HW#4 Report

COMP202 Spring 2022

Ameer Taweel - 0077340

Manual Questions

In this section, I will answer the three questions presented in the manual. After that, I will show my findings for the code part.

Question 1

HeapSort is an **in-place** algorithm. First, we heapify the input array. Then we split the array to two parts, a sorted part, and the remaining heap part. We keep extracting the minimum (or the maximum, depending on whether the sort is in ascending or descending order) until we have the whole array sorted.

However, in my implementation for the homework, I did not use the in-place approach. Because I would've had to implement a heap myself. I used Java's `PriorityQueue` for my implementation. While this causes a performance drop, it makes the code cleaner and less error-prone.

Question 2

HeapSort is $\Theta(N \cdot log(N))$, so the worst case is $N \cdot log(N)$, but the best case is also $N \cdot log(N)$. We should get the same performance even if we have a presorted.

However, InsertionSort runs in $O(N)$ for presorted input.

Question 3

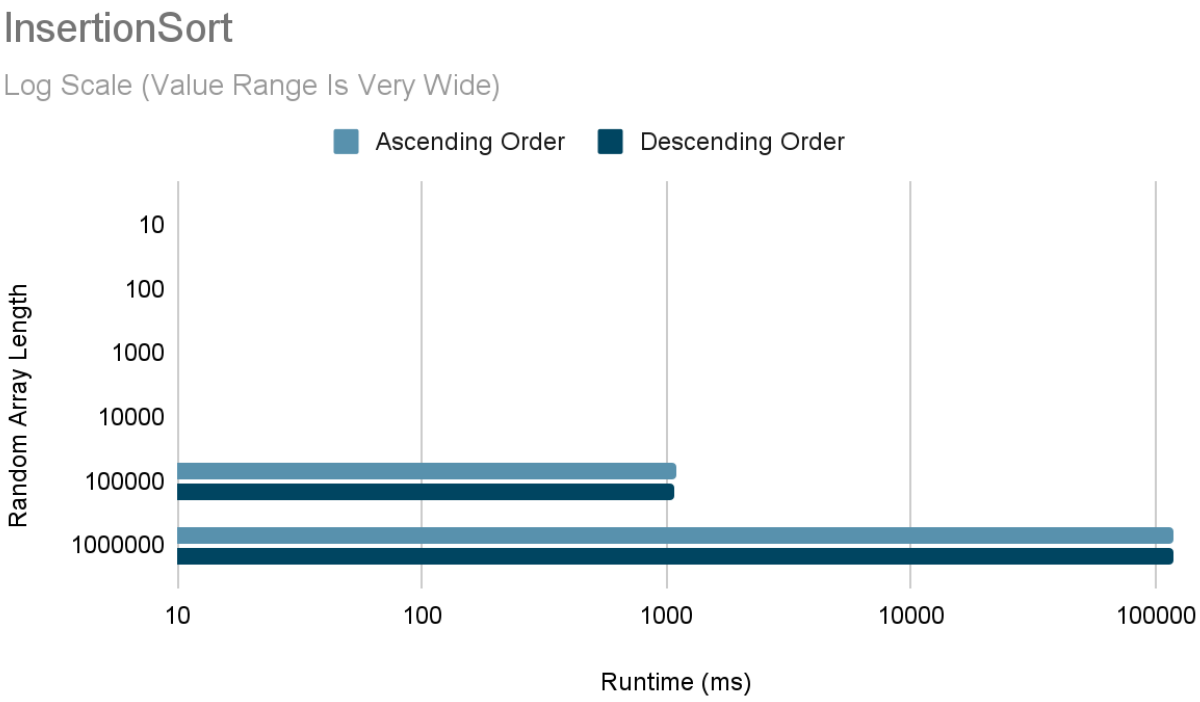
Any presorted input will do the job. For example: $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$. HeapSort will still take $O(N \cdot log(N))$, while InsertionSort will take $O(N)$.

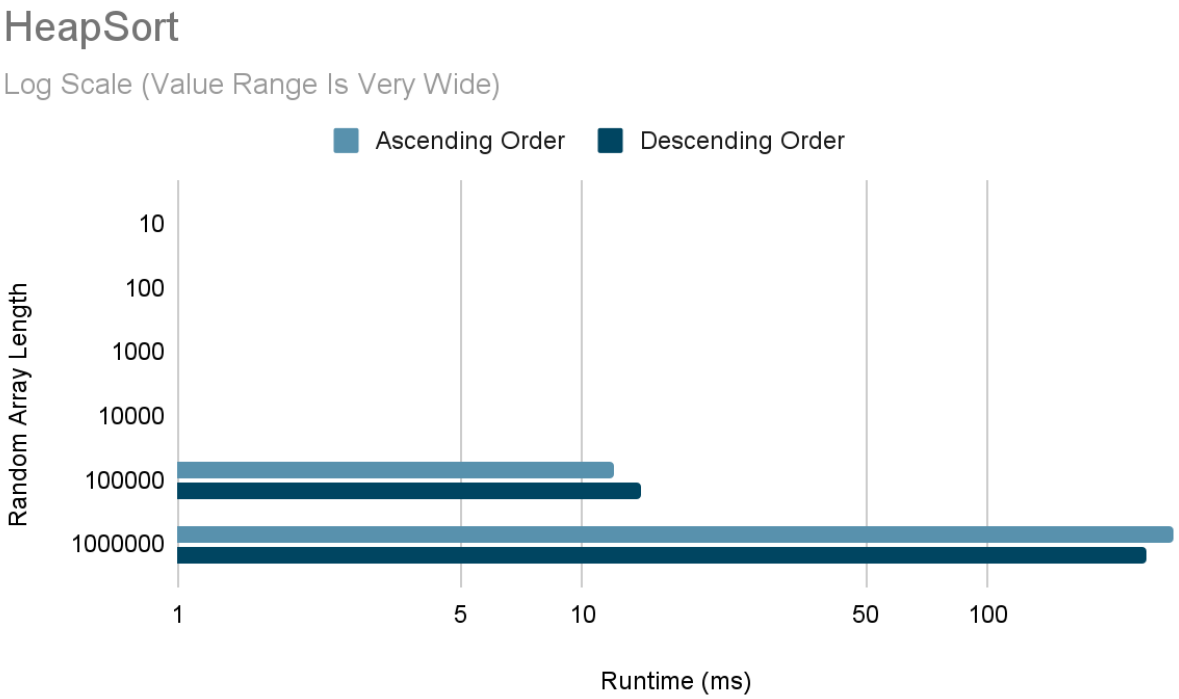
However, if we reverse the list, HeapSort will still take $O(N \cdot log(N))$, but InsertionSort will take $O(N^2)$.

Experiments and Findings

Experiment #001: Comparing Directional Performance

A sorting algorithm should give the same performance when it sorts in ascending or descending order. I will try to confirm this by empirical data.



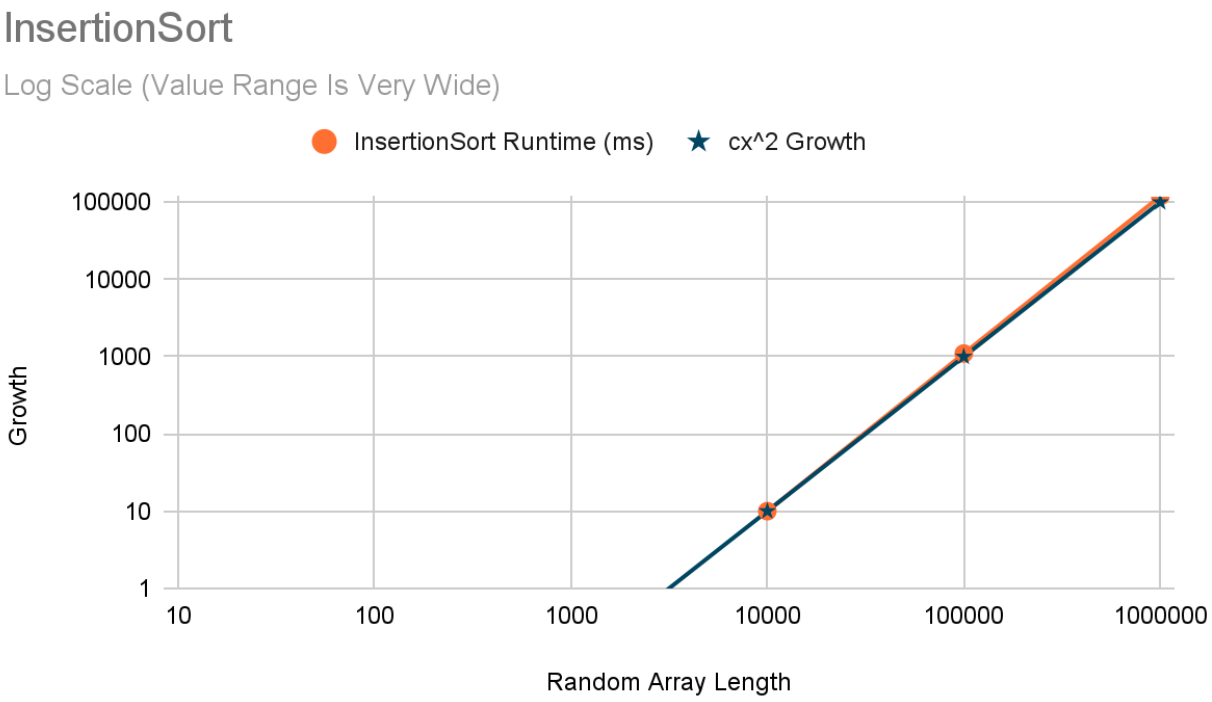


NOTE: I used the log scale for the charts of this experiment. Because the value's range is very wide.

We can clearly see from the previous results, that both InsertionSort and HeapSort give the same performance for ascending and descending order.

Experiment #002: InsertionSort Complexity

InsertionSort has two nested loops that go through all the elements of the input array in the worst case, so it has a complexity of $O(N^2)$. We want to confirm this by empirical data.



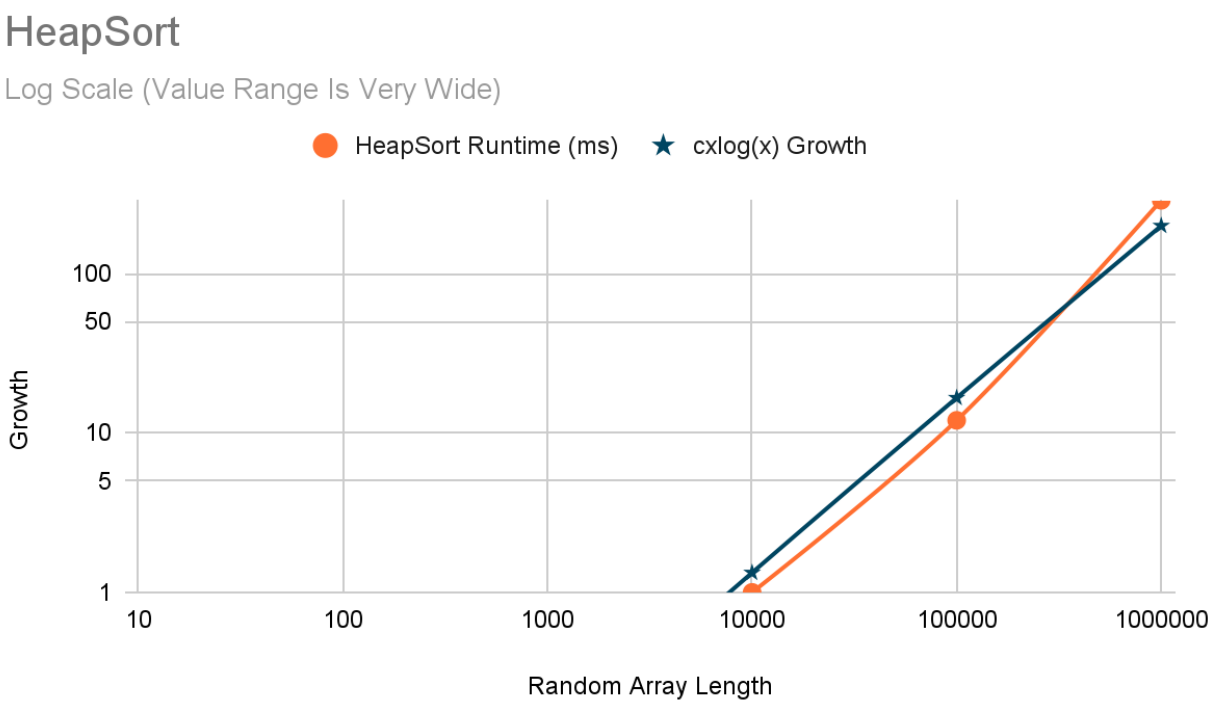
NOTE: I used the log scale for the chart of this experiment. Because the value's range is very wide.

I compared the growth of runtime for InsertionSort with the growth of cx^2, where $c = 0.0000001$. The result is almost two identical lines, which confirms that InsertionSort has a complexity of $O(N^2)$.

Experiment #003: HeapSort Complexity

My implementation of HeapSort inserts the elements of the input array to a `PriorityQueue`, so we have N insertions, each with a cost of $log(N)$. Then it extracts the minimum (or maximum if it's sorting in descending order) element from the `PriorityQueue` until it extracts all the elements. So we have N removals, each with a cost of $log(N)$. So, the overall complexity is $O(N \cdot log(N))$. We want to confirm this by empirical data.

NOTE: The in-place implementation is more performant, but it also has a complexity of $O(N \cdot log(N))$.

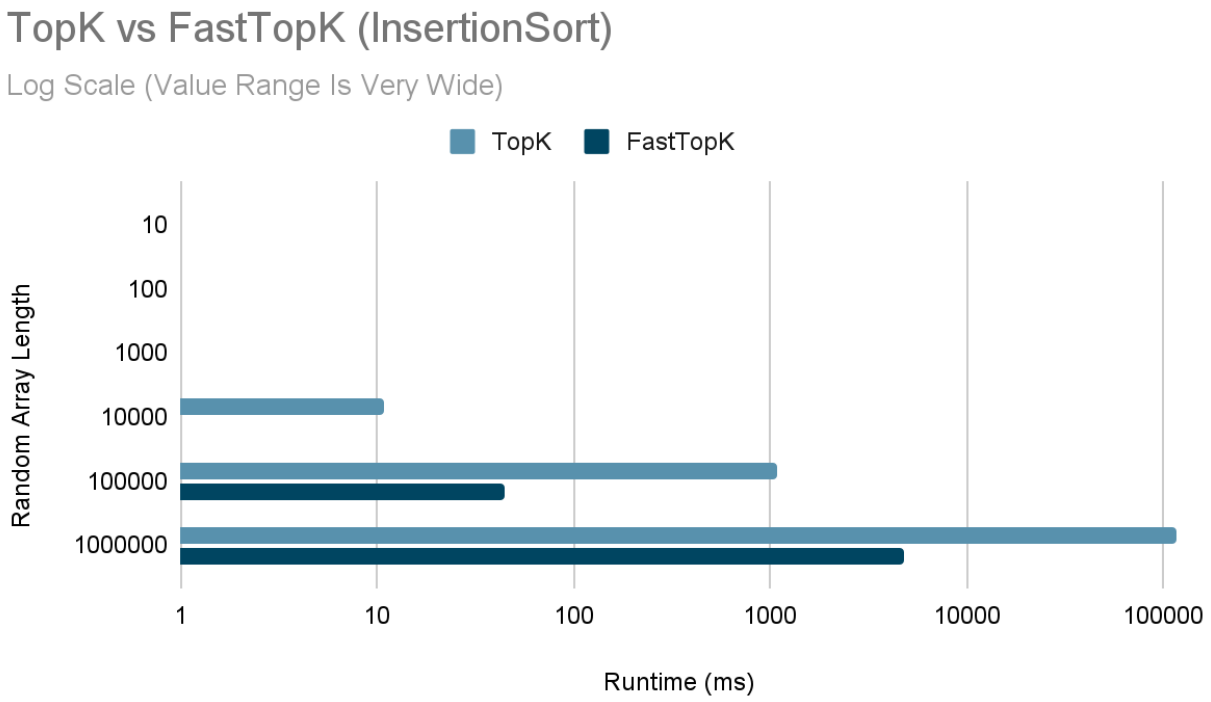


NOTE: I used the log scale for the chart of this experiment. Because the value's range is very wide.

I compared the growth of runtime for HeapSort with the growth of $cxlog_2(x)$, where $c = 0.00001$. The resulting two lines are very similar, which confirms that HeapSort has a complexity of $O(N \cdot log(N))$.

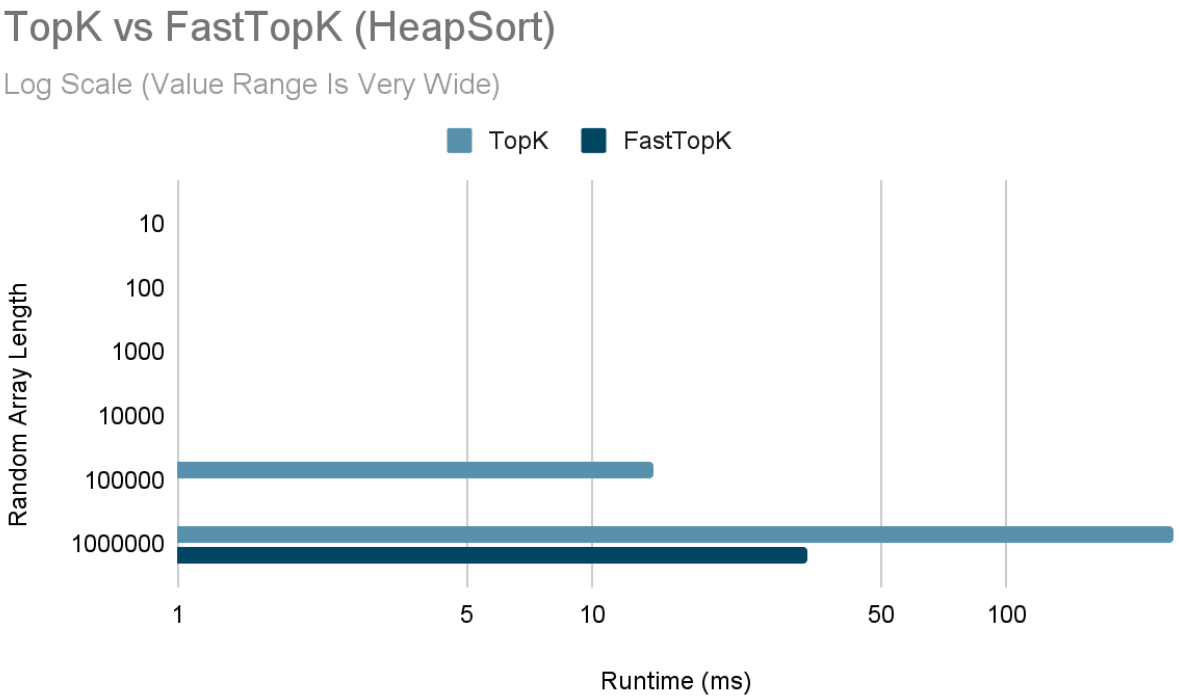
Experiment #004: Comparing TopK and FastTopK

My implementation of the FastTopK algorithm discards all the elements higher (or lower if it's sorting in descending order) than the mean of all values. In both the uniform and the normal distribution, this allows it to discard about half the values, which causes significant performance gain. I will compare the performance of the TopK and FastTopK algorithms using both InsertionSort and HeapSort.



NOTE: I used the log scale for the chart above. Because the value's range is very wide.

When using InsertionSort, FastTopK is around 25 times faster than TopK. This is a huge performance gain.



NOTE: I used the log scale for the chart above. Because the value's range is very wide.

When using HeapSort, FastTopK is between 7 to 14 times faster than TopK. Again, this is a huge performance gain.

Appendix A: All Test Data

I tested the code in 144 different combinations. The result of each combination is the average of 10 separate tests, to make the results as accurate as possible. This means that I performed 1440 sorting operations to write this report. Running all the tests took more than 3 hours.

You can find the data in the GitHub repository, in a file named `data.txt`.

Appendix B: Test Code

The code I used to generate all the data is in `Main.java`. The main test function is called `runTests()`, but there are several other helper functions called within it.