Assignment 4 Report - Sorting an array in Java

Dean Tsankov

September 22, 2024

Introduction

In this report I will present a couple of search algorithms, every one of which has its own positives and negatives. This in turn will make it important to know when to use each according to the particular use case. In this assignment all of the work will be done with arrays of a given size and there will not be any new elements added to the data set. For each of the algorithms I will explore the run time complexity as a function of the size of the array.

Selection sort - A simple and not so efficient sort algorithm

We begin with a simple algorithm that is not very efficient, but easy to implement, and works acceptably well with small array sizes. The general idea of the algorithm is to go through the array once and take the smallest value and put it at the beginning, next we go through the rest of the array (now without the element at the first index), find the smallest value for this sub-array and place it after the first element, shrink the sub-array where we search again and so on. Below is the provided method, filled in:

```
(...)
 public static void selectionSort(int[] array) {
    for (int i = 0; i < array.length - 1; i++) {
        int candidate = i;
        for (int j = i; j < array.length; j++) {
            if (array[j] < array[candidate]) {
                candidate = j;
            }
        }
        swap(array, i, candidate);
        }
    }
    (...)
```

I implemented a separate `swap` method, since it will also be useful in the next section.

```
    (...)
     private static void swap(int[] array, int i, int j) {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
    (...)
```

Here is a plot of the benchmarks ran on this sorting algorithm:

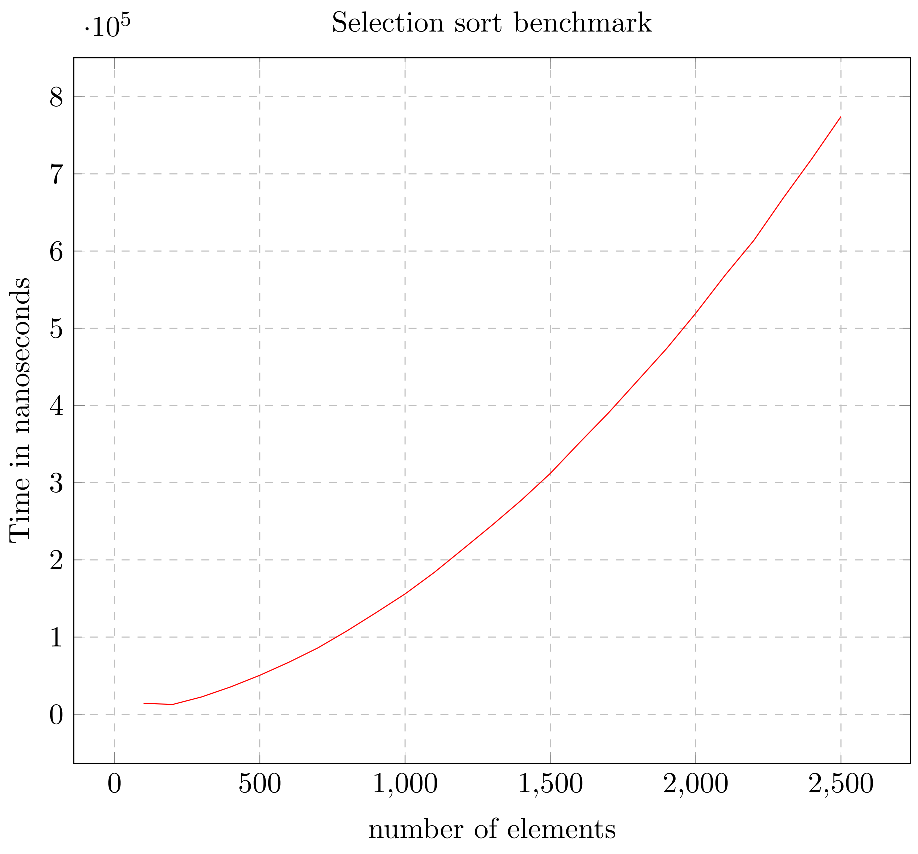

Figure 1: Sorting over randomly generated array

We can clearly see an exponential growth considering that both axes of the graph are linearly mapped and as such we would expect the time complexity with respect to the size of the sorted over array to be `O(`n^2`)`. And in fact if we think about how the algorithm works, we would come to the same conclusion, that is, if we consider the complexity as the area of our problem (since we have a nested `for` loop we can view it as a 2-dimensional area) both the width and height are functions of the size n.

Selection sort appears to perform very well if we want to quickly get, let us say, the first couple of smallest values from an array and are not interested in the rest of the data. It is, though, not a stable algorithm, meaning that if we sort data by multiple attributes, each time the previous sorting would be disregarded.

Insertion sort

The second algorithm we are taking a look at is in some aspects similar to the first. Here instead of selecting the smallest value, we insert the next one in the already sorted part of the array. For each value, when traversing the array, we compare it with the elements on the left side of the array, which we assume are already ordered. If the value is less we move it further back, if not we continue onto the next one in the array. The method is simple enough to implement and makes use of the previously declared `swap` function.

```
(...)
  public static void insertionSort(int[] array) {
    for (int i = 0; i < array.length; i++) {
        int point = i;
        for (int j = i - 1; ((j > -1) && (array[point] < array[j]));
                j--) {

            swap(array, point, j);
            point--;
        }
    }
}
(...)
```

Along with filling in the gaps of the given code I introduced a pointer index which represents the current of the sorted elements we are comparing with.

Next is this algorithm's time complexity graph, imposed onto the one for the selection sort.

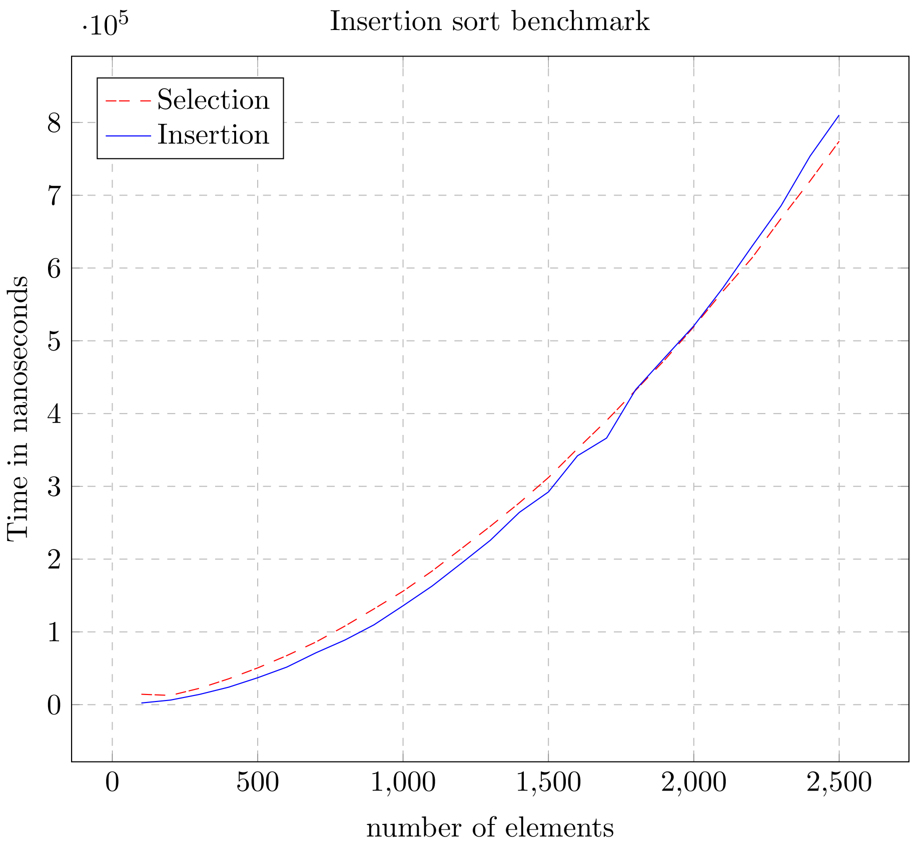

Figure 2: Sorting over randomly generated array

We see almost the same complexity with only slight variations. We can notice that as the problem size grows the insertion sort overtakes the selection sort gradually, meaning that it has better performance with larger n. The trade off of the two algorithms is that selection sort does more comparisons while insertion does more function calls (or generally just the swap operation). This would lead us to believe that comparison operations are generally more expensive than function calls (read/write actions) which seems to be the case.

Now insertion sort is in fact a stable algorithm, which is one advantage it has over selection sorting, but a con is the fact that it performs tremendously worse if the provided data is already sorted or semi-sorted.

Merge sort - Implementing using recursion

Finally we will explore merge sort. The difference here is that it will require an additional array, where temporary results will be stored, before producing the final sorted array. We will also implement the algorithm recursively, that is, there will be points in the algorithm where it will call upon itself. The idea behind it goes as follows, we take the array, split it in two parts, sort each part separately and then merge both the parts together in a way which preserves the order. But how do we sort the two split parts, we call upon the same function recursively and thus split the array into even smaller parts to solve. We do this until we are left with single elements, those elements when merged into pairs will be sorted and so when we go back up the recursive stack we will end up with a sorted array. We use an auxiliary array to store the separate sorted parts of the array before merging both in the final original one.

Bellow are the 3 functions the algorithm requires to work:

```
(...)
public static void mergeSort(int[] org) {
    if (org.length == 0)
        return;
    int[] aux = new int[org.length];
    mergeSortSort(org, aux, 0, org.length - 1);
}
(...)
```

First is the general function call which initiates the algorithm and passes the newly created auxiliary array.

```
(...)
private static void mergeSortSort(int[] org, int[] aux, int lo,
        int hi) {

    if (lo != hi) {
        int mid = (lo + hi) / 2;

        mergeSortSort(org, aux, lo, mid);

        mergeSortSort(org, aux, mid+1, hi);

        mergeSortMerge(org, aux, lo, mid, hi);
    }
    return;
}
(...)
```

The next method is the sorting logic i.e. the splitting of the array in depth (here the recursive call happens) and later calling the merging method.

```
(...)
private static void mergeSortMerge(int[] org, int[] aux, int lo,
        int mid, int hi) {

    for (int i = lo ; i <= hi ; i++) {
        aux[i] = org[i];
    }
    int i = lo;
    int j = mid+1;
    for ( int k = lo; k <= hi; k++) {
        if (i > mid) {
            org[k] = aux[j];
            j++;
        } else if (j > hi) {
            org[k] = aux[i];
            i++;
        } else if (aux[i] <= aux[j]) {
            org[k] = aux[i];
            i++;
        } else {
            org[k] = aux[j];
            j++;
        }
    }
}
(...)
```

Finally the merging operation of the values stored at the auxiliary array into the original one. The logic is very simple and we just keep track of the beginnings of the two sub-arrays, compare the current two values and place the smaller into the resulting array.

Below is a graphical plot of the merge sort compared to the previous two.

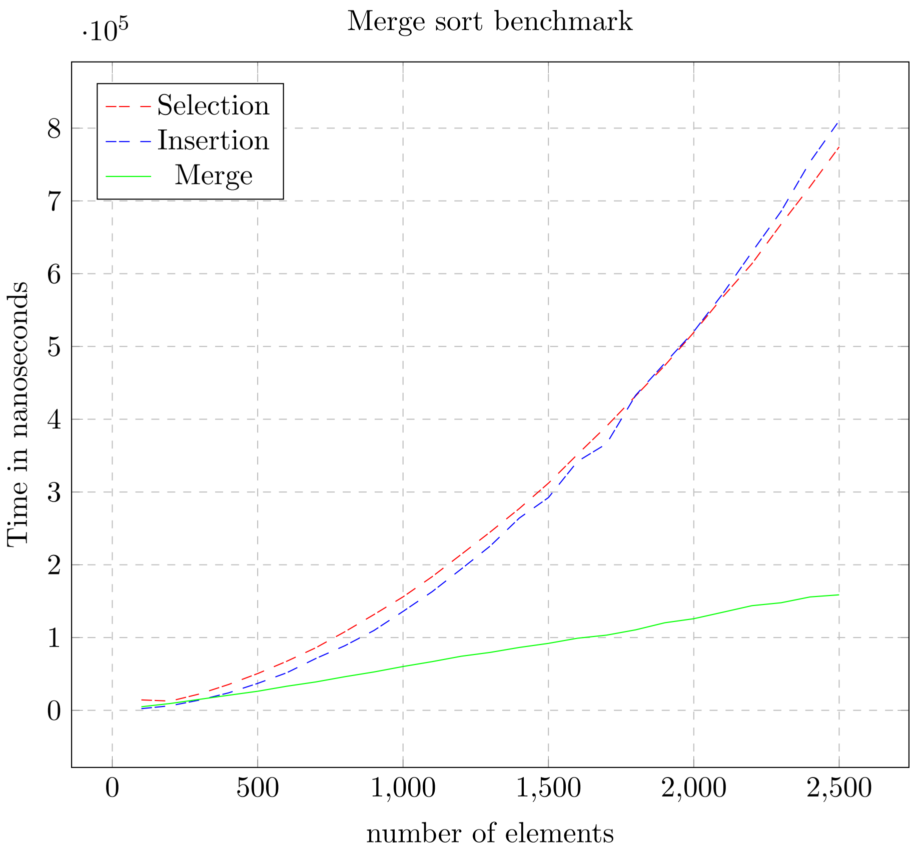

Figure 3: Sorting over randomly generated array

We can see a significant reduction in time complexity. Can be equated to `O(`$n * log(n)$`)` since for each level of the problem we are in essence splitting it in half and the complexity is taking the form of a reverse exponential function i.e. a logarithmic one. Later we have to merge all of the split parts which gives us the `n` term.

Taking a look at the application of merge sort, it might be not so well suited for smaller data sets. It is also worthwhile to consider that it requires an additional supporting array. Even still it is a stable and very efficient algorithm.

Let us try to optimize merge sorting even further

There is a very simple optimisation we can apply to our merge sort. It consists of simply moving the copying part of our merge function to the initial function call. This way we pass down two identical unsorted arrays to the `MergeSortSort()` method. Then inside it, during the recursive call we toggle passing the original and auxiliary arrays. What this does is that on each level of the recursion, we are working on a separate part of either the original or aux array. Then when it comes time to merge them, all the information is already in them so there is no need to transfer anything, and we can go on comparing values and merging as before.

Here is a comparison of the optimization we implemented:

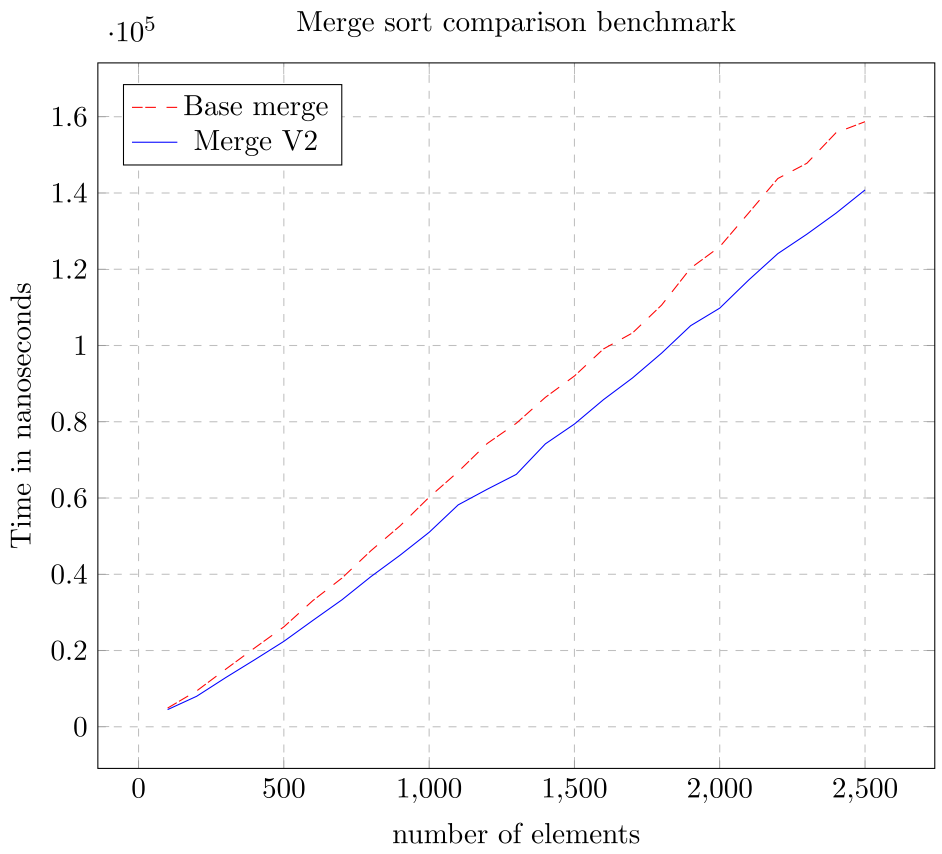

Figure 4: Sorting over randomly generated array

Now we definitely see there is an improvement in the execution, and if for small samples like the ones shown here the difference might seem not so significant, due to the fact that both algorithms retain their `O`$(n * log(n))$ complexity, this simple alteration could prove quite beneficial.