Report for 1DV516
Programming Assignment 3
Algorithms and Advanced Data Structures

Ingmar Immanuel Falk

October 27, 2023

Contents

1 Introduction

2 G

2.1 Problem 2 - Vehicle

For problem 2 I created a *Vehicle* class that contains the year the vehicle was made, the color of the vehicle, the make of the vehicle and the plate of the vehicle. The plate contains 3 letters, followed by 2 digits followed by either a digit or a string (according to the latest regulations in Sweden: Vehicle registration plates of Sweden).

I overrode the *hashCode* method of the *Vehicle* class with the following code:

```
@Override
public int hashCode() {
    return Math.abs (
        year
        + color.hashCode()
        + make.hashCode()
        + plate.hashCode()
    );
}
```

This is a very simple hash function that just adds the hash codes of the different fields of the *Vehicle* class together.

To test this function, I conducted an experiment where I created 50.000 vehicles with random values for each field. For example, the year is a random number from 1920-present and the make is a random choice of a list of predefined brands. When inserting the new vehicle, I keep track of the original destination, the actual index where the element ended up as well as the number of collisions on this index. The results for inserting 10.000 vehicles can be seen below.

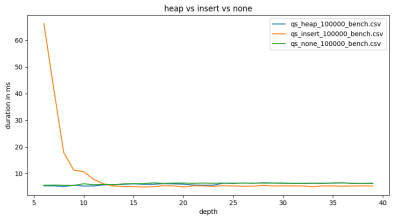
Average Offset: 357,
Number of Collisions per Index: 0.431,
Number of Buckets: 73.597,
Load Factor: 0.68

The load factor is ideal and is almost the same as the load factor that we set as a destination, which means the number of buckets is very good.

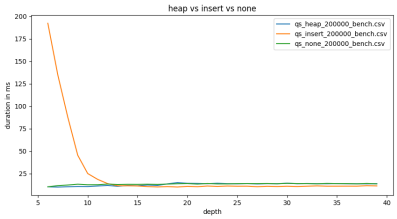
The same goes for the number of collisions, which are relatively low with an average of 0.431 collisions per index. This indicates a disparity. While many elements are inserted very smoothly, there are a couple of insertions whose probing sequence is very long. This is most likely due to clustering, which is caused by using quadratic probing.

2.2 Problem 5 - Quick Sort

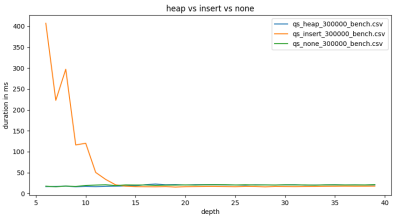
The experiment I conducted consisted of creating an array of random integers of size n. The sizes range from 10.000 to 200.000 in steps of 10.000. For each size, I test the time it takes to sort the array using different max depths. Those range from 0 to 50. In the end, after I wrote all the results to a file I extracted the lowest time and the corresponding depth with it. This is done for both the insertsort and the heapsort version of the algorithm and gives us the result displayed in the figure below.



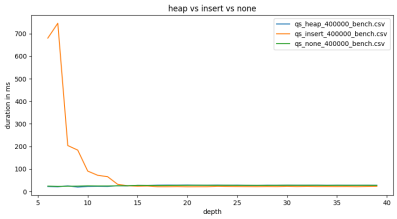
(a) Size 100.000



(b) Size 200.000



(c) Size 300.000



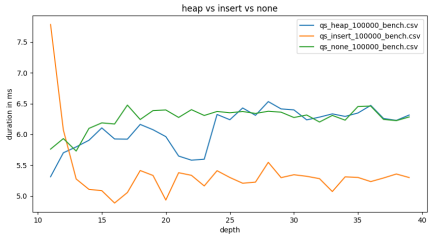
(d) Size 400.000

Figure 1: The time it takes to sort an array of size n using quicksort with different max depth. The x axis is the max depth (from depth 5 onwards) and the y axis is the time in milliseconds.

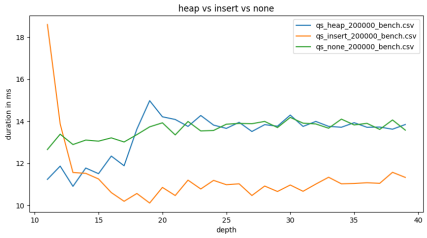
These figures clearly show that in the beginning, the insert-sort version is *significantly* slower than the heap-sort version. This is due to the fact that in the beginning, for low max recursion depths, we are sorting $large, unordered$ subarrays. This can easily trigger the worst-case scenario of insert-sort, whereas the heap-sort version will always be $O(n \log n)$, no matter the input. However, as the max depth increases, the heap-sort version becomes slower, while the insert-sort version becomes faster. This is caused by several factors. Insert sort can be faster for smaller arrays/subarrays, as it has

a lower constant factor than heap-sort. Furthermore, as the depth increases, we start dealing with more partially sorted subarrays, which is where insert sort shines due to its linear nature.

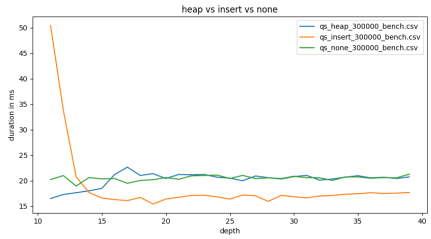
To determine the optimal threshold to switch from heap-sort to insert-sort, let's look at the same graphs again, but this time we zoom in a bit to the area where the heap-sort version starts to become slower than the insert-sort version.



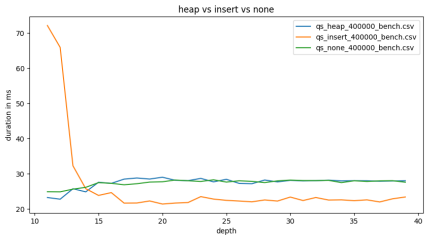
(a) Size 100.000



(b) Size 200.000



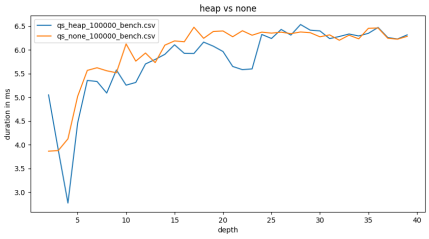
(c) Size 300.000



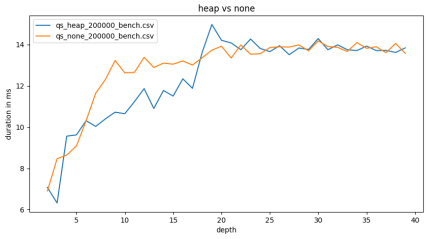
(d) Size 400.000

Figure 2: The time it takes to sort an array of size n using quicksort with different max depth. The x axis is the max depth (from depth 10 onwards) and the y axis is the time in milliseconds.

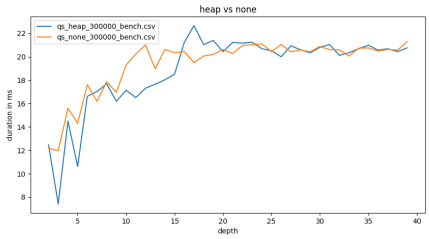
This time we can clearly see that the heap-sort version starts to become slower than the insert-sort version at around depth 12-15. This is where the partially sorted subarrays start to become more common, and the simple insert-sort algorithm can take advantage of this. However, we can also see that the heap-sort version is still faster than the insert-sort version for the first 12-15 depths. But not only is the heap-sort version faster than the insert-sort one, it is also faster than the version using no fallback at all. This can be seen in the next figure.



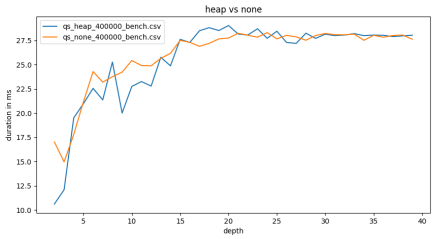
(a) Size 100.000



(b) Size 200.000



(c) Size 300.000



(d) Size 400.000

Figure 3: The time it takes to sort an array of size n using quicksort with different max depth. The x axis is the max depth (from depth 10 onwards) and the y axis is the time in milliseconds.

As can be seen, the speed difference is quite insignificant, but it is still there for depths until around 15-20, after that both implementations perform almost the same.

2.2.1 Conclusion

For depths up to 12-15, heap-sort should be used as a fallback, as it is faster than insert-sort for those depths. After that, insert-sort should be used for all depths after following, as there is a significant speed increase for those depths.

3 VG

3.1 Problem 6 - Merge Sort

For merge-sort, I created a simple benchmark that creates an array of random integers and then sorts it using merge-sort. One version uses the recursive version, the other one obviously then the iterative one. I do this for different array sizes, ranging from 1.000 to 1.000.000.

The results can be seen in the figure below.

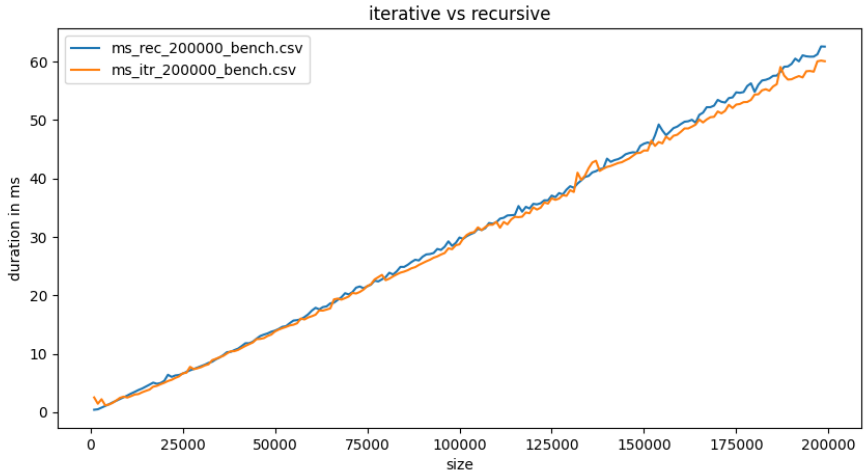
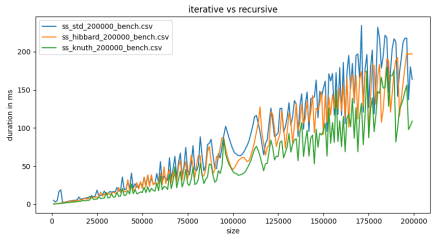


Figure 4: The time it takes to sort an array of size n using merge-sort with different max depth. The x axis is the max depth (from depth 10 onwards) and the y axis is the time in milliseconds.

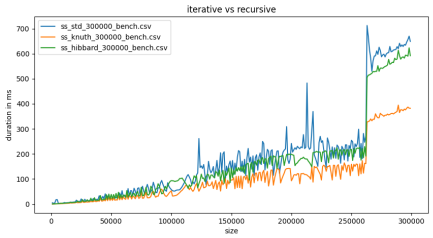
As can be seen, on average, the iterative version is minimally faster than the recursive version for all array sizes. This is due to the fact that the iterative version does not have to deal with the overhead of recursive calls, which can be quite expensive.

3.2 Problem 7 - Shell Sort

For this test, I once again created a benchmark consisting of an array of 200.000/300.000 random integers. I then sort this array using the standard shell sort algorithm, the Knuth sequence and the Hibbard sequence. The results can be seen in the figures below.



(a) Size 200.000



(b) Size 300.000

The standard sequence is quite simple with this formula: $\frac{n}{2^k}$, where k is the current iteration. The Knuth sequence is a bit more complicated and is defined as $h_k = 1 + 3^k + 2^{k-1}$, where k is the current iteration. The Hibbard sequence is defined as $h_k = 2^k - 1$, where k is the current iteration.

As can be seen in the figures above, the standard sequence is the slowest of the three, while the Knuth sequence is the fastest. This is due to the fact that both the Hibbard sequence and the Knuth are less likely to have repeating patterns that are hard to break, which is a common problem with the standard sequence. The Hibbard sequence, even tho it has the same worst-case time complexity of $O(n^{\frac{3}{2}})$ as the Knuth sequence, is still slower than the Knuth sequence.