Experiment No. : 2

Title: Divide and Conquer Strategy

Batch: A2 Roll No.: 16010421063
Experiment No.: 2

Aim: To implement and analyse time complexity of Quick-sort and Merge sort and compare both.

Explanation and Working of Quick sort & Merge sort:

Algorithm:
Quick sort:
Quick Sort Pivot Algorithm
Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right
Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot
Quick Sort Algorithm
Step 1 − Make the right-most index value pivot
Step 2 − partition the array using pivot value
Step 3 − quicksort left partition recursively
Step 4 − quicksort right partition recursively

Merge sort:
```
MergeSort(arr[], l, r)
If r > l
1. Find the middle point to divide the array into two halves:
        middle m = l+ (r-l)/2
2. Call mergeSort for first half:
        Call
mergeSort(arr, l, m)
3. Call mergeSort
for second half:
        Call mergeSort(arr, m+1, r)
4. Merge the two halves sorted in step 2 and 3:
        Call merge(arr, l, m, r)
```

Sample Quick Sort Step by Step Execution

Consider the following array: 50, 23, 9, 18, 61, 32. We need to sort this array in the most efficient manner without using extra place (inplace sorting).

1. Make any element as pivot: Decide any value to be the pivot from the list. For convenience of code, we often select the rightmost index as pivot or select any at random and swap with rightmost. Suppose for two values “Low” and “High” corresponding to the first index and last index respectively.

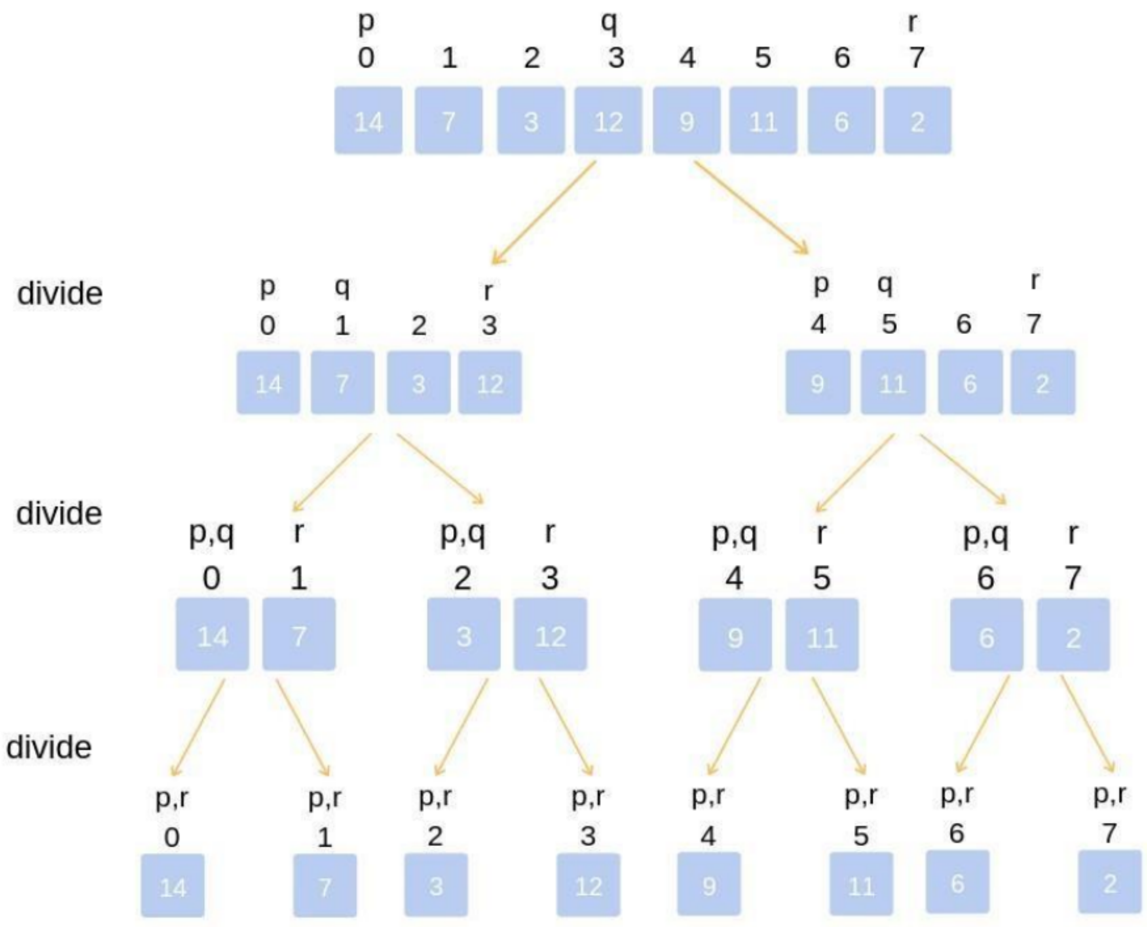
- **o In our case low is 0 and high is 5.**
- **Values at low and high are 50 and 32 and value at pivot is 32.**

2. Partition the array on the basis of pivot: Call for partitioning which rearranges the array in such a way that pivot (32) comes to its actual position (of the sorted array). And to the left of the pivot, the array has all the elements less than it, and to the right greater than it.
    a. In the partition function, we start from the first element and compare it with the pivot. Since 50 is greater than 32, we don't make any change and move on to the next element 23.
    b. Compare again with the pivot. Since 23 is less than 32, we swap 50 and 23. The array becomes 23, 50, 9, 18, 61, 32
    c. We move on to the next element 9 which is again less than pivot (32) thus swapping it with 50 makes our array as 23, 9, 50, 18, 61, 32. o Similarly, for next element 18 which is less than 32, the array becomes 23, 9, 18, 50, 61, 32.Now 61 is greater than pivot (32), hence no changes
    d. Lastly, we swap our pivot with 50 so that it comes to the correct position. Thus the pivot (32) comes at its actual position and all elements to its left are lesser, and all elements to the right are greater than itself. Step 2: The main array after the first step becomes
3. Now the list is divided into two parts:
    - Sublist before pivot element
    - Sublist after pivot element
4. Repeat the steps for the left and right sublists recursively. The final array thus becomes 9, 18, 23, 32, 50, 61.

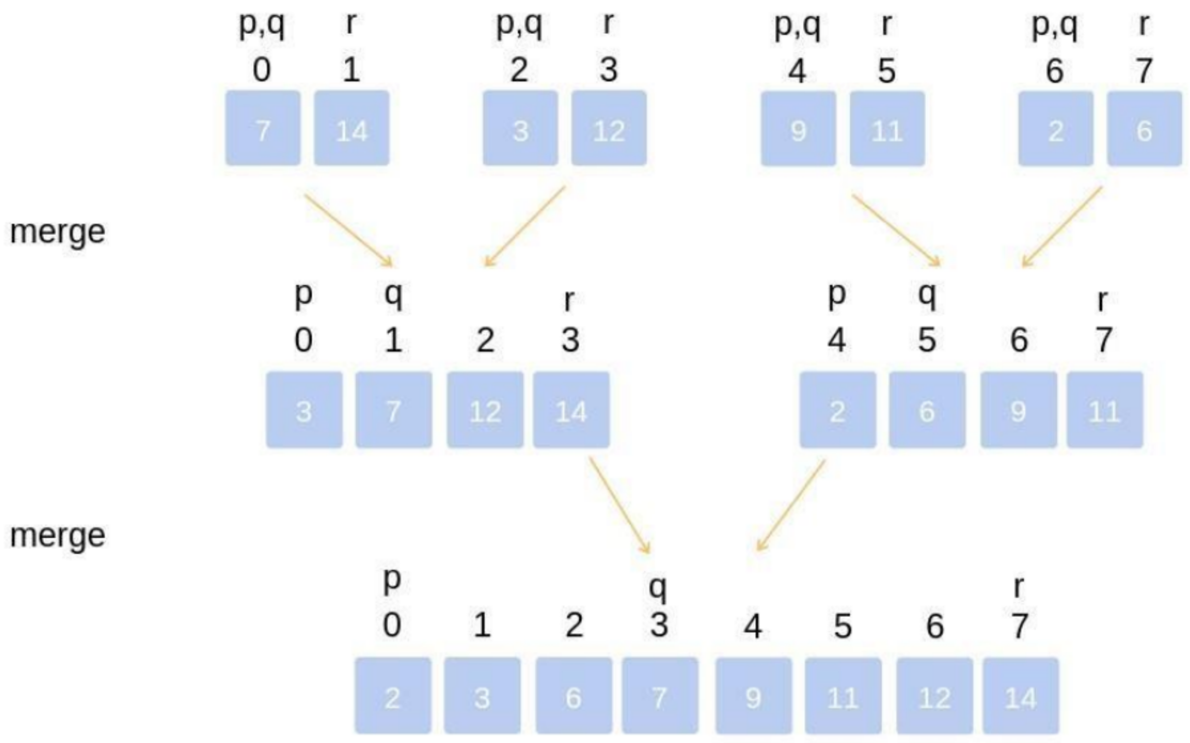
Sample Merge Sort Step by Step Execution:
The top-down merge sort approach is the methodology which uses recursion mechanism. It starts at the Top and proceeds downwards, with each recursive turn asking the same question such as "What is required to be done to sort the array?" and having the answer as, "split the array into two, make a recursive call, and merge the results.", until one gets to the bottom of the array-tree. Example: Let us consider an example to understand the approach better.

1. Divide the unsorted list into n sublists, each comprising 1 element (a list of 1

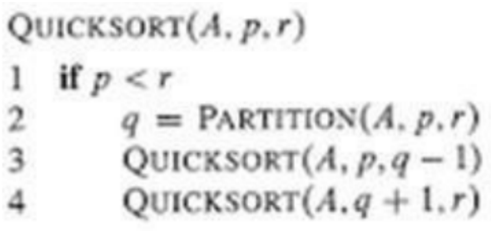
element is supposed sorted).



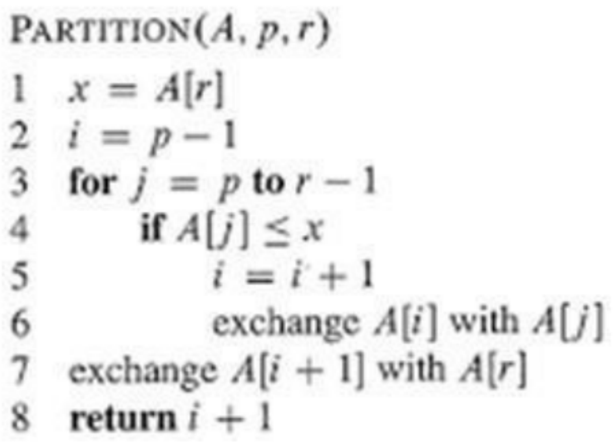
Repeatedly merge sublists to produce newly sorted sublists until there is only 1 sublist remaining. This will be the sorted list.
Merging of two lists done as follows:
The first element of both lists is compared. If sorting in ascending order, the smaller element among two becomes a new element of the sorted list. This procedure is repeated until both the smaller sublists are empty and the newly combined sublist covers all the elements of both the sublists.



Algorithm of Quick sort & Merge sort:



```
QUICKSORT(A, p, r)
1  if $p < r$
2      $q$ = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```



```
PARTITION(A, p, r)
1  $x = A[r]$
2  $i = p - 1$
3  for $j = p$ to $r - 1$
4      if $A[j] \leq x$
5          $i = i + 1$
6          exchange $A[i]$ with $A[j]$
7  exchange $A[i + 1]$ with $A[r]$
8  return $i + 1$
```

```
MERGE-SORT($A, p, r$)
if $p < r$
    $q = \lfloor (p + r)/2 \rfloor$
    MERGE-SORT($A, p, q$)
    MERGE-SORT($A, q + 1, r$)
    MERGE($A, p, q, r$)
```

```
MERGE($A, p, q, r$)
$n_1 = q - p + 1$
$n_2 = r - q$
let $L[1 \,..\, n_1 + 1]$ and $R[1 \,..\, n_2 + 1]$ be new arrays
for $i = 1$ to $n_1$
    $L[i] = A[p + i - 1]$
for $j = 1$ to $n_2$
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ to $r$
    if $L[i] \le R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$
```

Derivation of Analysis Quick sort & Merge sort:

Quick Sort
Worst Case Analysis

Proof:

The partitioning step: at least, $n - 1$ comparisons.

- At each next step for $n \geq 1$, the number of comparisons is one less, so that $T(n) = T(n - 1) + (n - 1)$; $T(1) = 0$. • $T(n) - T(n - 1) = n - 1$:

$T(n)+T(n - 1)+T(n - 2)+...+T(3)+T(2) -T(n - 1)-T(n - 2)-...-T(3)-T(2)-T(1)$

$= (n - 1) + (n - 2) +...+ 2 + 1 - 0$
$T(n)= (n - 1) + (n - 2) +...+ 2 + 1 =(n-1)n/2$

This yields that $T(n) \in O(n\text{^}2)$

Best Case Analysis

Quicksort's best case occurs when the partitions are as evenly balanced as possible: their sizes either are equal or are within 1 of each other. The former case occurs if the subarray has an odd number of elements and the pivot is right in the middle after partitioning, and each partition has (n-1)/2(n−1)/2left parenthesis, n, minus, 1, right parenthesis, slash, 2 elements. The latter case occurs if the subarray has an even number nnn of elements and one partition has n/2n/2n, slash, 2 elements with the other having n/2-1n/2−1n, slash, 2, minus, 1. In either of these cases, each partition has at most n/2n/2n, slash, 2 elements.

Using big-Θ notation, we get the result as: Θ(nlog2n)

Average Case Analysis

For any pivot position i; $i \in \{0, . . . , n - 1\}$:

- Time for partitioning an array : cn
- The head and tail subarrays contain i and $n - 1 - i$ items, respectively:
  $T(n) = cn + T(i) + T(n - 1 - i)$

Average running time for sorting (a more complex recurrence):
$T(n)= 1/n \sum_0^{n-1}(T(i) + T(n- 1 - i)+ cn)$

$= 2 n (T(0) + T(1) + . . . + T(n - 2) + T(n - 1)) + cn$, or $nT(n)=$
$2 (T(0) + T(1) + . . . + T(n - 2) + T(n - 1)) + cn^2$

$(n - 1)T(n - 1) = 2 (T(0) + T(1) + . . . + T(n - 2)) + c(n - 1)^2$ $nT(n)$
$- (n - 1)T(n - 1) = 2T(n - 1) + 2cn - c \approx 2T(n - 1) + 2cn$

Thus, $nT(n) \approx (n + 1)T(n - 1) + 2cn$, or $T(n)/(n+1) = T(n-1)/n + 2c/(n+1)$

$T(n)/(n+1) - T(n-1)/n = 2c/(n+1)$ to get the explicit form:

$T(n)/(n+1) + T(n-1)/n + T(n-2)/(n-1) + . . . + T(2)/3 + T(1)/2 - T(n-1)/n -$
$T(n-2)$ $n-1 - . . . -$ $T(2)/3 - T(1)/2 - T(0)/1$
$= 2c/(n+1) + 2c/n + . . . + 2c/3 + 2c/2$, or

$T(n)/(n+1) = T(0)/1 + 2c (1/2 + 1/3 + . . . + 1/n + 1/n+1) \approx 2c(H_{n+1} - 1)$
$\approx c'$ log n   This yields average case as $T(n) \approx c' (n + 1) \log n \in \Theta(n \log n)$.

Merge Sort
Time Complexity Analysis-
In merge sort, we divide the array into two (nearly) equal halves and solve them recursively using merge sort only.
So, we have-

$$T\left(\frac{n_L}{2}\right) + T\left(\frac{n_R}{2}\right) = 2T\left(\frac{n}{2}\right)$$

n_L = Left Half

n_R = Right Half

$$n_L \approx n_R$$

Finally, we merge these two sub arrays using merge procedure which takes Θ(n) time as explained above.
If T(n) is the time required by merge sort for sorting an array of size n, then the recurrence relation for time complexity of merge sort is-

$$\boxed{T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)}$$

\ Recurrence Relation

On solving this recurrence relation, we get T(n) = Θ(nlogn).

Thus, time complexity of merge sort algorithm is T(n) = Θ(nlogn).

Worst Case Time Complexity [Big-O]:

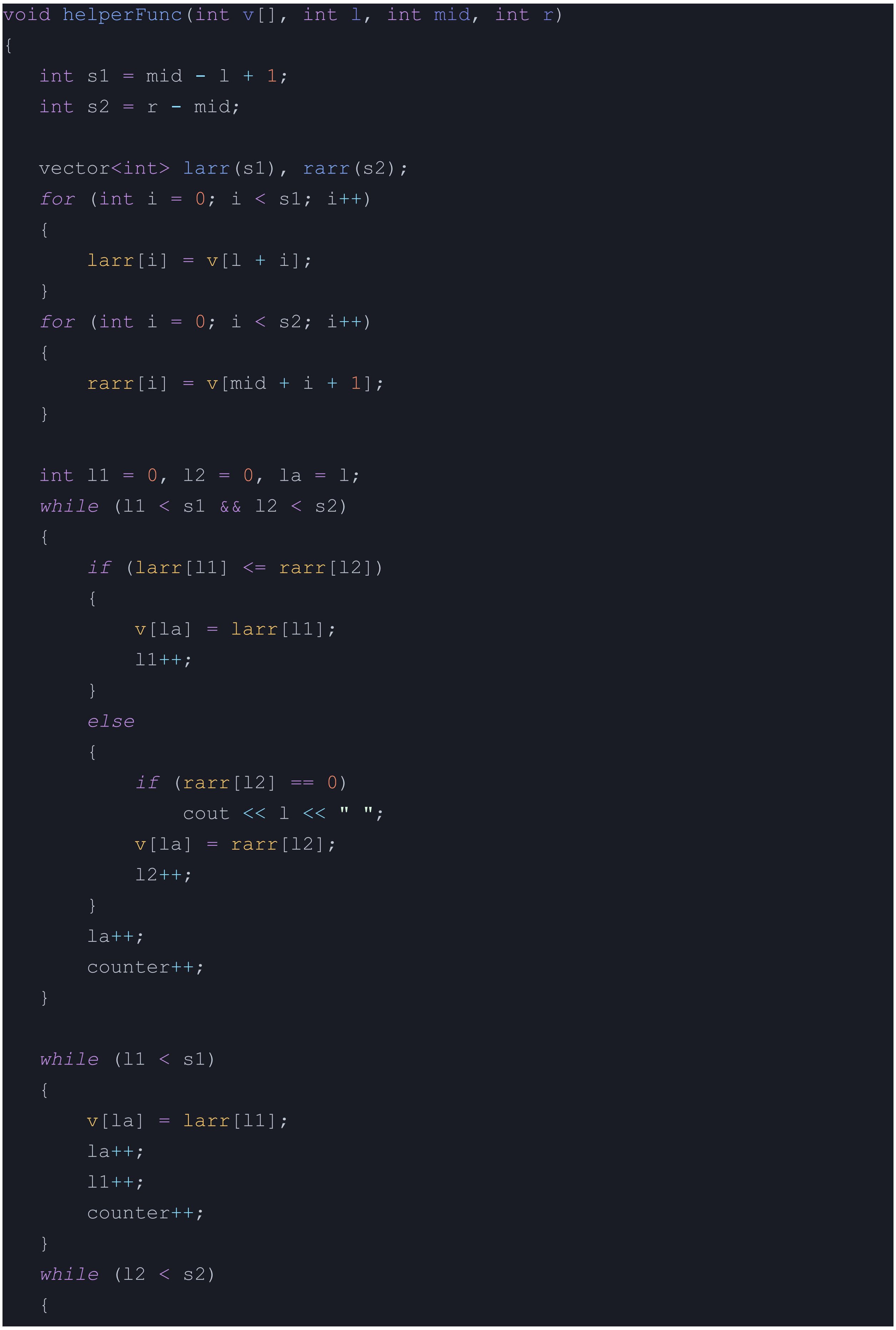
O(n*log n) Best Case Time Complexity

[Big-omega]: O(n*log n) Average Time

Complexity [Big-theta]: O(n*log n)

Program(s) of Quick sort & Merge sort:

```
#include <bits/stdc++.h>
using namespace std;
int counter=0;
```

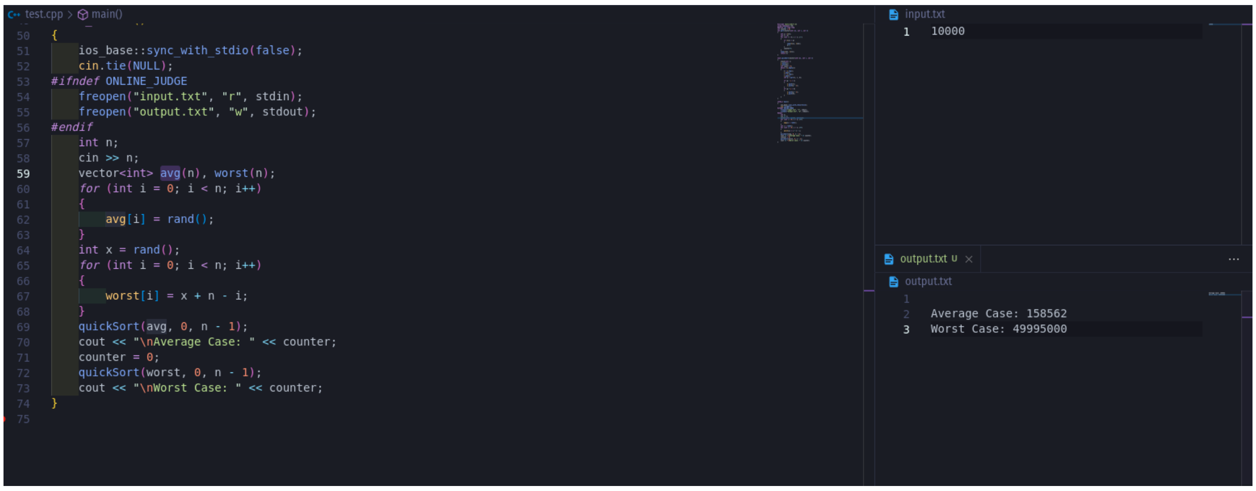


```
void helperFunc(int v[], int l, int mid, int r)
{
    int s1 = mid - l + 1;
    int s2 = r - mid;

    vector<int> larr(s1), rarr(s2);
    for (int i = 0; i < s1; i++)
    {
        larr[i] = v[l + i];
    }
    for (int i = 0; i < s2; i++)
    {
        rarr[i] = v[mid + i + 1];
    }

    int l1 = 0, l2 = 0, la = l;
    while (l1 < s1 && l2 < s2)
    {
        if (larr[l1] <= rarr[l2])
        {
            v[la] = larr[l1];
            l1++;
        }
        else
        {
            if (rarr[l2] == 0)
                cout << l << " ";
            v[la] = rarr[l2];
            l2++;
        }
        la++;
        counter++;
    }

    while (l1 < s1)
    {
        v[la] = larr[l1];
        la++;
        l1++;
        counter++;
    }
    while (l2 < s2)
    {
        v[la] = rarr[l2];
        l2++;
        la++;
        counter++;
    }
}

void mergeSort(int v[], int n)
{
    for (int cur = 1; cur < n; cur *= 2)
    {
        for (int lstart = 0; lstart < n - 1; lstart += 2 * cur)
        {
            int m = min(lstart + cur - 1, n - 1);
            int r = min(lstart + 2 * cur - 1, n - 1);
            helperFunc(v, lstart, m, r);
        }
    }
}

void createBest(int arr[], int n)
{
    int x = rand();
    for (int i = 0; i < n; i++)
    {
        arr[i] = x + i;
    }
}
void createAverage(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        arr[i] = rand();
    }
}
void createWorst(int arr[], int n)
{
    int x = rand();
    for (int i = 0; i < n; i++)
    {
        arr[i] = x + n - i;
    }
}
```

```
int main()
{

   ios_base::sync_with_stdio(false);
   cin.tie(NULL);
#ifndef ONLINE_JUDGE
   freopen("input.txt", "r", stdin);
   freopen("output.txt", "w", stdout);
#endif
   int n;
   cin >> n;
   int best[n], avg[n], worst[n];
   createBest(best, n);
   createWorst(worst, n);
   createAverage(avg, n);
   counter = 0;
   mergeSort(best, n);
   cout << "Best case: " << counter << "\n";
   counter = 0;
   mergeSort(best, n);

   cout << "Average case: " << counter << "\n";
   counter = 0;

   mergeSort(best, n);
   cout << "Worst case: " << counter << "\n";
}
```

Output(o) of Quick sort & Merge sort:

Quick Sort

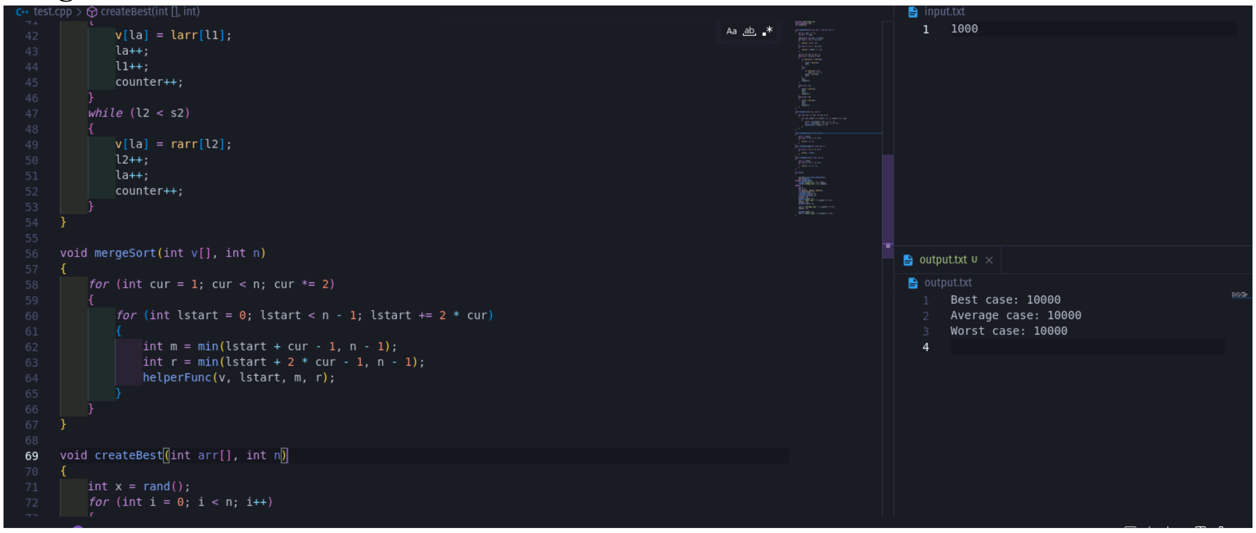


```
{
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
#ifndef ONLINE_JUDGE
    freopen("input.txt", "r", stdin);
    freopen("output.txt", "w", stdout);
#endif
    int n;
    cin >> n;
    vector<int> avg(n), worst(n);
    for (int i = 0; i < n; i++)
    {
        avg[i] = rand();
    }
    int x = rand();
    for (int i = 0; i < n; i++)
    {
        worst[i] = x + n - i;
    }
    quickSort(avg, 0, n - 1);
    cout << "\nAverage Case: " << counter;
    counter = 0;
    quickSort(worst, 0, n - 1);
    cout << "\nWorst Case: " << counter;
}
```

```
10000
```

```

Average Case: 158562
Worst Case: 49995000
```

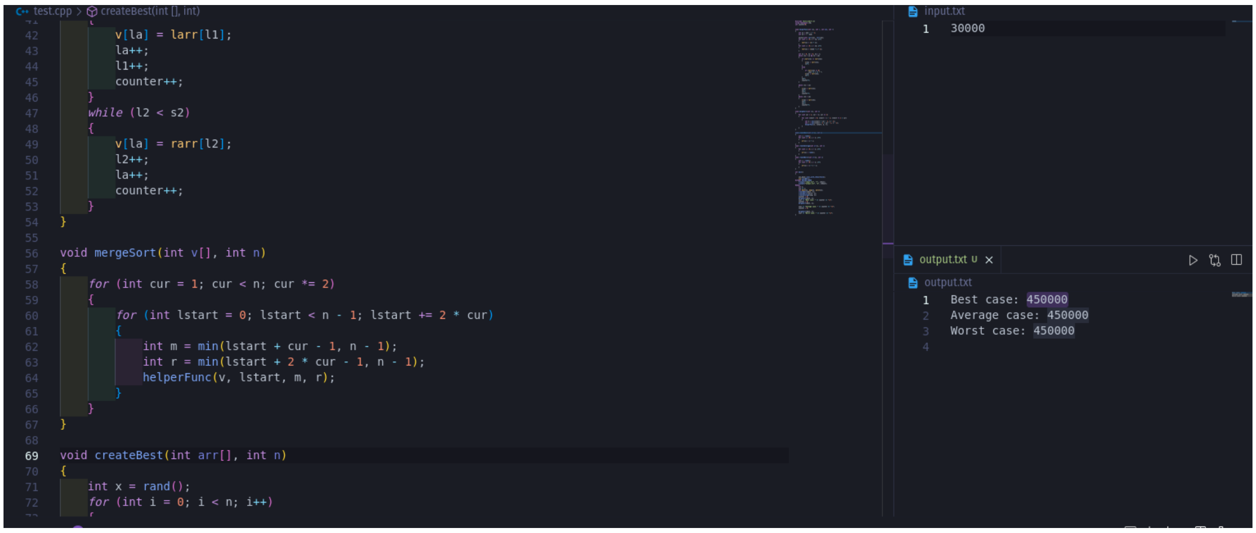
Merge Sort



```
        v[la] = larr[l1];
        la++;
        l1++;
        counter++;
    }
    while (l2 < s2)
    {
        v[la] = rarr[l2];
        l2++;
        la++;
        counter++;
    }
}

void mergeSort(int v[], int n)
{
    for (int cur = 1; cur < n; cur *= 2)
    {
        for (int lstart = 0; lstart < n - 1; lstart += 2 * cur)
        {
            int m = min(lstart + cur - 1, n - 1);
            int r = min(lstart + 2 * cur - 1, n - 1);
            helperFunc(v, lstart, m, r);
        }
    }
}

void createBest(int arr[], int n)
{
    int x = rand();
    for (int i = 0; i < n; i++)
```

```
1000
```

```
Best case: 10000
Average case: 10000
Worst case: 10000
```



```cpp
        v[la] = larr[l1];
        la++;
        l1++;
        counter++;
    }
    while (l2 < s2)
    {
        v[la] = rarr[l2];
        l2++;
        la++;
        counter++;
    }
}

void mergeSort(int v[], int n)
{
    for (int cur = 1; cur < n; cur *= 2)
    {
        for (int lstart = 0; lstart < n - 1; lstart += 2 * cur)
        {
            int m = min(lstart + cur - 1, n - 1);
            int r = min(lstart + 2 * cur - 1, n - 1);
            helperFunc(v, lstart, m, r);
        }
    }
}

void createBest(int arr[], int n)
{
    int x = rand();
    for (int i = 0; i < n; i++)
```

```
30000
```

```
Best case: 450000
Average case: 450000
Worst case: 450000
```

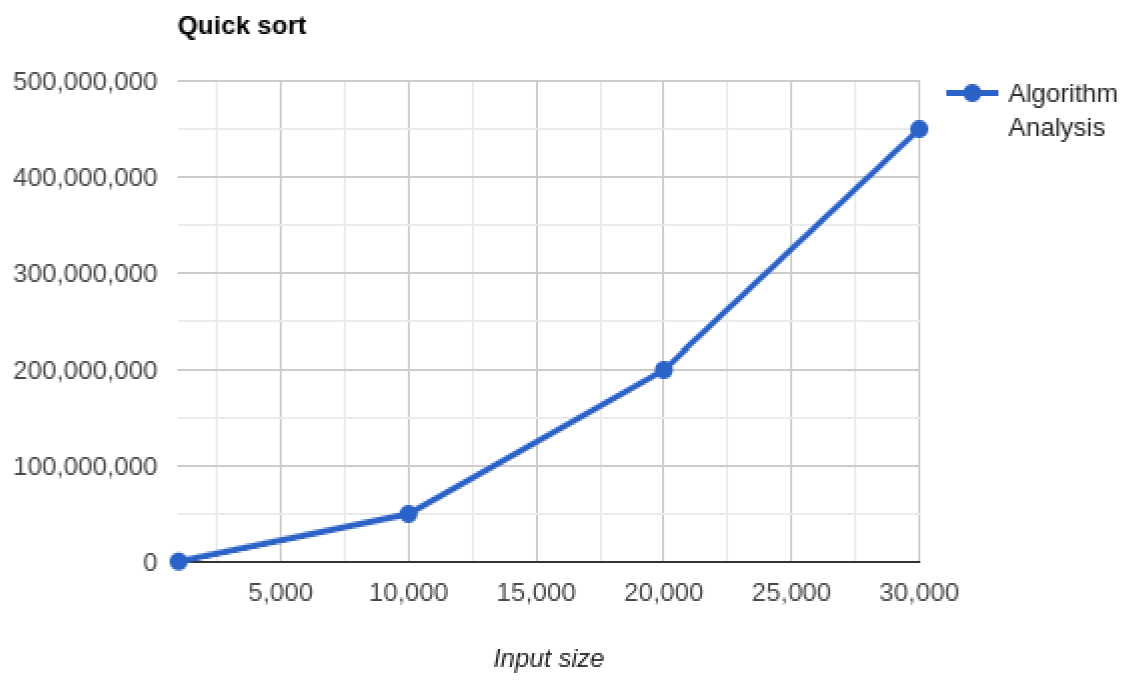
Results:

Time Complexity of Quick sort:

Worst Case Analysis:

Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
	1000	499500	6907
	10000	49995000	92103
	20000	199990000	198069
	30000	449985000	309268

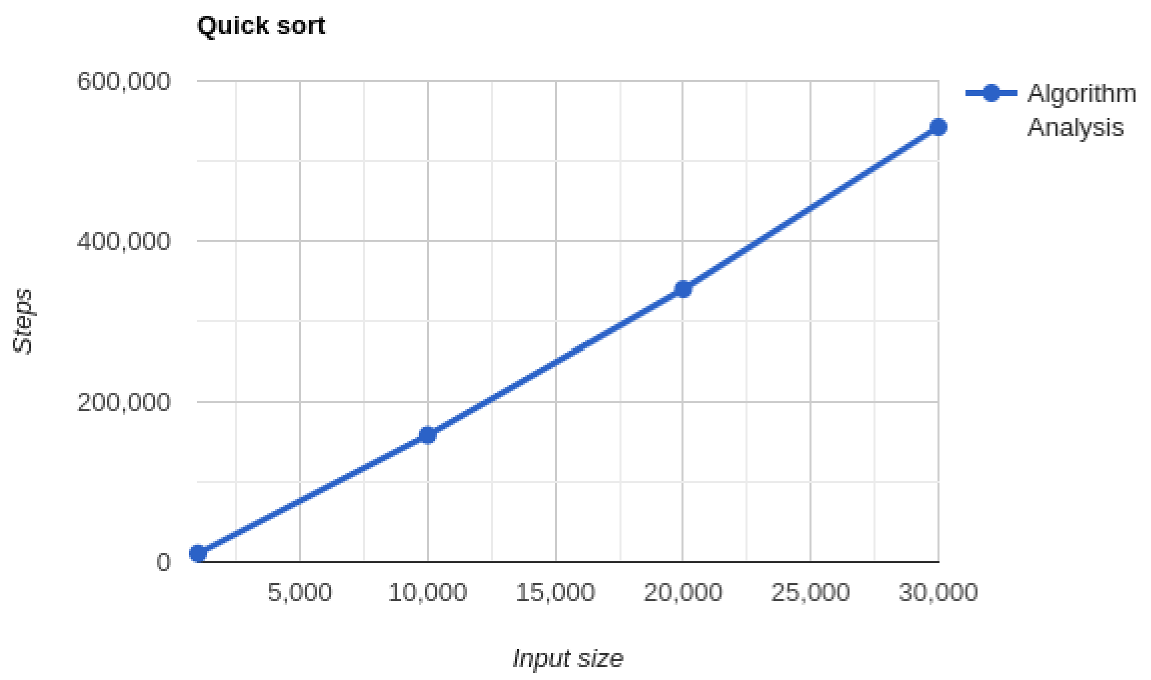
GRAPH:



Best Case Analysis:

Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
	1000	10676	6907
	10000	158562	92103
	20000	339756	198069
	30000	542423	309268

GRAPH

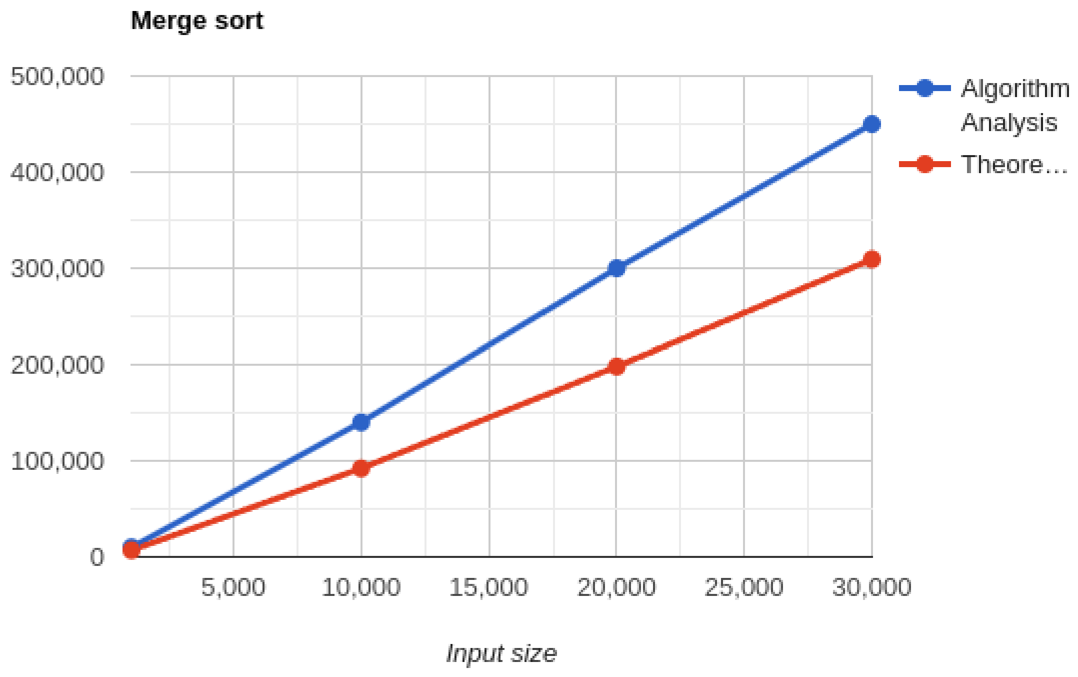

Quick sort
600,000
400,000
200,000
0
Steps
5,000
10,000
15,000
20,000
25,000
30,000
Input size
Algorithm Analysis

Time Complexity of Merge sort:

Worst Case Analysis:

Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
	1000	10000	6907
	10000	140000	92103
	20000	300000	198069
	30000	450000	309268

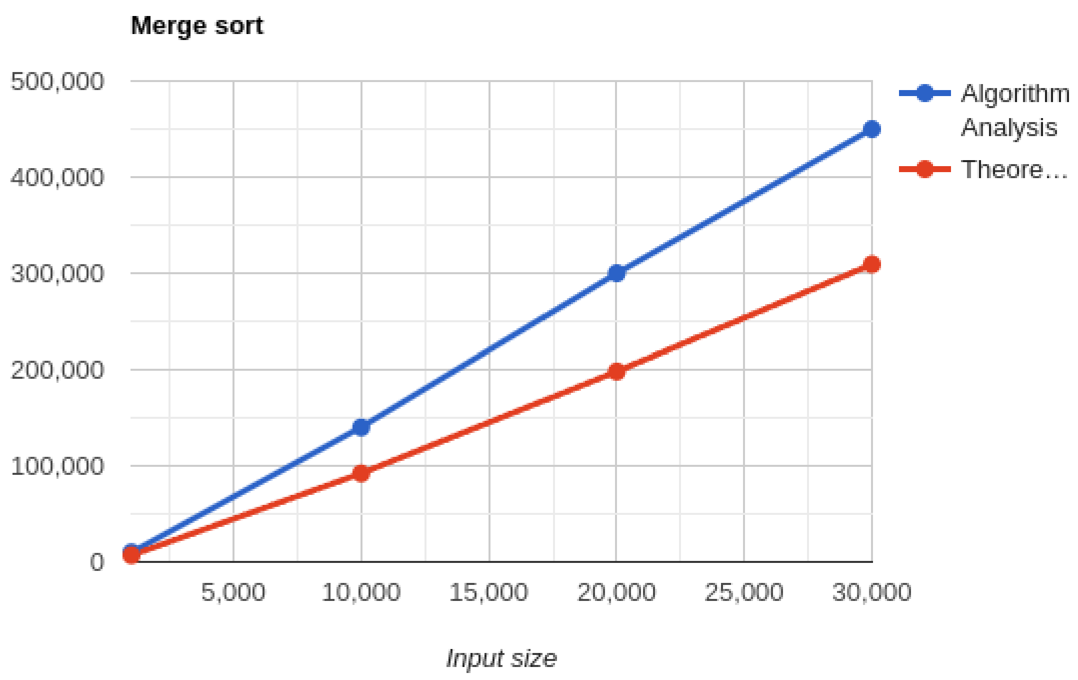
GRAPH:



Best Case Analysis:

Sr. No.	Input size	No: of steps from Algorithm analysis	No: of steps from Theoretical analysis
	1000	10000	6907
	10000	140000	92103
	20000	300000	198069
	30000	450000	309268

GRAPH



Conclusion: (Based on the observations):
The detailed analysis of the time complexity of quick sort and merge sort was done.

Outcome:Analyze time and space complexity of basic algorithms.

References:

1. Richard E. Neapolitan, " Foundation of Algorithms ", 5th Edition 2016, Jones & Bartlett Students Edition
2. Harsh Bhasin , " Algorithms : Design & Analysis", 1st Edition 2013, Oxford Higher education, India
3. T.H. Coreman ,C.E. Leiserson,R.L. Rivest, and C. Stein, " Introduction to algorithms", 3rd Edition 2009, Prentice Hall India Publication
4. Jon Kleinberg, Eva Tardos, " Algorithm Design", 10th Edition 2013, Pearson India Education Services Pvt. Ltd.