Name : Pratik Kanhaiya Rathod
Roll : 321056
PRN : 22010885
Batch : A2

Assignment No. 1

Aim : Implement and Analysis of Quick Sort And Merge Sort .

Objective :

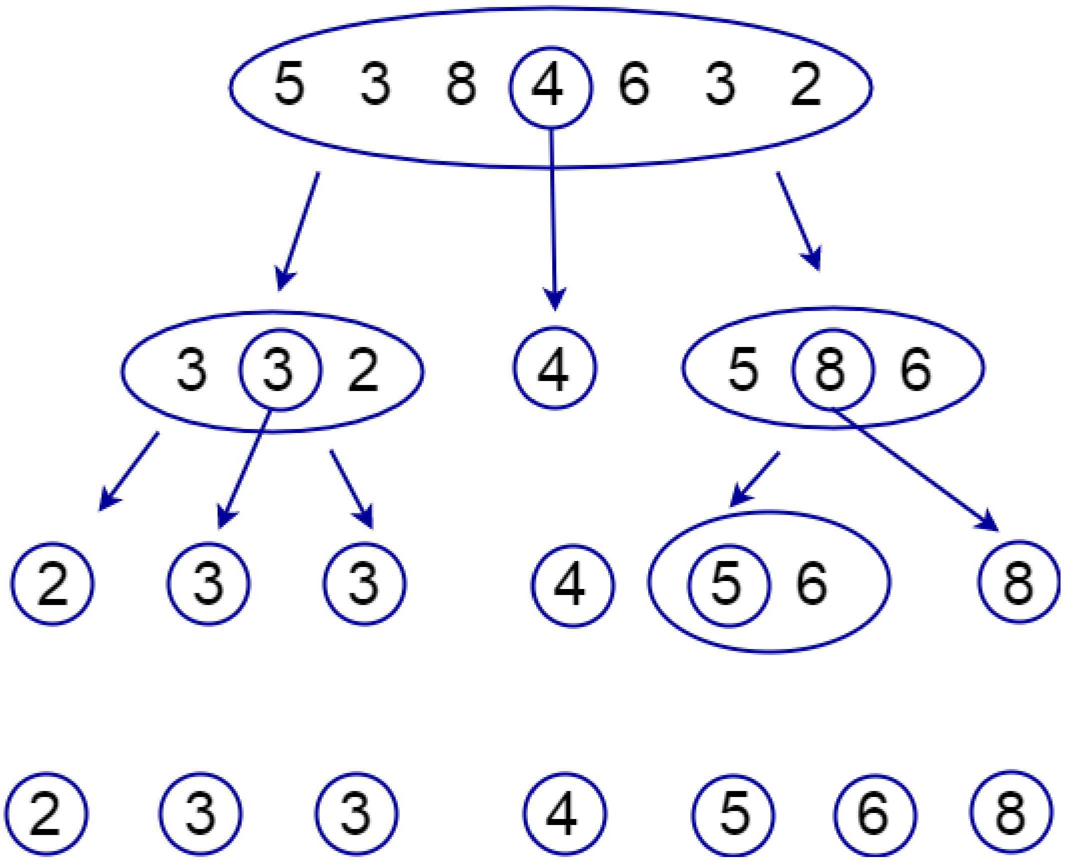
- Sorting algorithm is a method for reorganizing a large number of items into a specific order, such as alphabetical, highest to lowest value or shortest to longest distance.
- Sorting algorithms take lists of items as input data, perform specific operations on those lists and deliver ordered arrays as input.

Theory:

1) **Quick Sort :**

- We first select a pivot element and then divide the original subarray into the left and right subarray.
- After the first division of the array into the left and right subarray, we use the partitioning technique on both the left and the right subarrays(selecting a pivot and repeat the process until all elements are sorted in the given list.
- When all of these operations have been successfully performed we get the sorted resultant array
- **Time complexity:**
  - Best case: O(n*logn)
    This happens if we pick the median of the array as the pivot element every time. The size of subarrays will be half the size of the original array. In this case, the time taken will be O(n*logn).
  - Worst case: O(n^2)
    This happens if we pick the largest or the smallest element of the array as the pivot element every time. The size of the subarray after partitioning will be n-1 and 1.
  - Average case: O(n*logn)
    This is the average time taken for all n! permutations of n elements.

- **Space Complexity**
  - Best case: O(logn)
    This happens when the pivot element's correct position in the partitioned array is at the middle every time. The size of subarrays will be half the size of the original array. In this case, the recursive tree's size will be O(log(n)).
  - Worst case: O(n)
    This happens when the pivot element is the largest or smallest element of the array in every recursive call. The size of the subarray after partitioning will be n-1 and 1. In this case, the size of the recursive tree will be n.



Quick sort

2) Merge Sort :

- In this, we first find the middle element = (left + right)/2 and then divide the array into subarrays.
- After the first division, the subarrays keep on dividing until there is an element left at the end. As we know one element is sorted in itself therefore we combine these lone one-element subarrays to the nearest similar subarray. This process continues until we have two large left and the right subarrays
- During this process there can be some elements in then subarrays that are comparable to each other at the time of merging, so with merging the subarrays we perform the sorting of the subarrays too.
- Now for the two large subarrays, this condition can also arise so we follow the same procedure as before and merge and sort the array. Then, in the end, we get the sorted resultant array of the subarray.
- Time complexity
  - The time complexity of merge sort is O(n log(n)). This is how it's calculated:
  - First, Consider the Number of Rows

    Let's count the number of rows in the dividing step of the algorithm. We'll call the number of rows "num_rows" and the size of the list "n". We keep dividing the list until the size of the list reduces to 1.

    So, n / (2 * 2 * …….. num_rows times) = 1

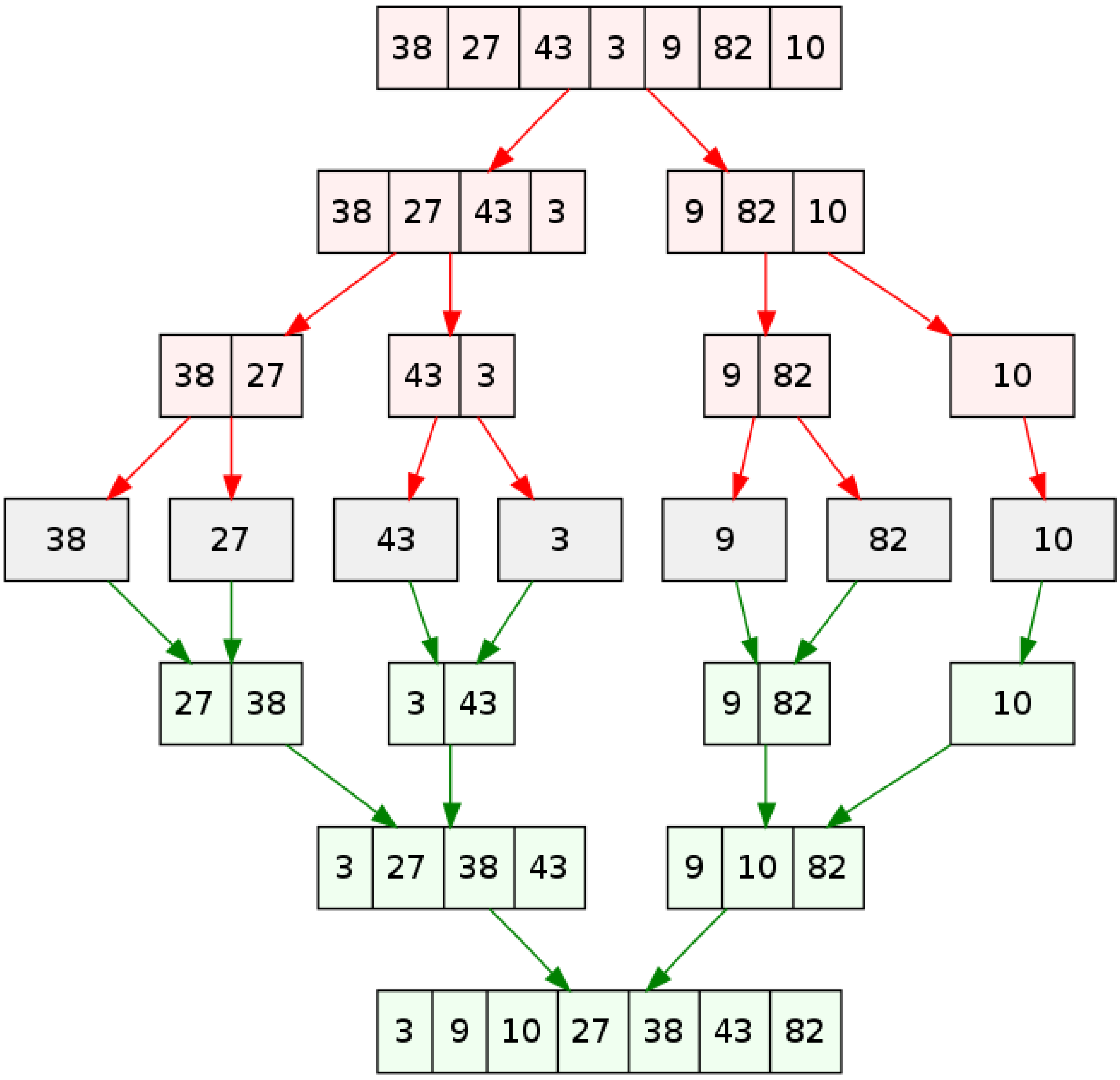
    n = 2 ^ num_rows

    num_rows = log2(n)

    num_rows = O(log(n))

- Space Complexity
  - The space complexity of merge sort is **O(n).**
  - Space complexity only takes into account the auxiliary space we use to solve the problem. Space used to store the input information doesn't matter here.
  - We used auxiliary space only for creating a temporary array to hold the result of merged lists. So, the space required is equal to the sum of the size of the lists being merged.

- This, in the worst case, can be equal to n. Hence, the space complexity is O(n).



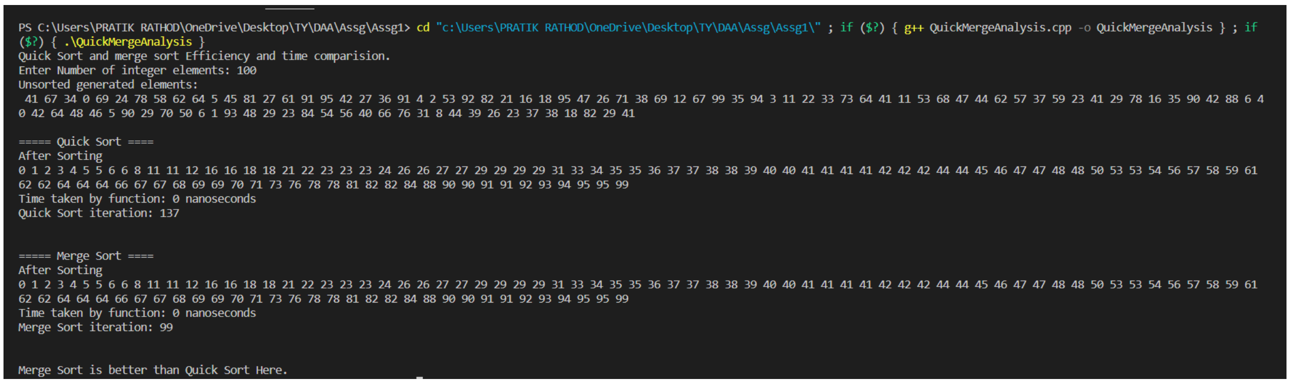
Merge sort

Code :

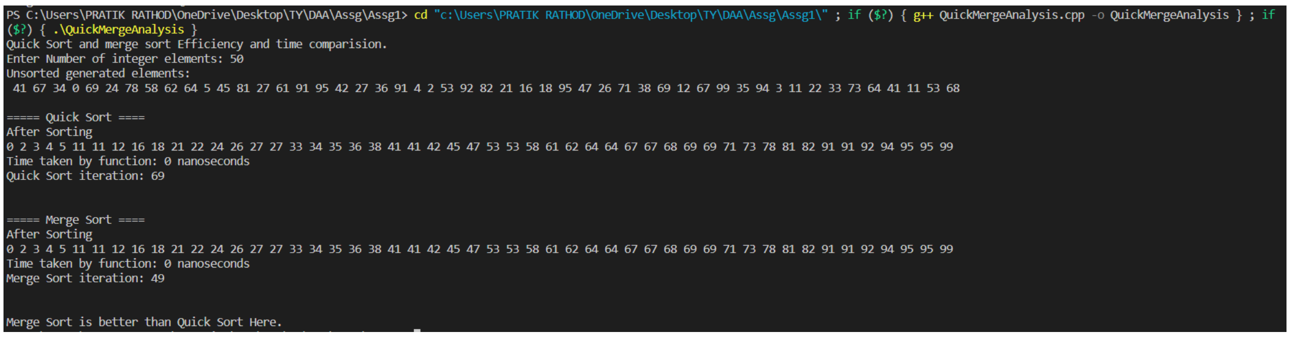
```
#include <algorithm>
#include <cstdlib>
#include <chrono>
#include <iostream>
#include <vector>
using namespace std;
using namespace std::chrono;
/*
Name : Pratik K. Rathod
Roll No : 321056
PRN No: 22010885
*/

int cnt1=0, cnt2=0;

void PrintArr(int arr[], int n){
    for(int i = 0; i < n; i++){
        cout<<arr[i]<<" ";
    }
}
/* -------- Quick Sort Program Starts Here -------- */
int Partition(int arr[], int start, int end)
{
  int pivot = arr[end];
  int i = (start-1);
  for(int j = start; j<= end-1; j++){
    if(arr[j] <= pivot){
        i++;
        swap(arr[i], arr[j]);
    }
  }
  swap(arr[i+1], arr[end]);
  return (i+1);
}

void QuickSort(int arr[], int start, int end)
{
  if (start < end)
  {
    cnt1++;
    int index = Partition(arr, start, end);
    QuickSort(arr, start, index - 1);
    QuickSort(arr, index + 1, end);
  }
  else{
    cnt1++;
  }
}

/* -------- Merge Sort Program Starts Here -------- */
void Merge(int arr[], int start, int mid, int end){
    int temp[end + 1];
    int i = start;
    int j = mid + 1;
    int k = 0;

    while (i <= mid && j <= end)
    {
        if (arr[i] <= arr[j])
            temp[k++] = arr[i++];
        else
            temp[k++] = arr[j++];
    }
    while (i <= mid)
        temp[k++] = arr[i++];
    while (j <= end)
        temp[k++] = arr[j++];
    k--;
    while (k >= 0)
    {
        arr[k + start] = temp[k];
        k--;
    }
}

void mergeSort(int arr[], int start, int end){

    if(start<end){

        int mid = (start+end) / 2;
        mergeSort(arr, start, mid);
        mergeSort(arr, mid+1, end);
        Merge(arr, start, mid, end);
        cnt2++;
    }
}

/* ----- Main Function ----- */
int main(){
    int n;

    cout<<"Quick Sort and merge sort Efficiency and time comparision."<<endl;
    cout<<"Enter Number of integer elements: ";
    cin>>n;
    int arr[n];
```

```cpp
    for(int i = 0; i < n; i++){
        arr[i] = rand() % 100;
    }
    cout<< "Unsorted generated elements: \n ";

    for(int i =0; i<n;i++){
        cout<< arr[i]<<" ";
    }

    cout<<"\n\n===== Quick Sort ====";

    auto start1 = high_resolution_clock::now();
    QuickSort(arr, 0, n - 1);
    auto stop1 = high_resolution_clock::now();
    auto duration1 = duration_cast<nanoseconds>(stop1 - start1);

    cout << "\nAfter Sorting" << endl;
    PrintArr(arr, n);
    cout << "\nTime taken by function: " << duration1.count() << " nanoseconds"
<< endl;

    cout << "Quick Sort iteration: " << cnt1 << endl;


    cout<<"\n\n===== Merge Sort ====";
    auto start2 = high_resolution_clock::now();
    mergeSort(arr, 0, n - 1);
    auto stop2 = high_resolution_clock::now();
    auto duration2 = duration_cast<nanoseconds>(stop2 - start2);

    cout << "\nAfter Sorting" << endl;
    PrintArr(arr, n);
    cout << "\nTime taken by function: " << duration2.count() << " nanoseconds"
<< endl;
    cout << "Merge Sort iteration: " << cnt2 << endl;

    if(cnt1<cnt2) cout<<"\n\nQuick Sort Is better than Merge Sort Here.";
    else cout<<"\n\nMerge Sort is better than Quick Sort Here.\n";

}
```

Output :



```
PS C:\Users\PRATIK RATHOD\OneDrive\Desktop\TY\DAA\Assg\Assg1> cd "c:\Users\PRATIK RATHOD\OneDrive\Desktop\TY\DAA\Assg\Assg1\" ; if ($?) { g++ QuickMergeAnalysis.cpp -o QuickMergeAnalysis } ; if
($?) { .\QuickMergeAnalysis }
Quick Sort and merge sort Efficiency and time comparision.
Enter Number of integer elements: 100
Unsorted generated elements:
 41 67 34 0 69 24 78 58 62 64 5 45 81 27 61 91 95 42 27 36 91 4 2 53 92 82 21 16 18 95 47 26 71 38 69 12 67 99 35 94 3 11 22 33 73 64 41 11 53 68 47 44 62 57 37 59 23 41 29 78 16 35 90 42 88 6 4
0 42 64 48 46 5 90 29 70 50 6 1 93 48 29 23 84 54 56 40 66 76 31 8 44 39 26 23 37 38 18 82 29 41

===== Quick Sort ====
After Sorting
0 1 2 3 4 5 5 6 6 8 11 11 12 16 16 18 18 21 22 23 23 23 24 26 26 27 27 29 29 29 29 31 33 34 35 35 36 37 37 38 38 39 40 40 41 41 41 41 42 42 42 44 44 45 46 47 47 48 48 50 53 53 54 56 57 58 59 61
62 62 64 64 64 66 67 67 68 69 69 70 71 73 76 78 78 81 82 82 84 88 90 90 91 91 92 93 94 95 95 99
Time taken by function: 0 nanoseconds
Quick Sort iteration: 137


===== Merge Sort ====
After Sorting
0 1 2 3 4 5 5 6 6 8 11 11 12 16 16 18 18 21 22 23 23 23 24 26 26 27 27 29 29 29 29 31 33 34 35 35 36 37 37 38 38 39 40 40 41 41 41 41 42 42 42 44 44 45 46 47 47 48 48 50 53 53 54 56 57 58 59 61
62 62 64 64 64 66 67 67 68 69 69 70 71 73 76 78 78 81 82 82 84 88 90 90 91 91 92 93 94 95 95 99
Time taken by function: 0 nanoseconds
Merge Sort iteration: 99


Merge Sort is better than Quick Sort Here.
```



```
PS C:\Users\PRATIK RATHOD\OneDrive\Desktop\TY\DAA\Assg\Assg1> cd "c:\Users\PRATIK RATHOD\OneDrive\Desktop\TY\DAA\Assg\Assg1\" ; if ($?) { g++ QuickMergeAnalysis.cpp -o QuickMergeAnalysis } ; if
($?) { .\QuickMergeAnalysis }
Quick Sort and merge sort Efficiency and time comparision.
Enter Number of integer elements: 50
Unsorted generated elements:
 41 67 34 0 69 24 78 58 62 64 5 45 81 27 61 91 95 42 27 36 91 4 2 53 92 82 21 16 18 95 47 26 71 38 69 12 67 99 35 94 3 11 22 33 73 64 41 11 53 68

===== Quick Sort ====
After Sorting
0 2 3 4 5 11 11 12 16 18 21 22 24 26 27 27 33 34 35 36 38 41 41 42 45 47 53 53 58 61 62 64 64 67 67 68 69 69 71 73 78 81 82 91 91 92 94 95 95 99
Time taken by function: 0 nanoseconds
Quick Sort iteration: 69


===== Merge Sort ====
After Sorting
0 2 3 4 5 11 11 12 16 18 21 22 24 26 27 27 33 34 35 36 38 41 41 42 45 47 53 53 58 61 62 64 64 67 67 68 69 69 71 73 78 81 82 91 91 92 94 95 95 99
Time taken by function: 0 nanoseconds
Merge Sort iteration: 49


Merge Sort is better than Quick Sort Here.
```

Discussion :

The time complexity of merge sort is always O(n log n), while the time complexity of quicksort varies between O(n log n) in the best case to O(n2) in the worst case. Based on the results of our analysis, here's our conclusion:

Conclusion :

- When to use quicksort: Quicksort performs better in situations where the dataset is random and doesn't have characteristics like being almost sorted or having a lot of duplicate entries.
- When to use merge sort: Though merge sort takes a little longer time than quicksort for a random dataset, it does have the advantage of being consistent with respect to the time taken for sorting, no matter the dataset's characteristics.