Comparison between insertion, merge, heap, and quick sort

MD RAKIB HASAN

20 December 2022

1 Introduction

In this paper, we implement four sorting algorithms, insertion, merge, heap, and quick sort. Moreover, comparing their time complexity as well as implementing hybrid sort, if needed.

1.1 Insertion sort

Insertion sort is an algorithm where small elements from the right side are inserted into the left side one by one and once at a time. It means that insert element in the perfect position. It is a simple algorithm.

In insertion sort, two loops use for sorting. In parent loop iteration cause from array* + 1 up to is less than the array's length. Current index value stored into val variable. Then in the inner loop current index and its left index has considered for comparison. If the current index is greater than 0 && the current index element is less than the left index element, later the left index element move to the current index. And the current index value decrement by one. If the current index is greater than 0 && the current index element is less than the left index element, then the inner loop performs again. And it is performing until (current index is greater than 0 && current index element is less than left index element) is false. At last, val value moved to the current index.

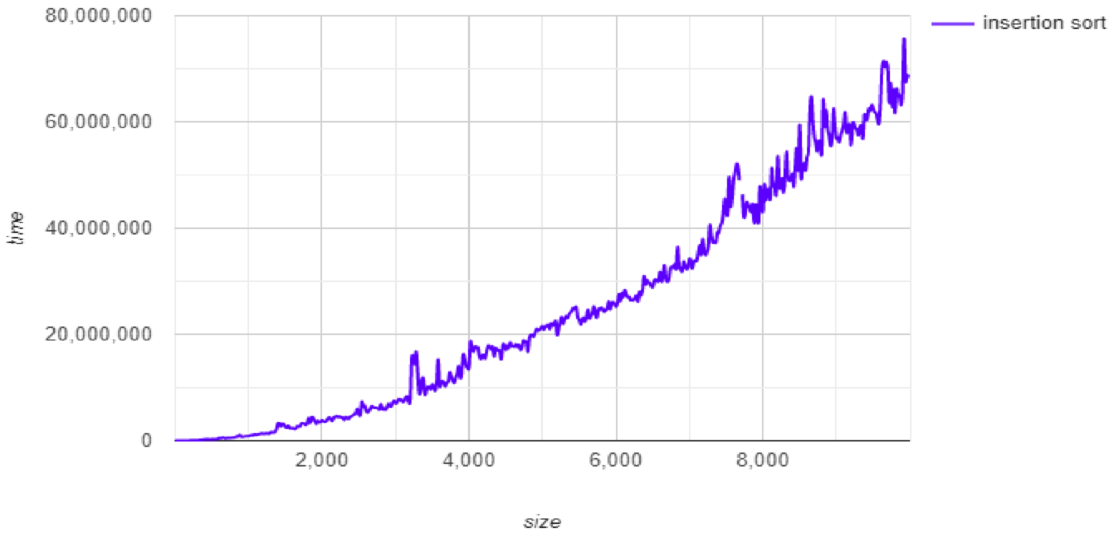
The best case of insertion sort is O(n). If an array is already sorted then the inner loop will not perform. So, the parent loop will perform from array* + 1 up to less than the array length. If an array has n elements then it will iterate from 1 to n - 1.

Worse case of this sort is when an array is in descending order. For inserting the first element to n-1 it performs n - 1 iteration. For inserting the last element it needs n-1 comparison and n-1 swap. For inserting the second last element in the second index it performs n-2 comparison and n-2 swap.

So to compare it needs 1+2+3+......+n-2+n-1 = n-1, which is almost n, and for swapping, it performs 1+2+3+.......+n-2+n-1 = n-1, which is almost n operations. Comparison time complexity is O(n) and swapping time complexity is O(n). Therefore, the time complexity of insertion worse case is O(n)xO(n) = O(n*n) = O(n^2).

Insertion sort's average case often behaves like a worse case. As a result, it can assume that the average case time complexity is O(n^2).

The insertion sort is stable.



1.2 Merge sort

Merge sort is the sorting algorithm that is based on the divide and conquers paradigm. This algorithm divides the array in half until it is not dividable It means that array division will be stopped when the array has 1 element or no element.

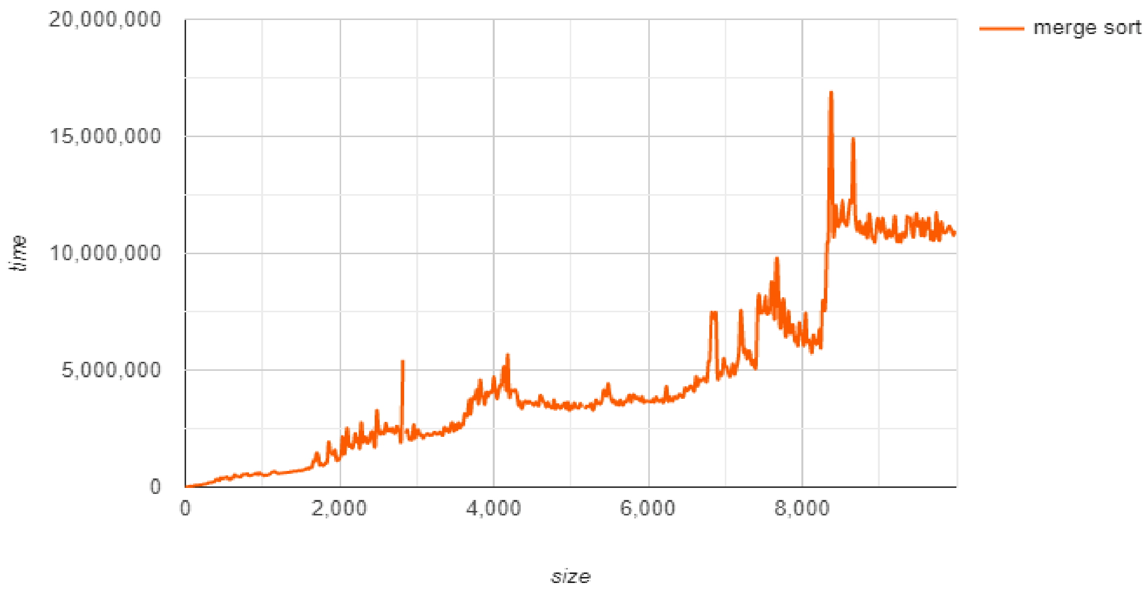
First, it checks left index of the array is smaller than the right index. If it is true then the array divides into half and finds the midpoint and it creates 2 sub-arrays. Later, each sub-array is divided into 2 small arrays and it continues until the array can not be divided able. Then it starts to merge the array recursively.

First of all mergeSort function is called with an array, begin = 0, end = n - 1 parameter. Here n is the size of the array. Inside function if begin is less than end == true then, after finding midpoint midPoint mergesort(array, begin, midPoint) and mergeSort(array, midPoint+1, end) is called again. And call another function merge(arr, begin, mid, end).

Now time complexity calculation. Firstly each time array is divided into 2 parts and creates a binary tree. Binary tree time complexity is o(logn). Secondly, merge back the arrays into a single array. Therefore it will take up to n steps. so total time complexity is o(nlogn).

In every case like best, average, and worse merge sort creates a binary tree structure by splitting the array. And again merge back arrays into one array. Therefore, o(nlogn) is the time complexity in best, average, and worse cases.

The merge sort is stable.



1.3 Heap sort

Heap sort is a comparison-based sorting algorithm based on the Binary Heap data structure. It is similar to the selection sort. Finding the minimum element and placing it at the beginning and repeating the same process for the rest of the elements. Or finding the maximum element and placing it at the end of the array.

Heap sort has 2 parts. Heapify and delete elements. Here, we will talk about max heap in the case of heapify.

Heapify is the process to make a binary tree. For making this n/2-1 is considered as root. According to this root left and right children can be created. The left child index is l = 2*root+1 when l¡n and the right child index is r=2*root+2 when r is less than n. Then from the left child may have another 2 children and the left child will be considered as a root of new children. Similarly, the right child may have another 2 children and the right child will be considered as a root of new children. This process will be continued until the leaf node has been found. In each step of creating binary, we are also doing max heap. To do this, there is one methodology that, if the child node value is greater than the root node value then they swap their position.

Now it is time to talk about the delete element. To do this a loop performs n-1 iteration in descending order. In every iteration first and highest indexes swap their value and the highest index decrement by one. It means that the highest index has been deleted. Then heapify again.

Firstly, quickSort(array, size = array size) function is called. Then inside function root = size / 2 - 1. In the loop, if root is less than and = 0, later maxHeapify(arr, size, root) is called and root values decrease one by one. Every step of the loop creates the step of a binary tree called maxHeapify. maxHeapiy function is called for making max heap. After that, a loop is called in descending order considering l = size - 1. Inside this loop swap the first index value and last index l value and l– and maxHeapify(arr, l, 0). This functionality is used for moving the largest element at the end of the array and deleting it.

Inside the heapsort function, there is a while loop for creating a heap. This loop iterate up to half of the array length. So time complexity of this part of the code is o(n/2), which can assume as o(n). Moreover, there is another maxHeapify function inside the while loop for creating a max heap. Here it is creating a binary tree. Binary tree time complexity is o(logn). As a result, the total time complexity of creating max heap is o(nlogn).

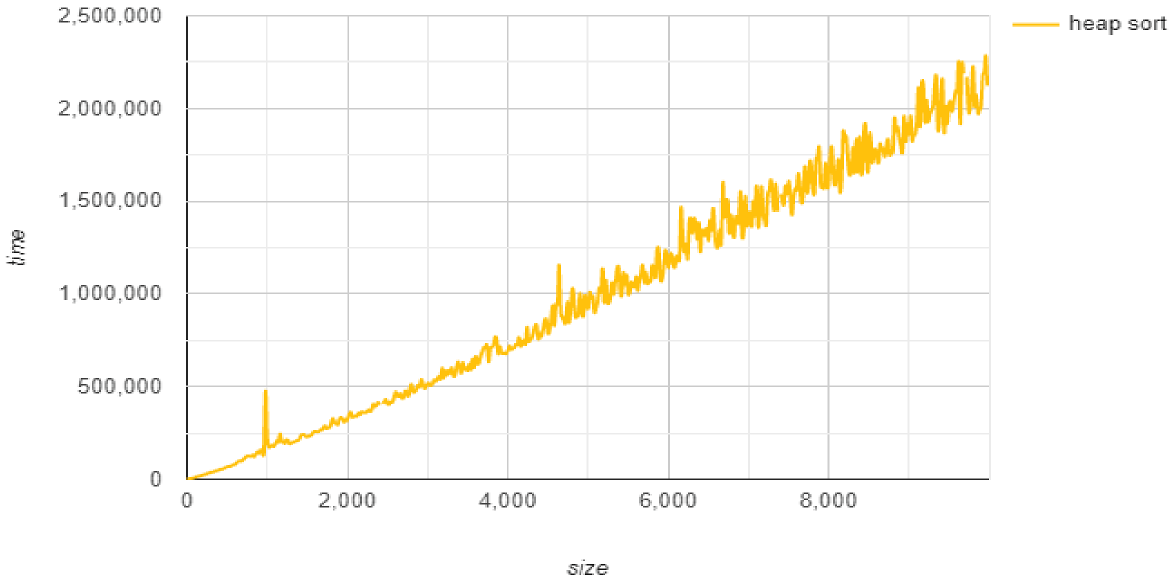
On the other hand at the time of removing a max element, again maxHeapify using for swapping the max-valued node until that node comes down to the bottommost level. It is also working over a binary tree. So its time complexity also o(logn). As well as it is repeating size up to array length. So the total time complexity of removing is o(nlogn).

Now total time complexity is o(nlogn + nlogn) = o(2nlogn), which can assume as o(nlogn).

The worse case of heap sort can happen if all the value in the array is distinct. So, at the time of removing a max element, the number of the swap would be logn and it is the most. So, for n time array size it is o(nlogn).

For calculating average case time complexity all possible inputs must be considered (best case and worse case). So, average case = (best case + worse case) / 2. Now, average case = (o(nlogn) + o(n)) / 2. Which can assume as o(nlogn). So average case time complexity is o(nlogn).

The heap sort is not stable.



1.4 Quick sort

Quick sort is similar to merge sort in the case of divide and conquer. It is partition based. In this algorithm, an index is considered a partitioned index which is also called the pivot. In 4 ways partition can be possible. 1. Always pick the first element as a pivot. 2. Always pick the last element as a pivot. 3. Pick a random element as a pivot. 4. Pick an element as a pivot.

In partition, an element is considered a pivot. Here we have used Hoare's partitioning scheme. In this partitioning method first element has taken for the pivot. And there is 2 index work in this method. One is left index l = 1 and the other one is right index r = size. A loop does some operation based on the l is less than r condition. Then if the value of the l index is not greater than the pivot it will do nothing. Else, it will go further to check that value of the r index is smaller than the pivot. If it is true then the value of the l and r indexes will swap their position and index r is decremented by one. Else, only r will decrement by one. And the value of the l index is greater than the pivot So, so the l value will be the same as the previous one. And at the end pivot value and r index value swap their positions.

We have used this method inside makePartition function by adding some other conditions. If Hoare's partitioning scheme happened then it will do the all process of Hoare's partitioning scheme method and return the pivot index which is l. Which is used as the breakpoint. If Hoare's partitioning scheme does not happen then it will swap pivot and pivot index + 1 value if the pivot is greater than pivot index + 1 value and return l. Else, it returns only l. (Here l is the pivot index).

Here we going to work with the first element as a pivot. Every first element of an array or subarray will consider a pivot. first, the quick sort function has called with 3 parameters which are array, begin = 0, and end = n-1. After that, based on begin is less than end == true condition partition function is called to get the breakpoint index breakPoint. And quicksort function has been called recursively 2 times, first, one with parameters array, begin, breakPoint - 1, and the second one with parameters array, breakPoint + 1, end. It continues recursively according to begin less than end == true condition.

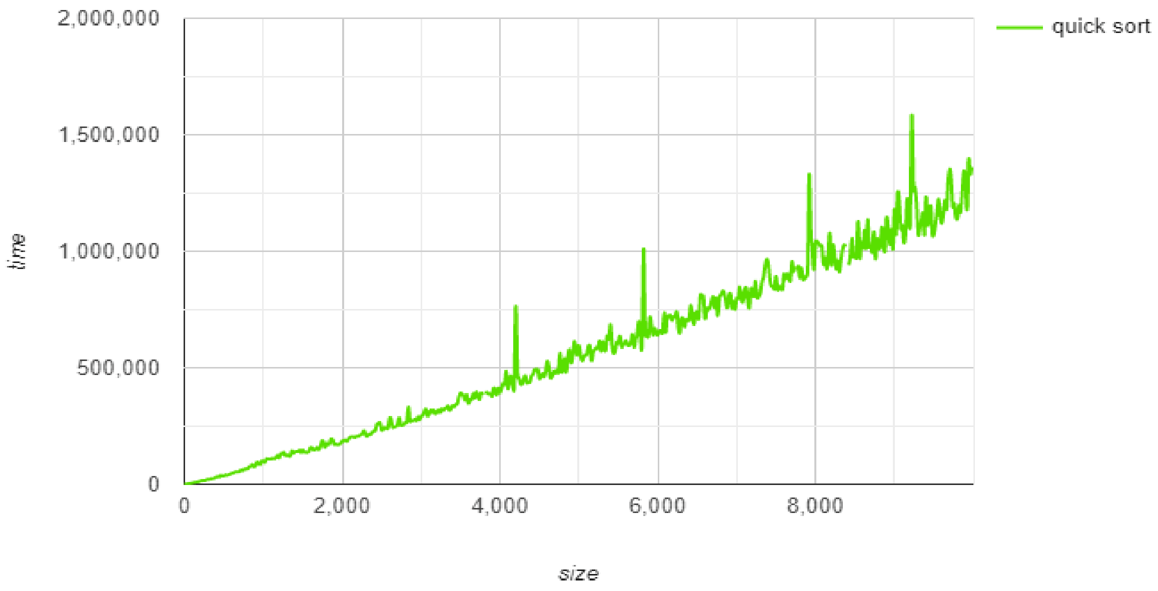
In quick sort, an index is selected as a pivot and the array is divided into 2 parts and makes a binary tree. So, the height of the tree is logn. And for inserting values that are less or equal to the pivot value on the left side will take n operations. So time complexity is o(nlogn).

The best case exists when the mean value of an array is selected as a pivot. The array is divided into 2 equal part and the height of the tree become minimum. But in this case, the time complexity is still (nlogn).

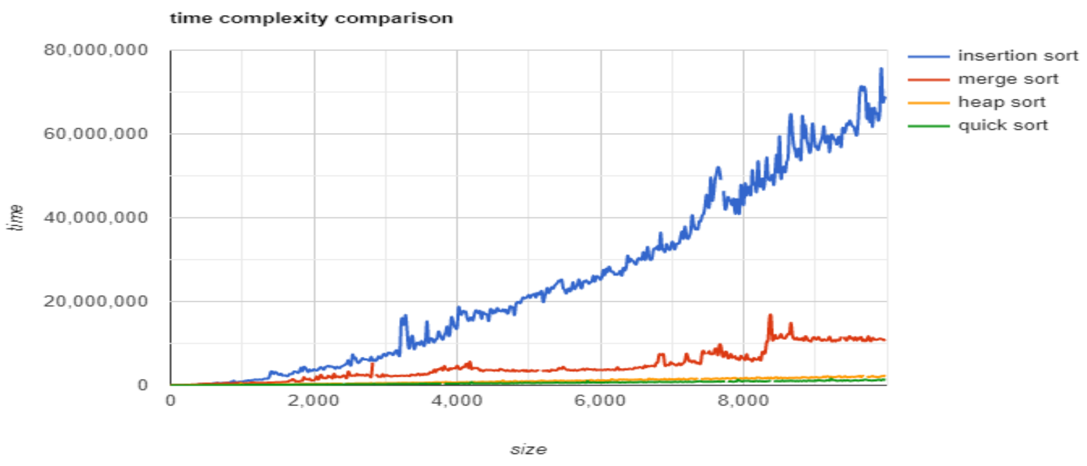
Worse case is possible when an array is sorted and the smallest or largest value is taken as a pivot. For this reason, binary trees turn into only left or right-sided. It means that the binary tree becomes only one side, the opposite side has no element. So tree height will be n. And the top node performs n operations, then n - 1,............,1. So, the worse case time complexity is o(n^2).

For average all possible conditions has considered. Average case time complexity can not be better than the best case. Average case time complexity o(nlogn).

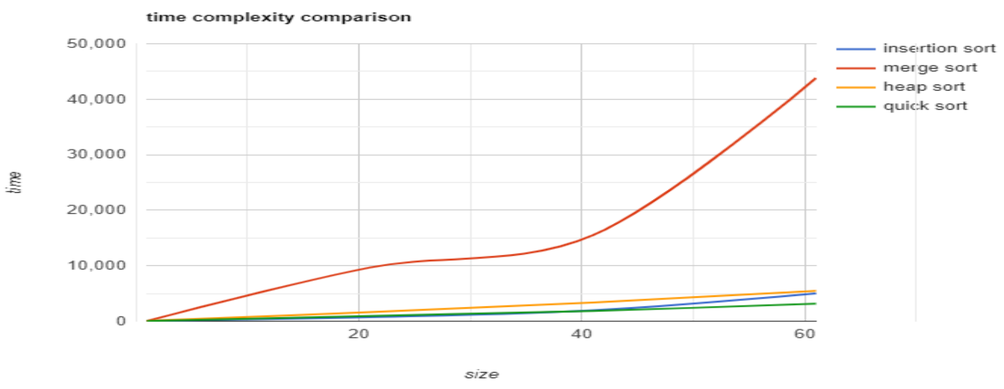
The quick sort is not stable.

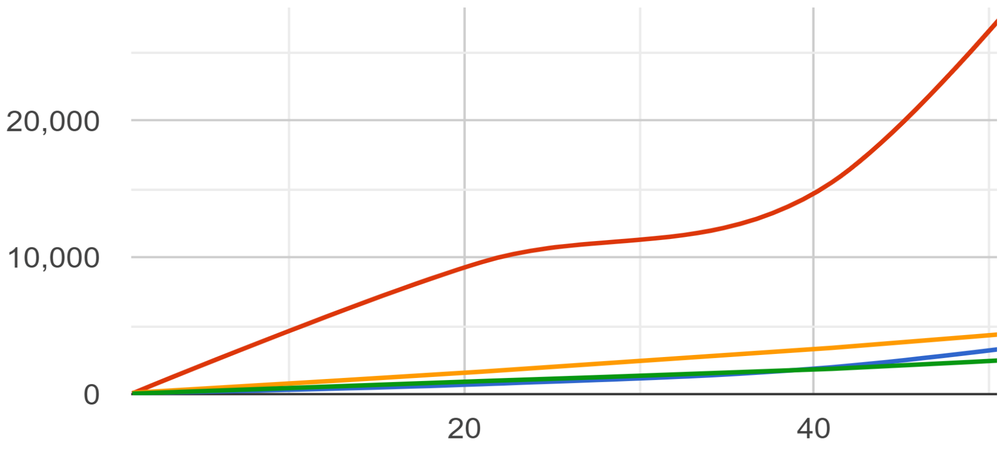


2 Comparison

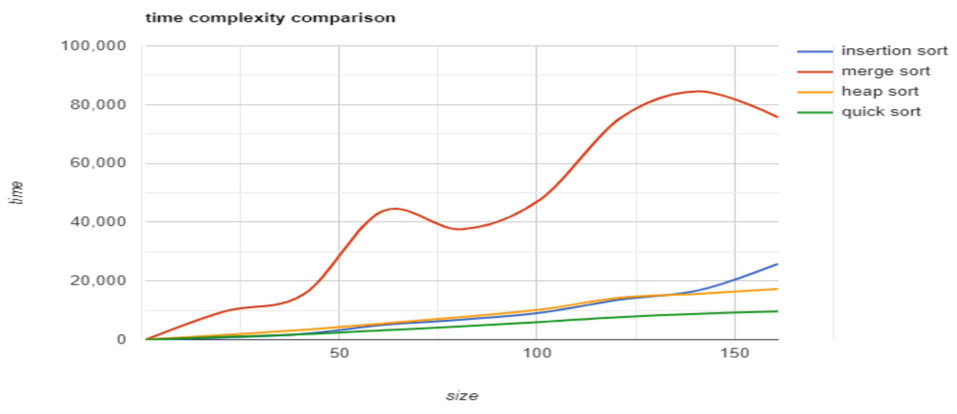


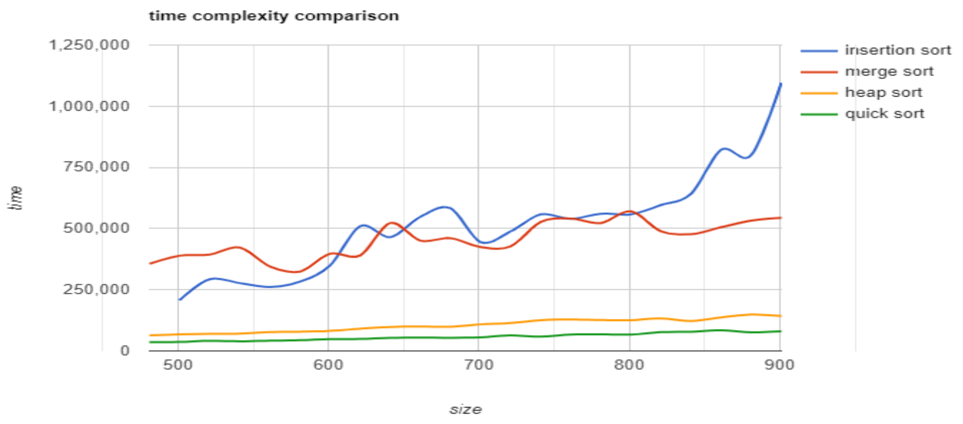
Array size from 1 to 10000 and size increased by 20





Insertion sort is the best around array size 55





After value around 600 merge sort is better than insertion sort

Insertion sort worse and the average case takes too much time for sorting. For a small size of the array(around 600), insertion sort is better than merge sort. Merge sort is better than insertion sort for large size of arrays. Heap sort time complexity is the same as merge sort in all types of cases[o(nlogn)]. But It performs better than merge sort. Quick sort is faster than heap sort. However, in the case of worse quick sort behave like insertion sort o(n^2).

That's why in this case heap sort is better. On the other hand, the quick sort average is o(nlogn), so it is preferable.

After comparing all cases it appeared that for small size insertion sort is get priority(size is less than 55). And large size array quick sort is best. From these two sorting algorithms another hybrid sorting algorithm implementation is possible.

Insertion sort worse and the average case takes too much time for sorting. For a small size of the array(around 600), insertion sort is better than merge sort. Merge sort is better than insertion sort for large size of arrays. Heap sort time complexity is the same as merge sort in all types of cases[o(nlogn)]. But It performs better than merge sort. Quick sort is faster than heap sort. However, in the case of worse quick sort behave like insertion sort [o(n^2)].

That's why in this case heap sort is better. On the other hand, the quick sort average is o(nlogn), so it is preferable.

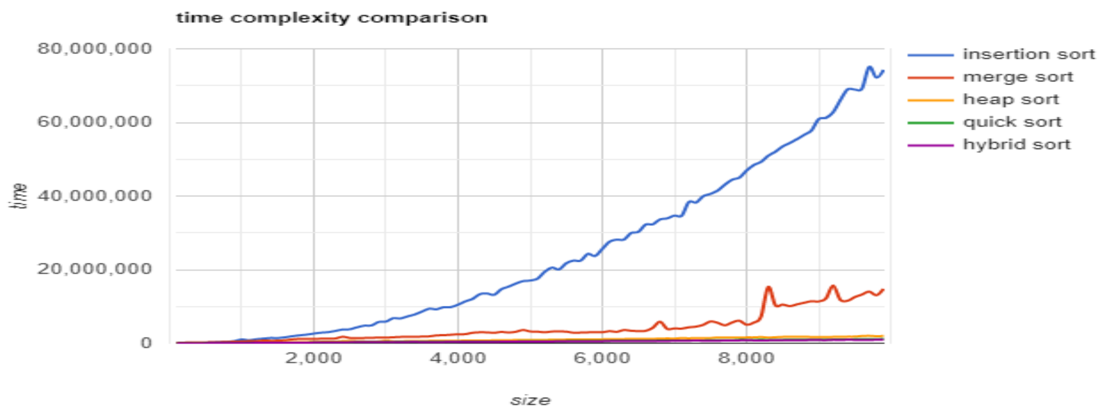
After comparing all cases it appeared that for small size insertion sort is get priority(size is less than 55). And large size array quick sort is best. From these two sorting algorithms another hybrid sorting algorithm implementation is possible.

3 Hybrid sort and comparison:

Hybrid sort is the addition of insertion and quick sort. hybridSort function takes 3 parameters and these are array, first index, and size - 1. Inside this function, there are two parts. If the array size is smaller than 55 then it will call the hybridGeneral function. hybridGeneral function is actually an insertion sort. Else, it will call the hybridSort function recursively, which is actually a quick sort when the array size is larger than 54.

In hybrid sort hybridGeneral(insertion sort) always work under constant size condition and it works for a less size array. So, quick sort has the highest priority in the hybrid sort. So its time complexity is the same as quick sort.

From our previous analysis, we have found that quick sort is better than insertion, merge, and heap sort. Let's compare quick sort and hybrid sort. Hybrid and quick sort time complexity is the same but inside hybrid sort, there is an insertion sort under constant size conditions. So when the size is less than 55 then the insertion sort is better than all other sorts. Insertion sort prevents quick sort work when the array size is smaller than 55 and omits more partitions for making binary trees up to the leaf. That's why the height of a binary tree becomes smaller than naturally quick sort. As a result hybrid sort is slightly better than quick sort.



4 Result:

The Hybrid sort is best without worse cases.

5 conclusion:

For small-size arrays, it is better to use insertion sort. And for big sizes of arrays, quick sorting is the best. But, in the case of quick sort worse case heap sort is much better. By using insertion and quick sort it is possible to implement hybrid sort. It is better than insertion, merge, heap, and quick sort except for the worse case.