Comparing sorting algorithms

Merge sort, Heap sort, Quick sort

Hrachya Harutyunyan

1 What we did in this project

In this project we investigated problem related sorting algorithms .

We will compare together merge sort, quick sort and heap sort. In the sub chapters I will explain how I implemented them and the difficulties which I faced during it implementation.

1.1 Merge sort

In this algorithm we divide the array into two parts until the size will be 1 and after we should repeatedly merge into sorted ones. Swapping is taking place when the elements are in wrong place. Now let's talk about attributes of the algorithm . Merge sort is in place algorithm because it does not allocate extra memory for merge operation. Merge sort is stable because elements with the same values are in the place after sorting. And finally Merge Sort is non adaptive because it does not change his behavior during his execution.

Best ,Average and worst cases for this algorithm are O(NlogN)

1.2 Heap sort

To implement Heap sort at first we should make array to be in heap structure. That's why I implemented heap data structure which's constructor is taking array with pointer and its size .After in constructor I am calling fill_heap()(which is filling the heap but it

is not heap structure still) to make it look like heap I am calling build_heap function(which is making it to look like heap).After having array in heap structure we are ready to do implement heap sort . In my sorting algorithm in each iteration I am taking maximum element with extract max function and putting it in the back of sorted array until all array will filled .

Now let's talk about attributes of heap sort it in place because it is taking elements from input array and moving it outside and after putting back . It is non stable because during heap operations it can change places of the elements with the same values . It is non adaptive because it doesn't change his behavior during execution.

Best ,Average and worst cases for this algorithm are O(NlogN)

1.3 Quick sort

The main issue of this algorithm to choose correct pivot . there are several ways to choose pivot for example chose first element or in the end , also we can choose median of the elements but for my implementation I chose partitioning which is similar to Hoare's partitioning but I am not explicitly declaring the pivot . In my partitioning I use two indices one in the beginning second in the end, then we move this indices toward to each other , until they detect an inversion: (pairs which one of them is greater than the other and they are in wrong order). After the elements are getting swapped . If the indices are the same iteration stops and the index is returned. In my partition I do not explicitly define the pivot but if we look carefully we can see that it is the element which is located in the right, because out iteration continues until the right element is in the correct position. After getting the index we should do recursively quick sort until the element in index of our returned index -1 and from index+1 until the end . In the end we will get our sorted array .

Now let's talk about algorithm attributes it is in place because it didn't create new array for his operations. It is non stable because it changes positions of elements with same values . it is non adaptive because it is not changing his behavior.

Best,Average case is O(nlogn)
Worst case is $O(n^2)$ and it happens when array has same elements or it is sorted or reverse sorted.

P.S. During Quick sort benchmarking I faced difficulties because of Stack overflow because we know that quicksort is calling a lot of recursion so stack is getting full . So I increased the size of Stack from project properties to get result of more bigger arrays.

2.Methodology for comparing algorithms

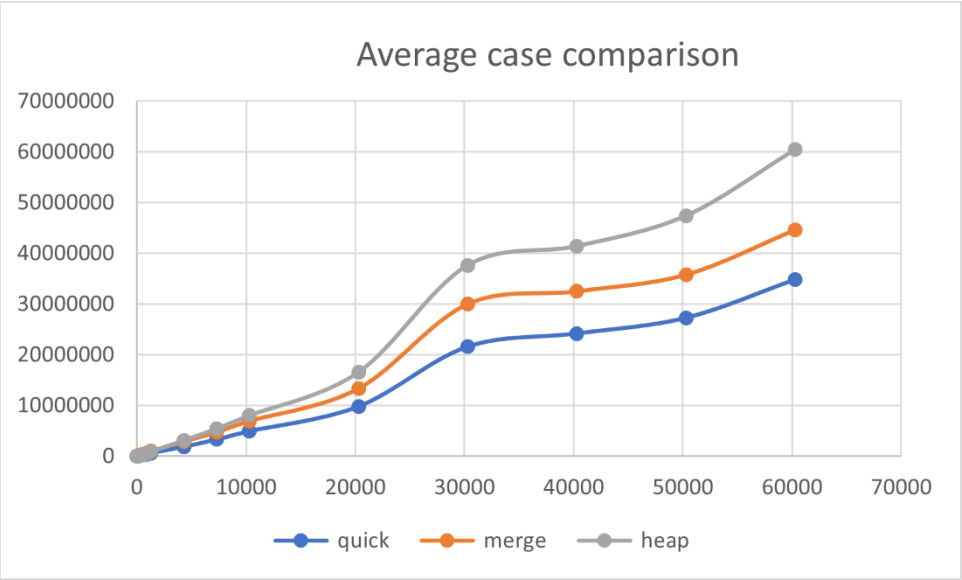
We have two main goals to check how they perform on inputs for the average case and secondly how will quick sort execution time differ in his worst case input compared to same input given to merge sort and heap sort. Most importantly thing is to do benchmark for all of this cases before we define how many time for each size algorithm should run and in the end we are taking average value for our comparisons. All of this will be achieved if we follow this steps.

1)Generate random numbers and fill it in array
2)Give that unsorted array to algorithms .
3)Give the Sorted array to algorithm to check how the quick sort differs from other algorithm
4) Record the times and do the comparison

3. Results

For my final result I used x axis for the sizes , y axis for the time .

In the first graph we will compare average cases of all algorithms



In the second graph we will compare worst case of quick sort with merge sort and heap sort. To show it on graph we will create graph for smaller values to make it more visible.

4. Conclusions

From the first graph we can see clearly that quick sort is more faster than heap and merge. And about Heap comparison to

Merge we can see that in more bigger values heap is slower than merge. Now coming to our second graph where we want to show that Quick sort is slower which is clearly visible. About Merge and heap we can say they have almost same execution time .