Hybrid Sorting

Hypothesis

As we noticed from the previous experiment that below 32, it is always more efficient to do insertion sort and above 34 , to do merge sort. I think that the optimal value of K should be nearly in the range 32-34 because then the new algorithm is choosing the fastest sorting algorithm at each stage and that should provide optimal speed. Also, I think the TimSort algorithm will perform better than each of the two sorting algorithms MergeSort and InsertionSort.

Methods

I choose the range of N as 1-50 both inclusive similar to the previous experiment.

For every value of N, I generate an array of random integers in the range (0,1000000). I conduct this experiment 1000 times and take the average over these 1000 experiments to get a more accurate average of the runtime.

For each value of N and the randomly sampled array, I run timsort for K=0,5,10,15,20,25,30,35,40,45,50 and record the running time of Timsort.

Finally, I plot the observations using the matplotlib library of python.

After, identifying the optimal value of k from the graph plotted above, I compare the running time of Insertion Sort, Merge Sort and TimSort over different array sizes.

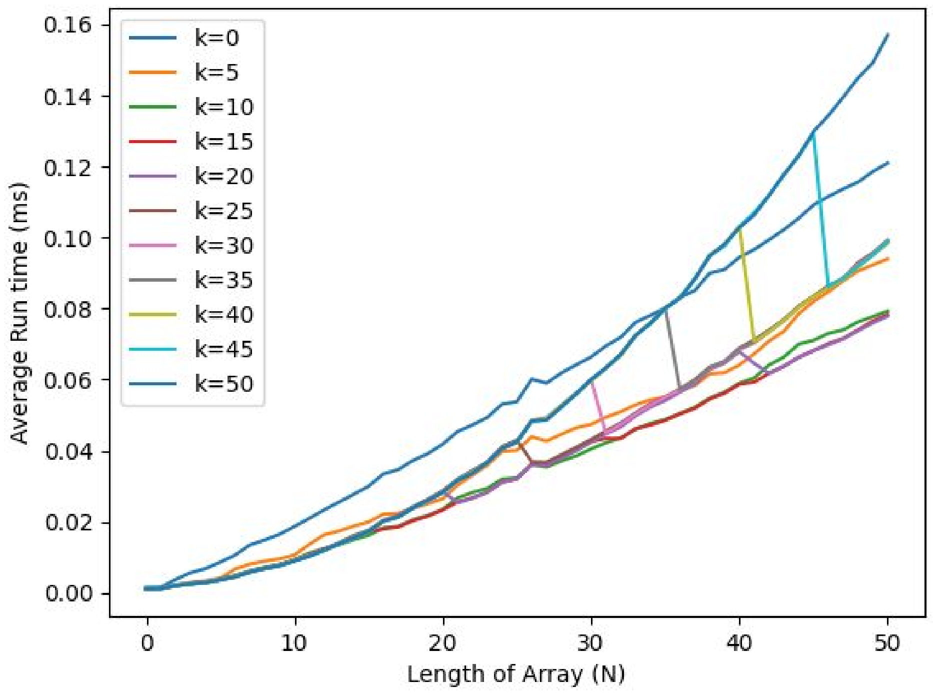
For this experiment as well, I choose the range of N as 1-50 both inclusive and generate an array of random integers in the range of (0,1000000). I conduct the experiment a 1000 times for each value of N and take the average running time as an accurate approximate for the running time of that algorithm at that N.

For measuring the time elapsed I use the perf_counter() function in the time library in python.

The code is available at
https://github.com/divyanshaggarwal/CSE830/blob/main/HW4/Ques2/hw4_ques2.py

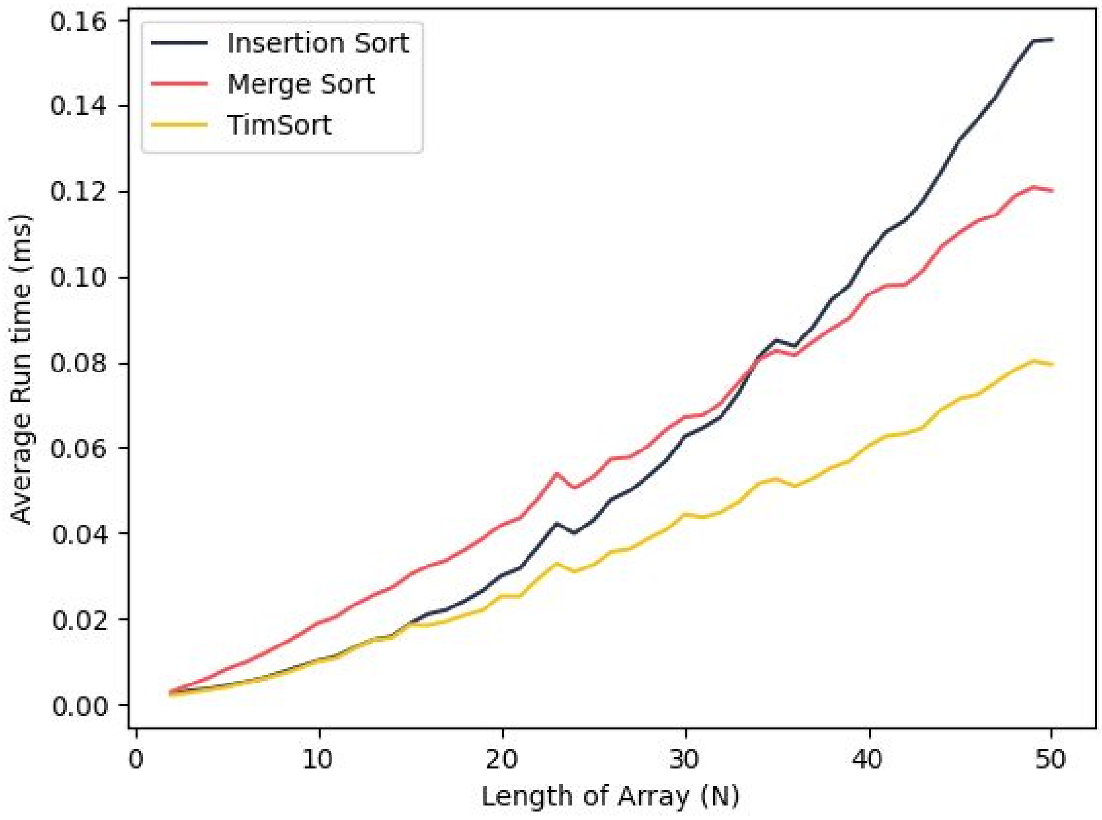
Results

The plot for the run-time of Timsort with different values of K over different values of N are shown below. The x-axis represents the value of N. The y-axis represents the average run-time of Timsort over the 1000 runs at each value of N in ms. The different lines represent the different values of K.



From the above plot we observe that the value of K that leads to lowest run-time is K=15. And therefore, I use K=15 for further experiments.

Comparison of runtime complexities of InsertionSort, MergeSort and TimSort over different values of N are plotted as below. The x-axis represents the different values of N and the y-axis represents the average runtime of the three different sorting algorithms over 1000 runs for each value of N in ms.



Discussion

From the first experiment, I observed that the optimal value of K was 15 which is much lower than the cross-over value in the first experiment. This was quite contrary to my hypothesis that these two values should be the same. However, when I thought about it, I realised that the runtime of InsertionSort increases exponentially with increasing N and therefore, even if the array is small enough for insertion sort to be more efficient than MergeSort, it could be even more beneficial to break the array into halves and apply insertion sorts to both of them. This will potentially lead to a smaller optimum value of K.

For example, if we have an array of size say 30. The crossover point suggests that we should apply insertion sort to this array which would probably give us a runtime of $O(n^2) = 30^2 = 900$. However, if we break this into two arrays as suggested by the optimum value K=15. We could potentially reduce the runtime to $O(2*(n/2)^2 + n)$ {i.e apply insertion sort to each of the two halves and then merge them} = $15^2 + 15^2 + 30 = 480$ which is much less than 900.

This basically explains why the optimum value of K is lower than the cross-over value. In short, choosing the most optimal sorting algorithm at current value of N may not lead to a global optimum sorting procedure.

One key thing that I also observe from the first plot is that at N=K+1 we see a sharp drop in the runtime of Timsort. This is basically because for N<=K, Timsort essentially behaves like InsertionSort but after N=K, the runtime significantly drops as the idea of TimSort comes into being. This also happens at every 2 power times K because at such values of N, an additional MergeSort step is added in the sorting procedure and thus provides a marginal improvement in the runtime over the last value of N.

From the second plot we observe that as we expected, Timsort performs exactly the same as InsertionSort until N=Optimal K(15) and thereafter continually provides improvement over both MergeSort and InsertionSort. For larger values of N it combines the ability of MergeSort to be efficient at larger values of N and InsertionSort’s ability to be faster for smaller values of N to provide a much faster sorting procedure overall.

Conclusion

From the experiments, I conclude that the optimal value of K is 15 which is much lower than the crossover point (32-34) we achieved in the previous experiment.

Moreover, TimSort performs the same as InsertionSort for N<=K and for N>K it combines the power of both MergeSort and InsertionSort to give a much more efficient sorting procedure than each of the individual sorting algorithms.