Algorithms Lab

Assignment 1: Running time of Quick Sort, Merge Sort and Heap Sort

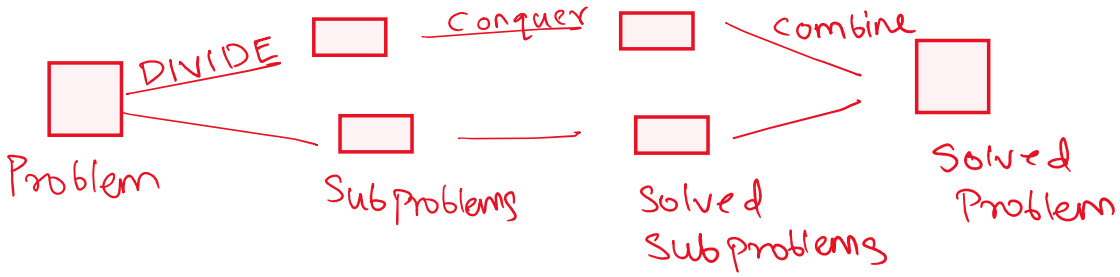
Aditya Badayalya

510819056

IT-Hy

Divide and Conquer Approach:

In divide and conquer algorithms, the problem is divided into multiple sub problems and then later the subproblems are solved accordingly by breaking them down to the smallest of levels. After solving them, the subproblems then are combined together to obtain the solution of the original problem.



Merge Sort Algorithm:

Merge sort is a divide and conquer type algorithm and as the name suggests, it is a sorting algorithm. In Merge Sort, the given array is broken or divided into two subarrays midway and later these subarrays are again divided into smaller parts to the atomic levels and then as an array with a single element is already sorted, these arrays are merged accordingly to obtain the original array in the sorted order.

```
class Solution
{
    public:
    void merge(long long arr[], long long l, long long m, long long r){
        long long leftLenth=m-l+1;
        long long rightLength=r-m;
        auto leftAr = new long long[leftLenth];
        auto rightAr = new long long[rightLength];
        for(long long i=0;i<leftLenth;i++){
            leftAr[i]=arr[l+i];
        }
        for(long long i=0;i<rightLength;i++){
            rightAr[i]=arr[i+m+1];
        }
        long long i1=0,i2=0,im=l;
        while(i1<leftLenth&&i2<rightLength){
            if(leftAr[i1]<=leftAr[i2]){
                arr[im]=leftAr[i1];
                i1++;
            }
            else{
```

```
                arr[im]=rightAr[i2];
                i2++;
            }
            im++;
        }
        while(i1<leftLenth){
            arr[im]=leftAr[i1];
            im++;
            i1++;
        }
        while(i2<rightLength){
            arr[im]=rightAr[i2];
            im++;
            i2++;
        }
        free(rightAr);
        free(leftAr);
    }
    public:
    void mergeSort(long long arr[], long long l, long long r){
        if(l<r){
            int mid = l +(r-l)/2;
            mergeSort(arr,l,mid);
            mergeSort(arr,mid+1,r);
            merge(arr,l,mid,r);
        }
    }
};
```

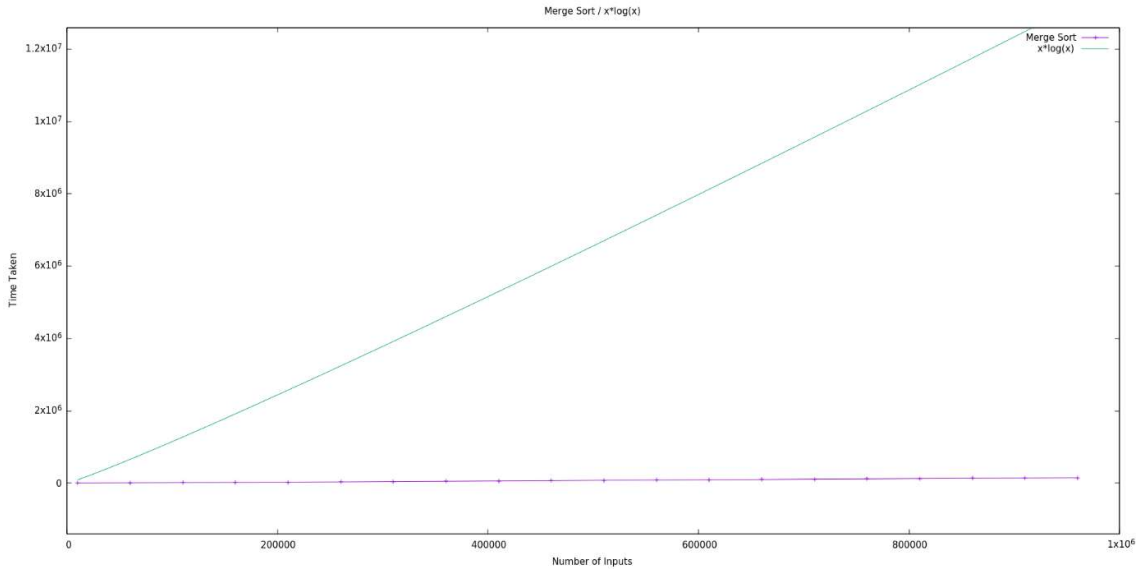
Here, the `mergeSort` function determines the middle point of the array and then recursively divides the array till a single element is obtained making this function recursive in nature. The `merge` function then accordingly merges the individually sorted arrays to form the original array but now it is sorted.

The `merge` function has in total four loops in it and each of them can run to a maximum of n times. Neither of these four loops is nested in each other so the maximum time complexity would be O(n). On closer inspection it can also be seen that these four loops can be executed to a maximum of n times together which further confirms that the function has a linear time complexity. The array is divided into two subarrays each time, so the maximum space required would be equal to the space required by the original array making the space complexity O(n) as well.

The `mergeSort` function is recursive in nature, its time complexity can be determined using a recurrence relation T(n)=2*T(n/2) + O(n)

This recurrence relation can be solved by Recurrence Tree Method which involves breaking down of the tree to certain levels and then determining the sum of the work done on all the levels (which would turn out to be a geometric progression). And hence it can be observed that the overall complexity of the algorithm would amount to O(nlog(n)) implying that the time required by the algorithm in the worst case would amount to O(nlog(n)) where 'n' is the number of elements in the array.

Below is a graph depicting the same (time is in milliseconds):



Quick Sort Algorithm:

Quick Sort like merge sort is also a divide and conquer algorithm. It involves picking a pivotal value in the array and then partitioning the array around the pivot point. There are various ways of picking the pivot value such as:

1. Picking the first element as pivot
2. Picking the last element as pivot
3. Picking a random element as pivot

The key process in quicksort involves the arrangement of elements smaller than the pivot to its left and the elements larger than it to the right thus determining the appropriate locations for them.

```
class Solution
{
    public:
    int partition(long long arr[], long long l, long long r){
        long long pivot = arr[r];
        long long i = l-1;
        for(long long j=l;j<r;j++){
            if(arr[j]<pivot){
                i++;
                swap(arr[i],arr[j]);
            }
        }
        swap(arr[i+1],arr[r]);
        return (i+1);
    }
    public:
    void quickSort(long long arr[], long long l, long long r){
        if(l<r){
            long long pi = partition(arr,l,r);
            quickSort(arr,l,pi-1);
            quickSort(arr,pi+1,r);
        }
    }
};
```

In the above algorithm, the last value is always picked as the pivotal value from the provided array or subarray. Then the elements are rearranged accordingly in the array bringing all the elements less than that of the pivotal value to it's left and the others to its right. The elements are swapped with other elements in the `partition` function to match the condition i.e., elements less than pivot to its left and greater than it to its right. All of this occurs in linear time i.e.,

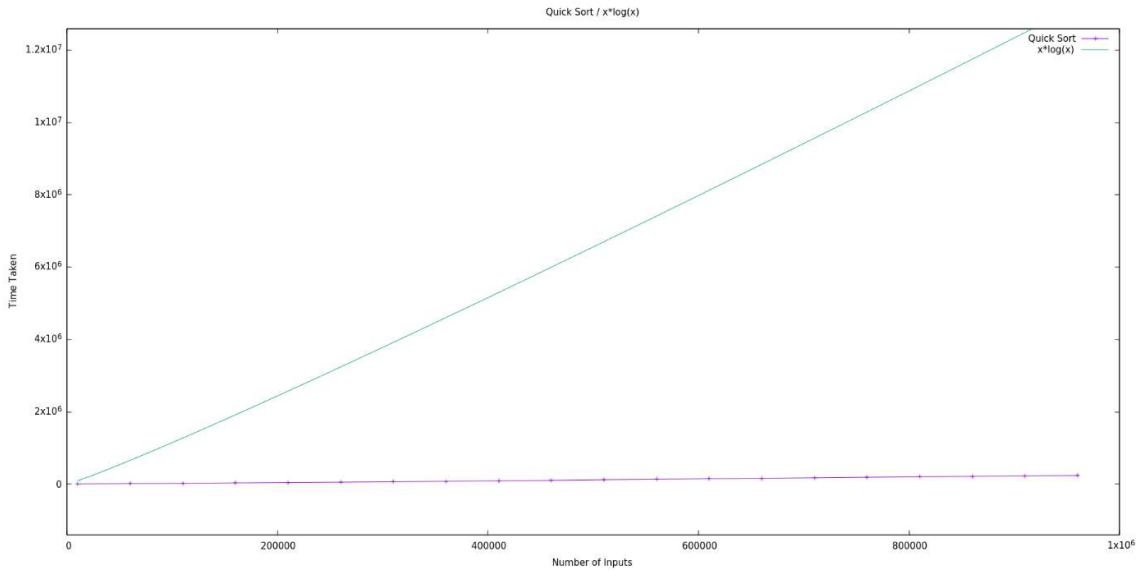
O(n) and in the end the final position of the pivotal value is returned after which `quicksort` is applied again to the left and the right halves from the pivotal position. The space complexity of this algorithm is constant i.e., O(1).

The `quicksort` function is recursive in nature and its time complexity can be determined using a recurrence relation T(n) = T(k) + T(n-k-1) + O(n) where first two elements are recursive in nature and the third is the partition process, 'k' is the number of elements smaller than pivot and 'n' is the total number of elements.

The worst case scenario for this algorithm would involve always picking either first or last element as the pivot which would lead to the recurrence relation being T(n) = T(0) +T(n-1) + O(n) which is equivalent to T(n) = T(n-1) + O(n) and on solving this recurrence relation we can obtain a result from which it can be determined that the worst case time complexity is equivalent to $O(n^2)$.

For the best-case scenario in this algorithm, picking the middle value as pivot and then working on the algorithm which would make the recurrence relation as T(n) = T(n/2) +T(n/2) + O(n) i.e., T(n) = 2T(n/2) + O(n) and on solving this recurrence relation it can obtained the time required by the algorithm in best-case would amount to O(nlog(n)) where 'n' is the total number of elements in the array.

Below is the graph depicting the behavior of the algorithm (time is in milliseconds):



Heap Sort:

Heap sort is the comparison-based algorithm that employs the use of Binary Heap data structure. In this algorithm we find the minimum element first and then place it at the beginning and then repeat the process for the remaining elements. A Binary Heap is a complete binary tree in which the value of the node is either greater than both of its children or smaller than it. The first one here is called the max-heap and the latter one is called the min-heap.

The Heap Data structure can be represented as an array as the heap is a complete binary tree and the children to the node can be found at the indices (2*i +1) and (2*i +2) where i is the index of the node and the indexing begins at 0.

A heap has to be built from the array which can be either min heap or mx heap according to the requirements. A heap can only be built if all of its children are heapified i.e., all the children of the nodes are turned into heaps. Therefore building a heap has to be a bottom up process which means we start from behind in the array.

```
class Solution
{
    public:
    void heapify(long long arr[], long long n, long long i){
        long long largest=i;
        long long left = 2*i +1;
        long long right = 2*i +2;
        if((left<n)&&(arr[left]>arr[largest])){
            largest=left;
        }
        if((right<n)&&(arr[right]>arr[largest])){
            largest=right;
        }
        if(largest!=i){
            swap(arr[largest],arr[i]);
            heapify(arr,n,largest);
        }
    }
    public:
    void heapSort(long long arr[], long long n){
        for(long long i=n/2-1;i>=0;i--){
            heapify(arr,n,i);
        }
        for(long long i=n-1;i>0;i--){
            swap(arr[i],arr[0]);
            heapify(arr,i,0);
```

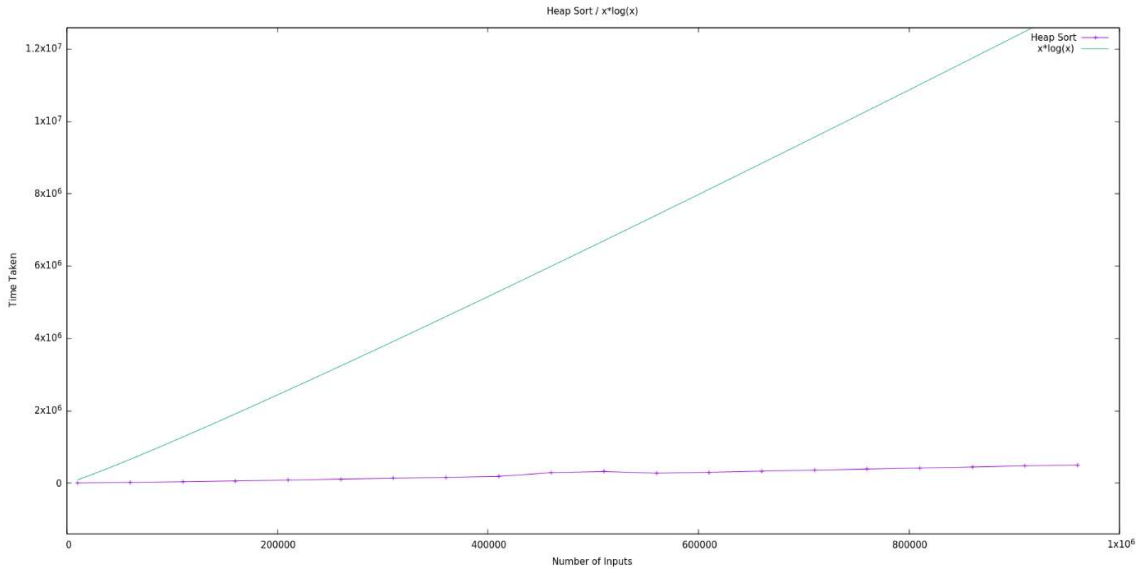
```
        }
    }
};
```

The `heapify` function involves creating a heap from the given data. Checking the children of the given node, which can be found at the indices (2*i+1) and (2*i+2) (assuming the indexing starts at 0) and computations are performed accordingly i.e., swapping the elements if required and hence creating a proper heap.

The `heapSort` function involves building a heap in a bottom-up manner. In the above algorithm, a min-heap is created to sort the data in the ascending order. After the completion of this function, the smallest element in the array is stored in the root.

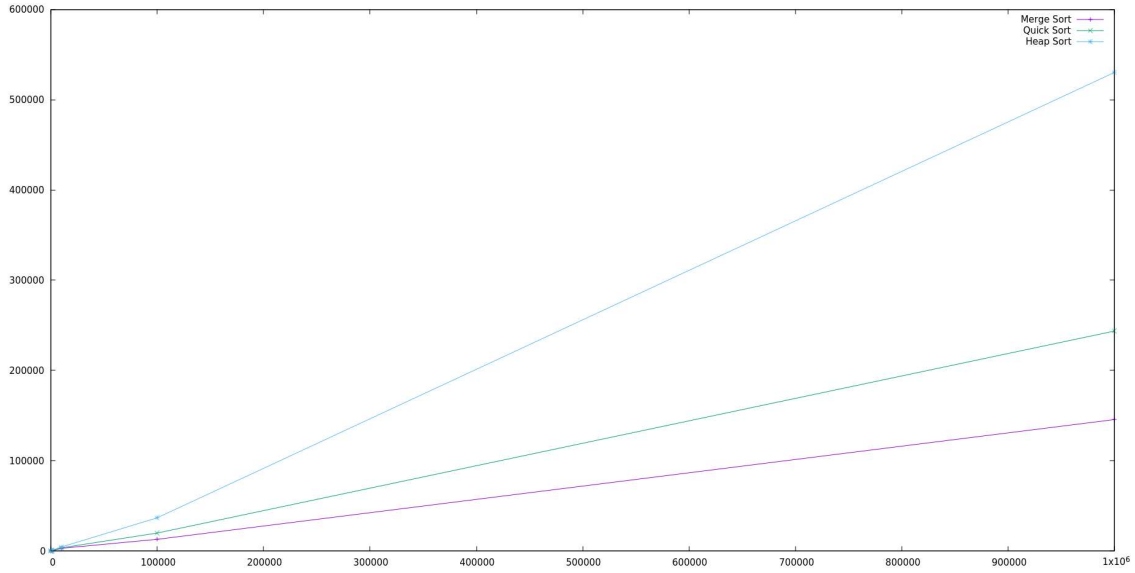
The heapsort algorithm has an over all time complexity of O(nlog(n)) which can be determined from the above implementation. The linear part of the algorithm comes out from the loop in the `heapSort` function which has the nested `heapify` function which has the logarithmic complexity and thus giving the overall complexity to be O(nlog(n)) where 'n' is the total number of elements in the array.

Below is the graph depicting the behavior of algorithm (time is in milliseconds):



Comparative Analysis:

Here is a graph depicting the behavior of Heap-sort, Merge-sort and Quick-sort algorithms at a certain number of inputs:

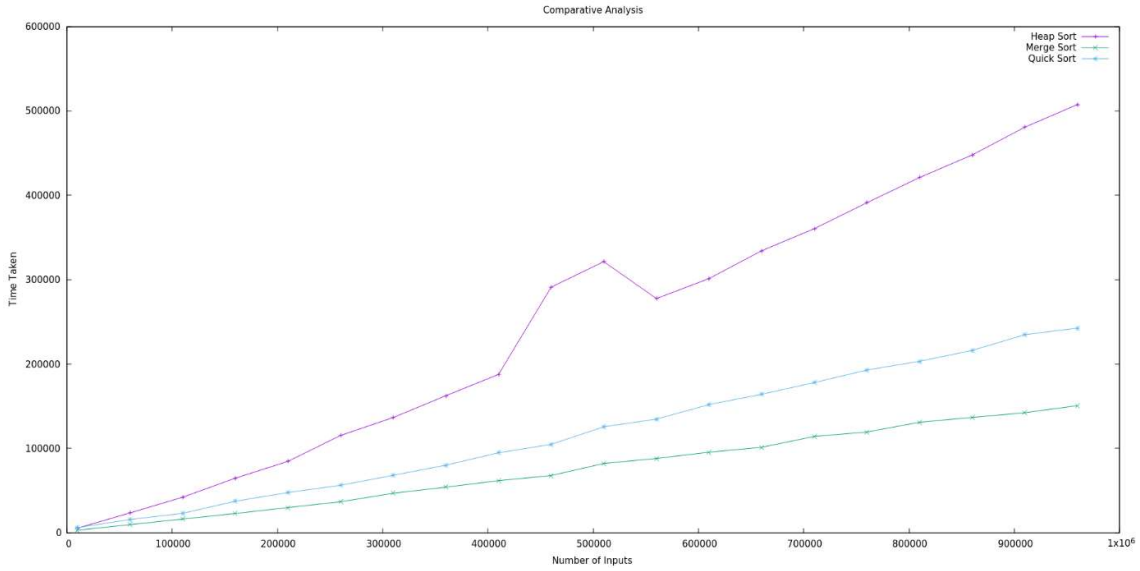


The x-axis represents the number of inputs while the y-axis represents the time taken in milliseconds.

For the above graph, the algorithms were run on random datasets of length 10, 100, 1000,10000, 100000 and 1000000 10 times each and the time was calculated each time consequently the average value of the time required for each length of the dataset and by each algorithm was plotted.

Another graph was plotted for the datasets having length between 10000-1000000 for a closer look at the behavior of the curves. The graph obtained was as follows:

The y-axis denotes time in milliseconds while the x-axis denote the length of the input dataset.



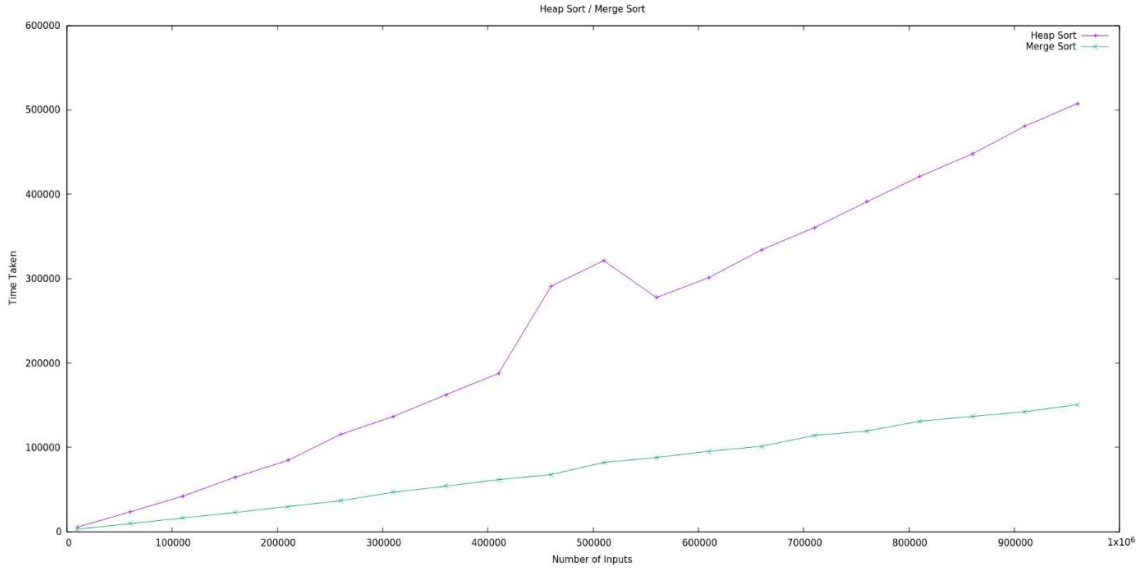
When Quick Sort algorithm is compared with merge sort, for smaller values Quick Sort proves to be faster than Merge Sort as the Quick sort is a cache friendly sorting algorithm and has a good locality of reference however Merge Sort requires an extra space of O(n) and the time required for allocation and deallocation of the space is also considered in the overall runtime of the algorithm proving it to be slightly slower than Quick Sort initially. Merge Sort always splits the elements of the dataset into half as compared to Quick Sort which splits the dataset randomly providing Merge Sort an edge over Quick Sort when large datasets are taken into consideration.

The worst-case complexity of Quick Sort is $O(n^2)$ where as the complexity of Merge Sort is always O(nlog(n)) proving Merge Sort more effective for larger datasets.

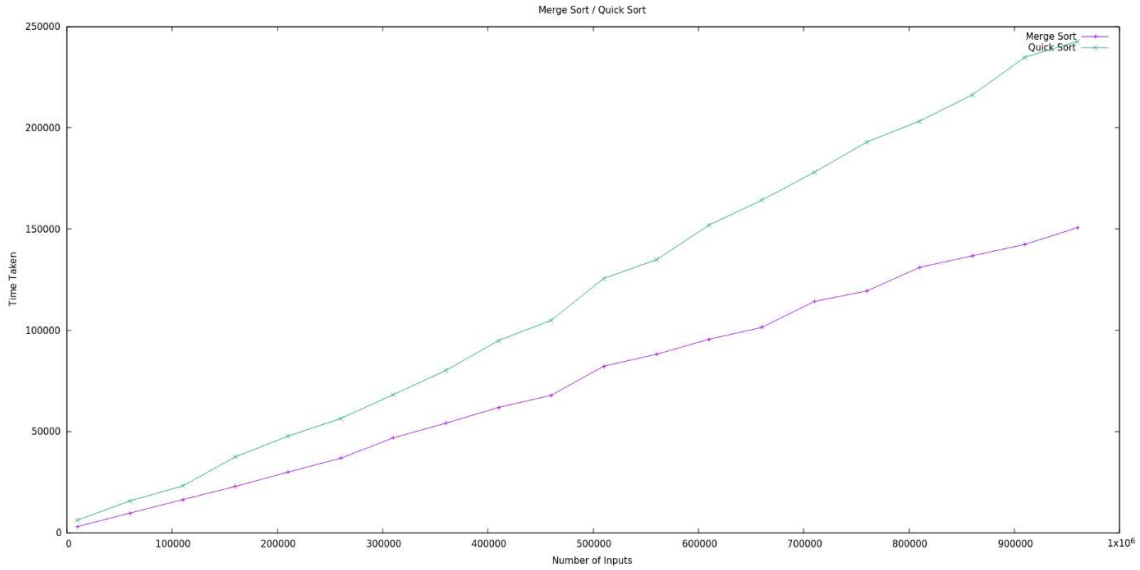
Heap Sort algorithm like Quick Sort does not require any additional space and therefore is an in-place algorithm. However, there might be a lot of un-necessary swapping in the algorithm proving it to be a little less effective. For example, in a sorted array of length 10, 21 swaps occur in total which increases the time taken by the algorithm as a whole where is when the same is done with Merge Sort or Quick Sort, the elements are just compared and then no other operation is performed on them after comparison making them relatively faster.

Here are some graphs depicting the behavior of these algorithms w.r.t each other in the range of input size (10000-1000000):

Heap Sort v/s Merge Sort



Merge Sort v/s Quick Sort



Quick Sort v/s Heap Sort