Comparing Quicksort, Mergesort

and Heapsort

Bar Segal

December 17/2021

1.)Introduction:

In this paper I will shed some light on a well known problem in the data structures and algorithms world – sorting an integers vector of size n. I will be using 3 sorting algorithms: Quicksort, Mergesort and Heapsort. I will investigate how long it take them to sort given the same vector data and their time complexity which is computed by a big O notation considering different cases of vector data.

1.1)Quicksort: is a divide and conquer algorithm. You pick an element called a pivot and you partition the container around it (in our case a vector).

The Partition() function is the key element of quick and happens recursively, what it does is making sure our Pivot is at the correct position that is that elements that are smaller than the Pivot are on it left side and elements that are larger than the Pivot are on the right. There’s a few options as for which position to choose for the Pivot:

1. Always pick first element as pivot.
2. Always pick last element as pivot (The option the I chose to use)
3. Pick a random element as pivot.
4. Pick median as pivot.

In terms of time complexity to run Quicksort ~

The best case Best Case is when the pivot element divides the container into two equal halves by coming exactly in the middle position which will result in O(nlogn).

The worst case is when the list is either arranged in ascending or descending order which means that it is not dividing the container into anything. Which will result in the time complexity increases to O(n^2).

The average case is when you pick the Pivot, the worst you can do is to not divide and the best you can do is to divide into two equal halves. The average case will find you getting a split of something in the middle of it. Which will result in a time complexity of O(nlogn) once constants are eliminated.

Quicksort is in place and not stable.

1.2)Mergesort: is also a divide and conquer algorithm. It divides the container into two halves and recursively calls itself for the two halves, and then merges the two sorted halves. **It uses a function called Merge()** for merging two halves in a sorted order. The Merge() function is very important and it assumes that the two halves are already sorted which is achieved by cutting it into small pieces.

Time complexity of Merge Sort is O(n\*Log n) in all the 3 cases (worst, average and best) as merge sort always divides the array in two halves and takes linear time to merge two halves.

It requires equal amount of additional space as the unsorted array. Hence its not at all recommended for searching large unsorted arrays.

Merge sort is stable and not in place.

1.3)Heapsort: is a comparison-based sorting technique based on Binary Heap data structure. We extract the root element and repeat the same process recursively the remaining elements.

We first build a binary tree out of our input and then we make it into a max or min heap, which is done using a function called Heapify().  a min heap is used to access the minimum element in the heap whereas the max heap is used when accessing the maximum element in the heap. Then we extract the max or min element and Heapify() again until we extracted all the elements.

The time complexity of Heapsort for all cases is O(n log n) building a heap takes O(n) and extracting the elements takes O(log n) which results in O(n log n).

Heapsort is in place and not stable.

2)Methodology:

I implemented the algorithms in C++. The results were generated for two categories: sorted vector(the worst case for Quicksort) and randomly shuffled vector of values (0, 1, . . . , n – 1). The algorithms were tested on vectors of sizes from 100, 200, 300, . . . , 10000. In both categories, each algorithm was given the same input data. Each vector size was tested 100 times. The result for each size is the average time it took for each algorithm to sort the input vector.

3)Results:

A graph for all 3 algorithms results of sorted vectors of up to 10000 elements are presented in figure 1, it is clear that Quicksort is several times slower than Mergesort or Heapsort for sorted vectors, In order to make a better comparison of Mergesort and Heapsort there’s figure 2 which shows us that Mergesort is roughly 2 times slower than Heapsort.

Figure 3 there’s again all 3 algorithms but this time for shuffled vectors – again up to 10000 elements, we can that Mergesort is the slowest one followed by Heapsort and Quicksort is the fastest, In comparison to the sorted vectors Quicksort have became drastically faster while Mergesort slowed down a little and Heapsort more or less stayed indifferent.

Figure 4 and 5 represents Sorted and Shuffled vectors respectively for up to 1000 elements. In figure 4 we can see the quick sort actually starts faster than Merge sort and around the 300 elements marks fall behind and become a lot slower, while Mergesort in general is a bit slower than Heapsort. In figure 5 we can see that Quick sort is the fastest followed by Heapsort and Mergesort being the slowest.

Chart, line chart

Description automatically generatedFigure 1 – All 3 algorithms tested on Sorted vectors in size of up to 1000 elements

Chart, line chart

Description automatically generated

Figure 2- same as Figure one but a side by side comparison of Mergesort and Heapsort

Figure 3 - All 3 algorithms tested on shuffled vectors in size of up to 10000 elements Chart, line chart

Description automatically generated

Chart, line chart

Description automatically generated

Figure 4 – All 3 algorithms tested on sorted vectors in size of up to 1000 elements

Chart, line chart

Description automatically generated

Figure 5 – All 3 algorithms tested on shuffled vectors of size of up to 1000 elements

4)Conclusion:

Quicksort performed best on shuffled vectors however on sorted vectors the performance took a major hit. Heapsort performed best on sorted vectors and was second to Quicksort on shuffled vectors. Overall Heapsort is the best out of the 3.