Runtime and Time Complexity of Sorting Algorithms on n Sized Arrays

Malte Berg

HT 2024

Introduction

This report covers the fourth assignment assigned in the course ID1021. The assignment is to analyze the runtime and time complexity for different sorting algorithms on arrays of different sizes. The three main sorting algorithms covered in the report are selection sort, insertion sort and merge sort. A comparison between the different sorting algorithms will be made, as well as explanations for how the different sorting algorithms function.

Selection Sort

The first sorting method is a simple one, which also results in it not being that efficient.

```java
private static void selection(int[] arr) {
    for (int i = 0; i < arr.length -1; i++) {
        int candidate = i;
        for (int j = i+1; j < arr.length ; j++) {
            if (arr[j] < arr[candidate]) candidate = j;
        }
        int temp = arr[i];
        arr[i] = arr[candidate];
        arr[candidate] = temp;
    }
}
```

Selection sort starts with an index of the first element in a given array, `candidate`. It then loops through the array from position i+1 onward to see if there is an integer smaller than the first one. If found, the index of that integer is saved to `candidate`. The integers are then swapped and the loop starts over with the candidate becoming the second element, and so on.

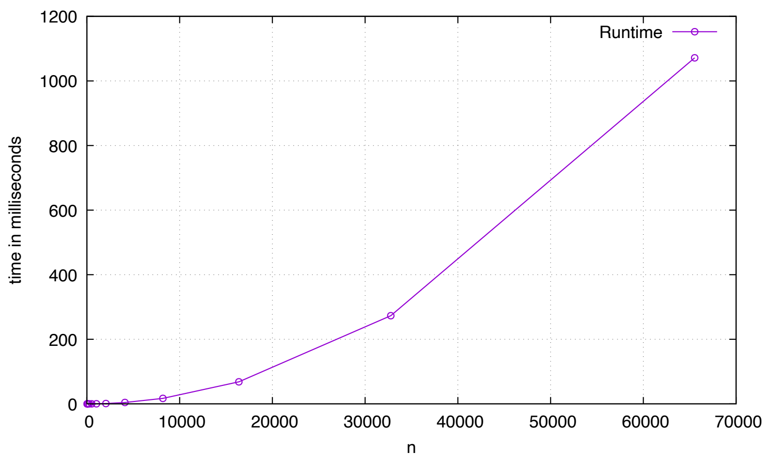


Figure 1: Runtime of selection sort on n sized arrays, in milliseconds

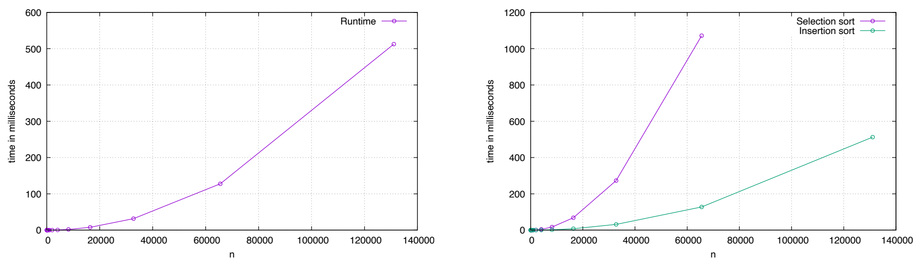
As can be seen in Figure 1, the runtime increase is almost four times as large as the array size doubles. From this, a conclusion can be found that the time complexity for the Selection sort comes out to $O(n^2)$. The reason the sorting algorithm is not so efficient is due to the fact that it will always look through every element on every loop, and there is no way that it will exit early out of the method as it well never see that it might even be finished sorting. Selection sort performs exactly n swaps.

Insertion Sort

A more efficient sorting algorithm is Insertion sort. It can take use of the part of the array it has already sorted, as well as benefiting from partially sorted arrays.

```
private static boolean insertion(int[] arr) {
    for (int i = 0; i < arr.length; i++) {
        for (int j = i; j > 0 && arr[j] < arr[j-1]; j--) {
            swap(arr, j, j-1);
        }
    }
    return true;
}
```

Insertion sort works by "inserting" the integers to the already sorted start. When a smaller integer is found before the integer being sorted it places the integer right after, creating a sorted order. Compared to selection sort, it's not guaranteed the amount of swaps that will be done as the



(a) Insertion sort (b) Insertion and Selection sort compared

Figure 2: Runtime of selection and insertion sort on n sized arrays, in milliseconds

algorithm only swaps numbers when necessary, as well as not having on average as many comparisons as selection sort.

Looking at Figure 2 (a), it might seem very similar to Figure 1, but plotting the two against one another as seen in Figure 2 (b), it is clear that insertion sort performs much better, and could also be tested with larger n.

While it performs much better, its time complexity, just like selection sort, is $O(n^2)$. Although, it can run in $O(n)$ time if the passed array is already sorted.

Merge Sort

The last of the three search algorithms is merge sort, saving the most efficient and most complicated for last. The solved code for the algorithm is as follows:

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

private static void sort(int[] org, int[] aux, int lo, int hi) {
    if (lo != hi) {
        int mid = (lo + hi)/2;
        sort(org, aux, lo, mid);
        sort(org, aux, mid+1, hi);
        merge(org, aux, lo, mid, hi);
    }
}
private static void merge(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++];
            else if (j > hi) org[k] = aux[i++];
            else if (aux[i] <= aux[j]) org[k] = aux[i++];
            else org[k] = aux[j++];
        }
    }
```

The implemented merge sort algorithm uses recursive calls to continuously ”split” and sort smaller and smaller arrays to then merge them together after they are sorted to finally join the full array together fully sorted.

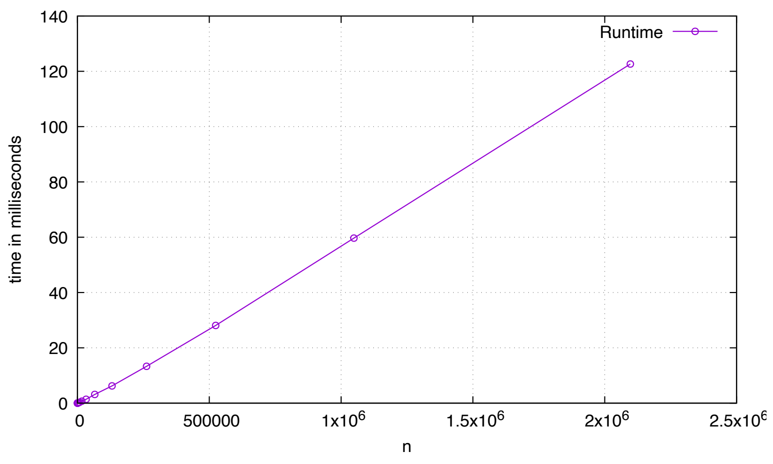


Figure 3: Runtime of merge sort on n sized arrays, in milliseconds

As seen in Figure 3, an almost linear graph is created as the array size n increases. However, it is not really linear, or $O(n)$. Rather, it has the time complexity $O(n \log(n))$ which is almost identical to a linear graph for smaller numbers.

A modified version was made to the merge sort algorithm with the help of AI to see what the effect of not using the `aux` array too heavily would have on the runtime, but there were not too big of an improvement.

Comparison

As can be seen in Figure 4, the search algorithms ranks from least to most efficient like the sections in the report with merge sort being the most efficient of them all.

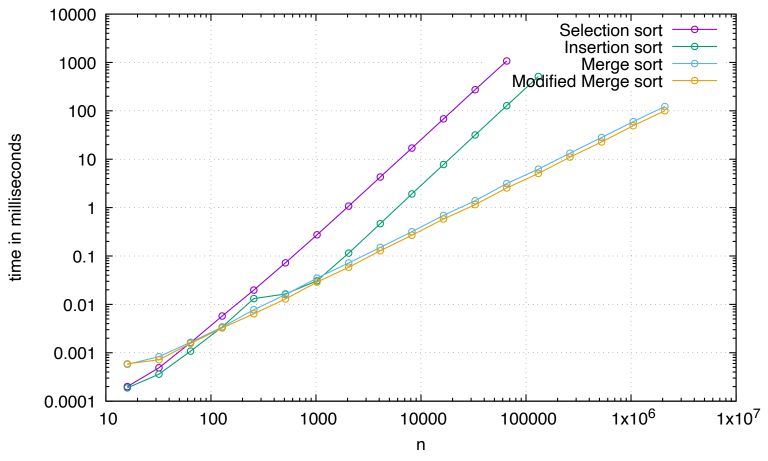


Figure 4: Runtime of all algorithms on n sized arrays, in milliseconds (logarithmic axles)