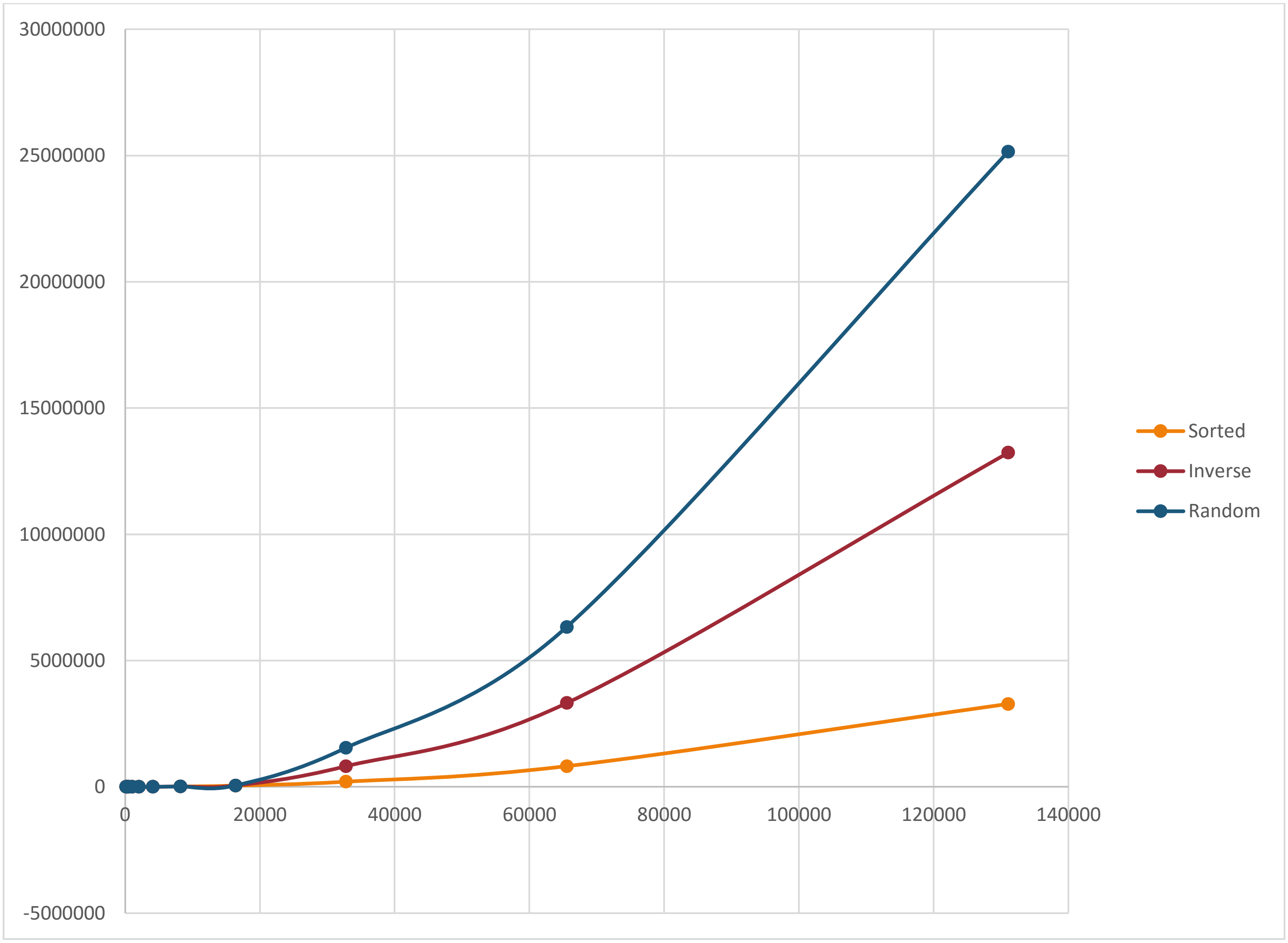
We were asked to fill some java files with the different sorting methods we saw during theory lectures. These were: bubble, insertion, selection and quicksort (with the central element as a pivot). In the following pages I will explain the results.

Note: for random sorting of the vector I chose a maximum random value of 100.

Note: times in the graphs axis are in micro seconds; they can be consulted in the Excel file.

Bubble algorithm:

From the bubble results I got the following graph:

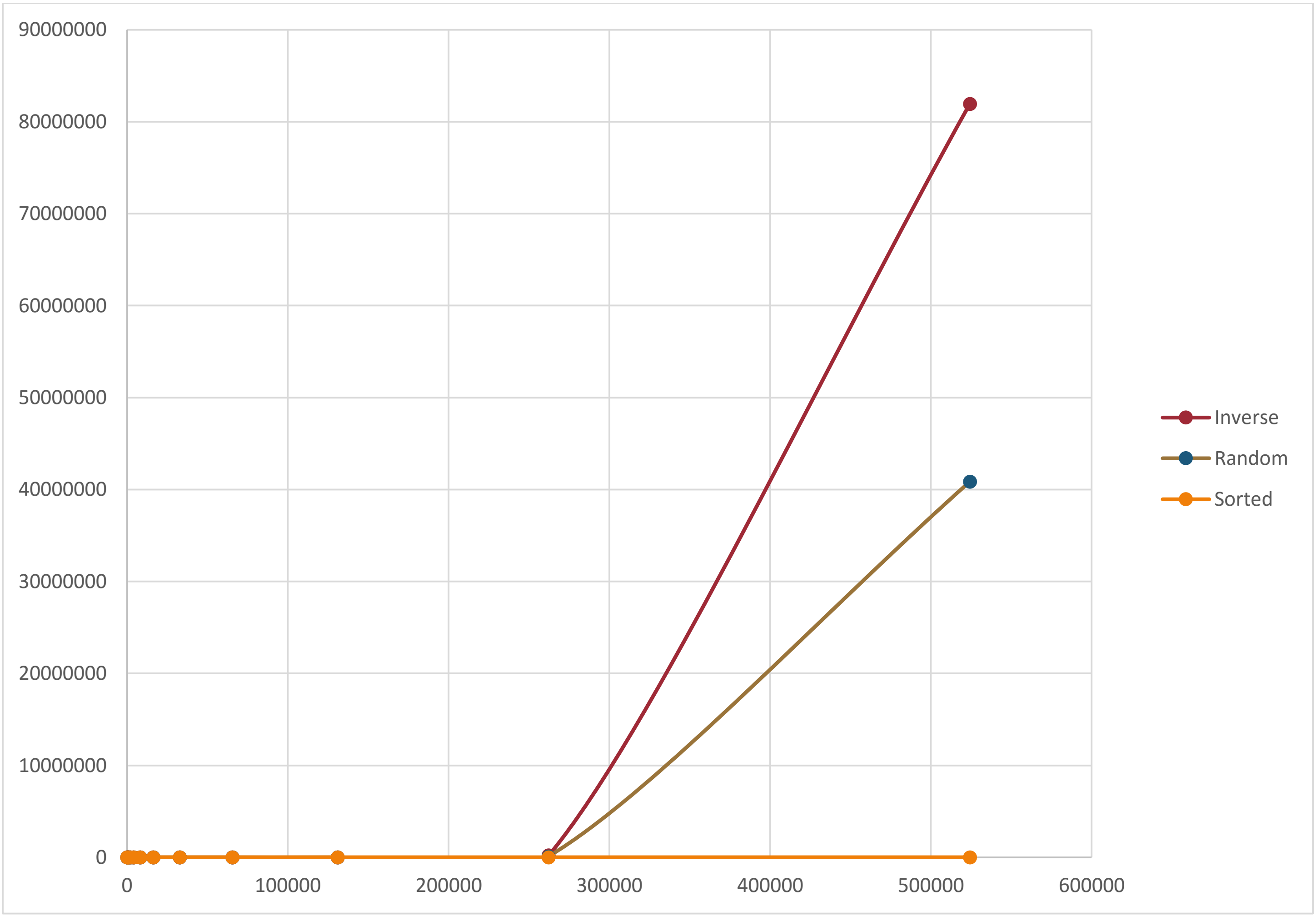


We can see that, logically, the bubble algorithm performs best when the vector to be sorted is *already* sorted. However, it takes quite a lot of time because it must still iterate through the whole array with a complexity of O (n^2). That is, it does not "*detect*" that the vector is already sorted. This special case would be immensely improved by adding a break condition in the loop if we have not made any exchange in the last iteration (bubble with sentinel).

As for the inverse and the random sorts, I cannot explain why sorting an inversely sorted vector is faster than a vector with random numbers. An inversely sorted vector should be the worst case scenario, requiring one interchange for every number on the vector minus one (that is, i – 1 interchanges). A randomly sorted vector should not have to reach that number of interchanges because probably not every number would need replacing.

In conclusion, we can see that the complexity for all the different types of sorting is the same, O (n^2), and that the algorithm performs quite differently depending on the sorting of the vector given.

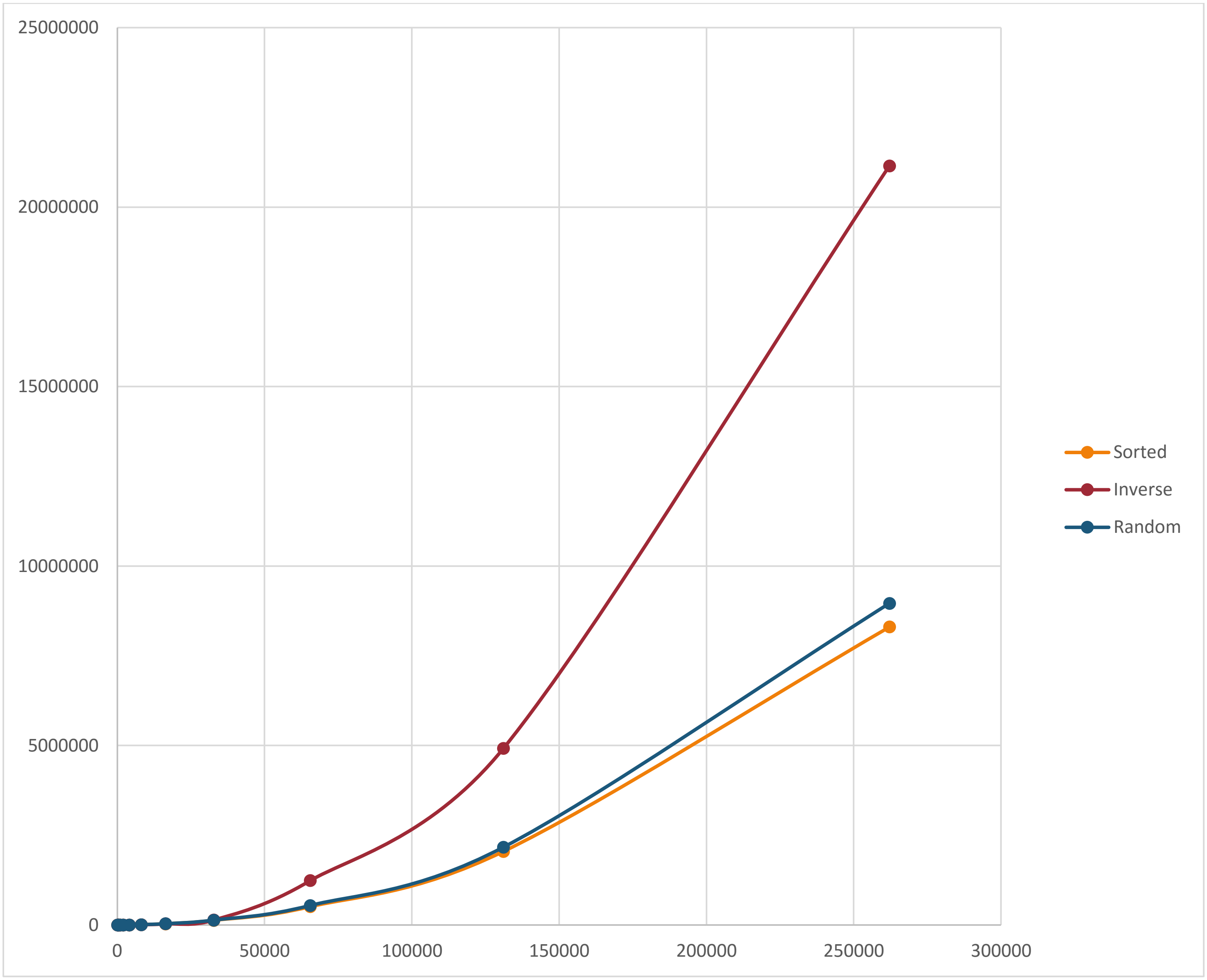
__Insertion algorithm:__



The insertion algorithm performs in the predicted way. Please note that some of the points are hidden under the "sorted" line and that the sorted vector results resulted in a linear function, while the inversely and randomly sorted results resulted in quadratic functions.

Having that into account, we can see how, as expected, the best case scenario, which is the sorted vector, performed the best among the three options. In fact, it has an O (n) while the others have an O (n^2) complexity. This is because, when the elements are all sorted, we never have to execute the second loop inside our function. Next in line was the randomly sorted vector. Finally, we have the worst case scenario, which means every single item in the vector needs to be reordered.

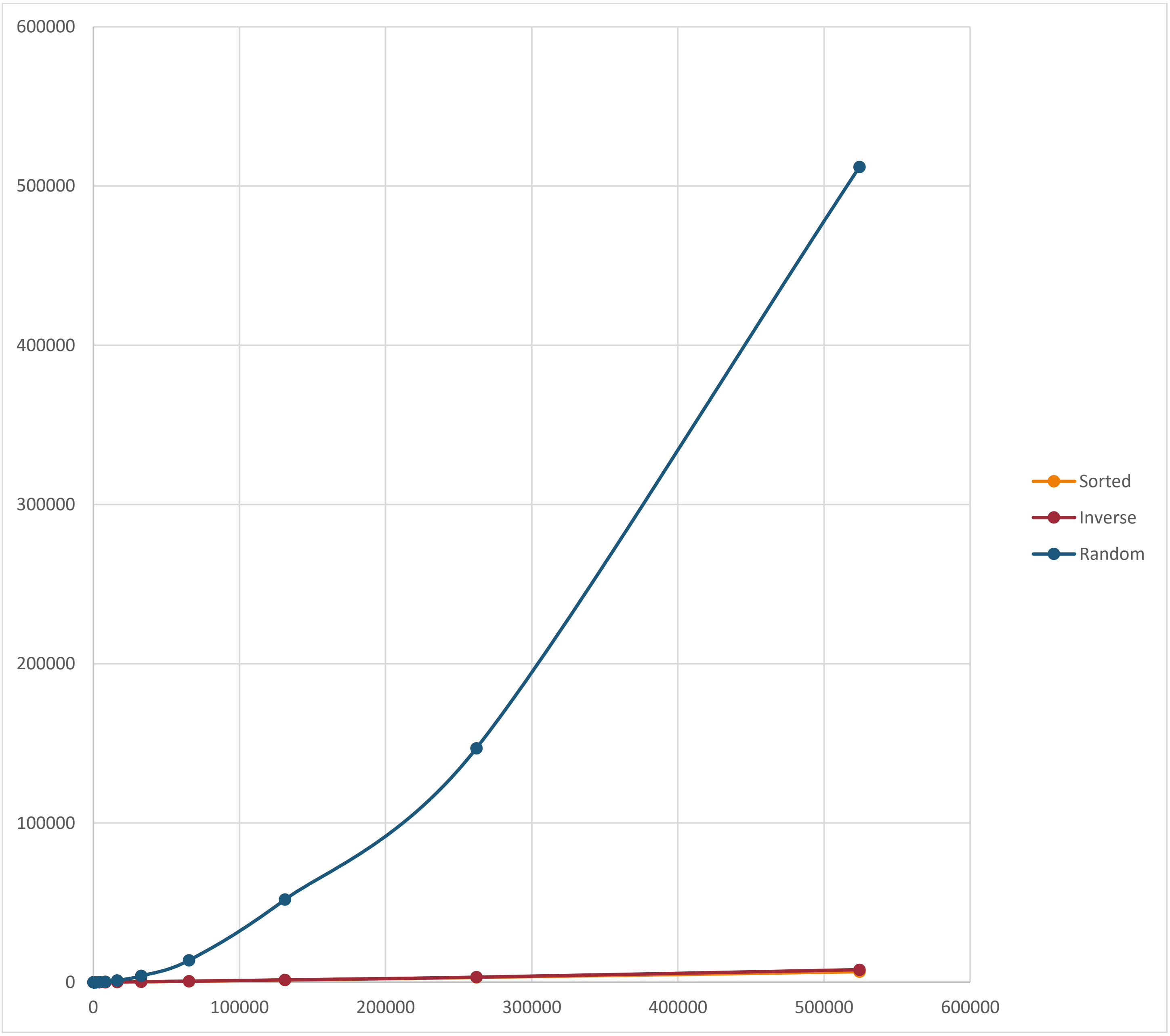
Selection algorithm:



The selection algorithm also behaved as predicted. We can appreciate that in every case, the complexity we reached was O (n^2). This is because, independently of the sorting of the vector, the algorithm needs to traverse it and call a findMinPos() method, which has an O (n) complexity. That means we will always perform O (n) * O (n) = O (n^2).

Apart from this, we appreciate that the worst case scenario performs way worse than the average and best case scenarios (which are pretty similar among themselves). One reason for this could be that the findMinPos() method needs to iterate first until the last positions to find the minimum elements when the vector is inversely sorted, thus consuming more time than it would if the minimum element were, for example, in the middle of the vector.

Quicksort algorithm:



For the quicksort algorithm we chose the central element as a pivot. In the graph we can see that the sorted and inversely sorted times are almost the same. This is because we are actually picking the same element both times! In this cases, we don't care that they are sorted ascendingly ("sorted") or descendingly ("inversely sorted"), the central value will be exactly the same.

As for the randomly sorted vector, picking the central element is no longer such a good idea. With certain combinations, you could end up with a pivot which required many exchanges and did not result in a tree, but in a list of numbers resulting in an O (n^2) complexity

Anyway, the overall times are much better than those of any other algorithm we studied. Thus, Quicksort would be the choice algorithm for any upcoming sorting task.

LAB 3 CONCLUSION:

All the algorithms behaved as studied, excepting the Bubble algorithm which somehow managed to perform better in the worst case scenario than in the average case scenario (inversely sorted vs randomly sorted).

The best performing algorithm was, as expected, Quicksort, even when choosing the central element as the pivot.