Name	ARYAMAN AGARWAL
UID no.	2021700002
Experiment No.	2

AIM:	Experiment based on divide and conquer approach
Program 2	
PROBLEM STATEMENT :	For this experiment, you need to implement two sorting algorithms namely Quicksort and Merge sort methods. Compare these algorithms based on time and space complexity. Time required for sorting algorithms can be performed using high_resolution_clock::now() under namespace std::chrono. You have to generate 1,00,000 integer numbers using C/C++ Rand function and save them in a text file. Both the sorting algorithms uses these 1,00,000 integer numbers as input as follows. Each sorting algorithm sorts a block of 100,200,300,...,100000 integer numbers with array indexes numbers A[0..99], A[100..199], A[200..299],…, A[99900..99999]. You need to use high_resolution_clock::now() function to find the time required for 100, 200, 300…. 100000 integer numbers. Finally, compare two algorithms namely Quicksort and Merge sort by plotting the time required to sort integers using LibreOffice Calc/MS Excel. The x-axis of 2-D plot represents the block no. of 1000 blocks. The y-axis of 2-D plot represents the tunning time to sort 1000 blocks of 100,200,300,...,100000 integer numbers.
ALGORITHM/THEORY:	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 Quick Sort Partition 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

Merge Sort Algorithm:
Step 1 - If only one element is there then return
Step 2 – Find the middle element of the array
Step 3 – Recursively call the merge sort for left half of the array.
Step 4 – Recursively call the merge sort for right half of the array.
Step 5 – Merge the smaller parts of the array in a sorted manner.

In order to implement solution for this problem statement I used the rand() function to generate 100000 random numbers in C and saved them to a text file. Then I sorted these numbers using both the algorithms. Alongside, the time taken for sorting each number as also calculated and lastly a graph was plotted in which the Y-axis represents the running time to sort 100 blocks.

PROGRAM:

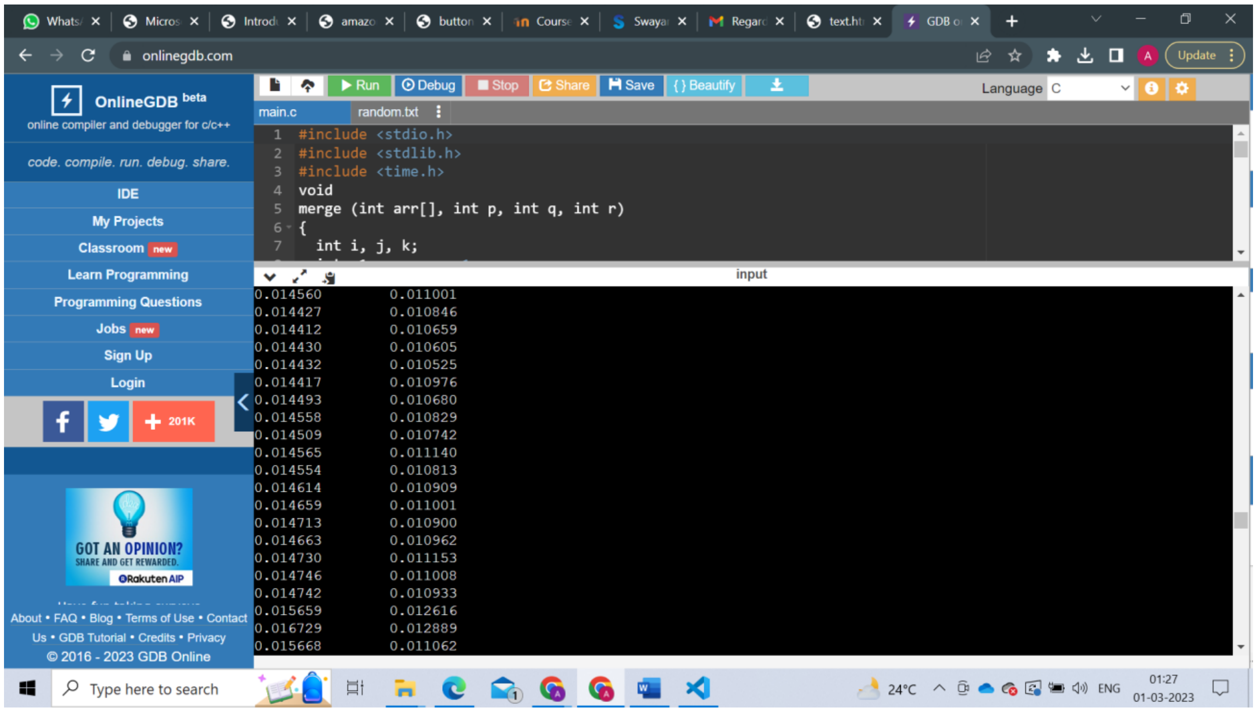
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void merge(int arr[], int p, int q, int r)
{
int i, j, k;
int n1 = q - p + 1;
int n2 = r - q;
int L[n1], R[n2];
for (i = 0; i < n1; i++)
L[i] = arr[p + i];
for (j = 0; j < n2; j++)
R[j] = arr[q + 1 + j];
i = 0;
j = 0;
k = p;
while (i < n1 && j < n2)
{
```

```
if (L[i] <= R[j])
{
arr[k] = L[i];
i++;
}
else
{
arr[k] = R[j];
j++;
}
k++;
}
while (i < n1)
{
arr[k] = L[i];
i++;
k++;
}
while (j < n2)
{
arr[k] = R[j];
j++;
k++;
}
}
void mergeSort(int arr[], int l, int r)
{
if (l < r)
{
int m = l + (r - l) / 2;
mergeSort(arr, l, m);
mergeSort(arr, m + 1, r);
merge(arr, l, m, r);
}
}
int quicksort(int a[], int start, int end)
{
```

```
int pivot = a[end];
int i = (start - 1);
for (int j = start; j <= end - 1; j++)
{
if (a[j] < pivot)
{
i++;
int t = a[i];
a[i] = a[j];
a[j] = t;
}
}
int t = a[i + 1];
a[i + 1] = a[end];
a[end] = t;
return (i + 1);
}
double quick(int a[], int start, int end)
{
if (start < end)
{
int p = quicksort(a, start, end);
quick(a, start, p - 1);
quick(a, p + 1, end);
}
}
int main()
{
double qust,mest;
srand(time(0));
FILE *fp;
fp = fopen("random.txt", "w");
for (int i = 0; i < 100000; i++)
{
fprintf(fp, "%d\n", rand() % 100000);
}
int upper_limit = 100;
```

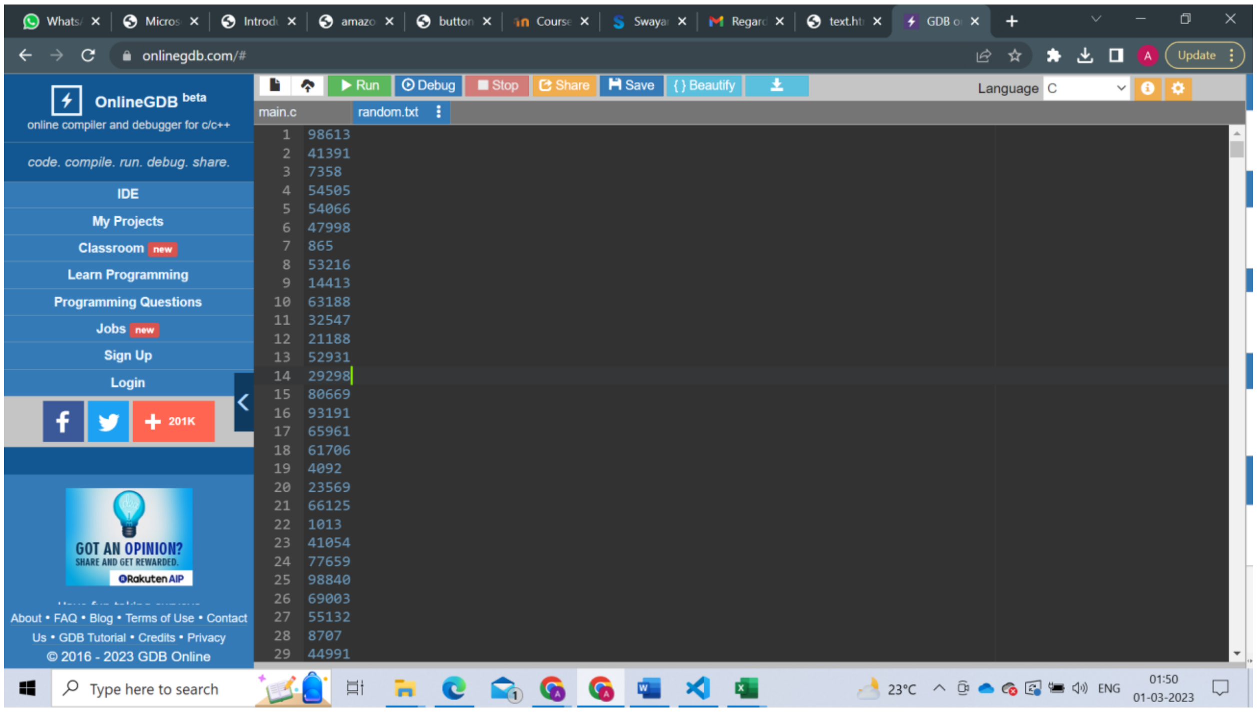
```
fclose(fp);
printf("Merge Sort\tQuick Sort\n");
for (int i = 0; i < 1000; i++)
{
fp = fopen("random.txt", "r");
int arr1[upper_limit], arr2[upper_limit], temp_num;
for (int j = 0; j < upper_limit; j++)
{
fscanf(fp, "%d", &temp_num);
arr1[j] = temp_num;
arr2[j] = temp_num;
}
fclose(fp);
clock_t t;
t = clock();
mergeSort(arr2,0,upper_limit-1);
t = clock() - t;
mest = ((double)t) / CLOCKS_PER_SEC;
clock_t t1;
t1 = clock();
qust=quick(arr1,0,upper_limit-1);
t1 = clock() - t1;
qust = ((double)t1) / CLOCKS_PER_SEC;
printf("%lf\t%lf\n", mest, qust);
fflush(stdout);
upper_limit += 100;
}
return 0;
}
```

OUTPUT AND RESULT ANALYSIS:

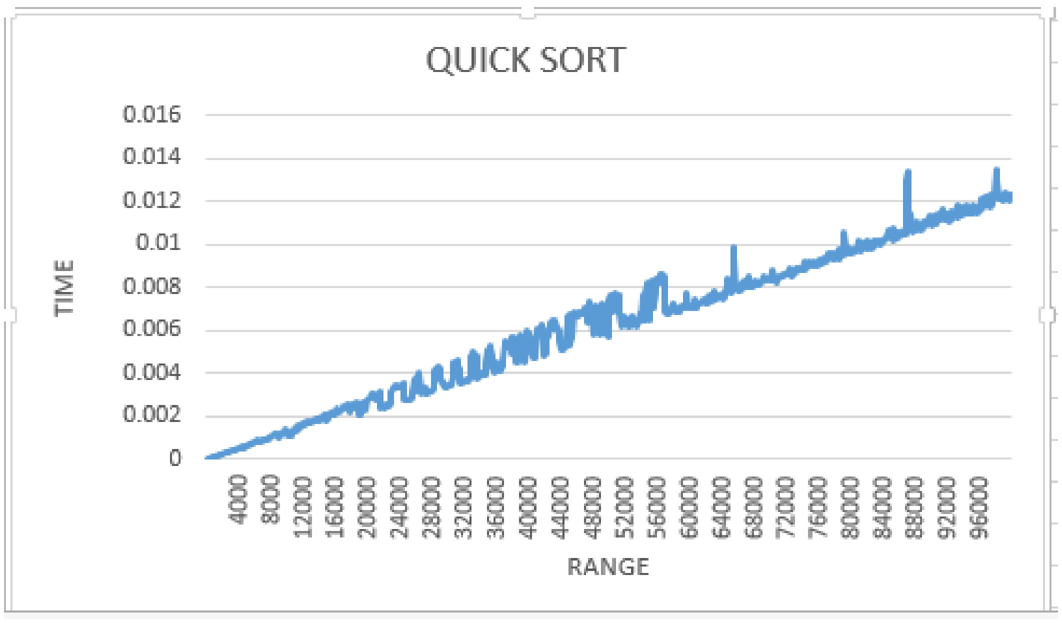


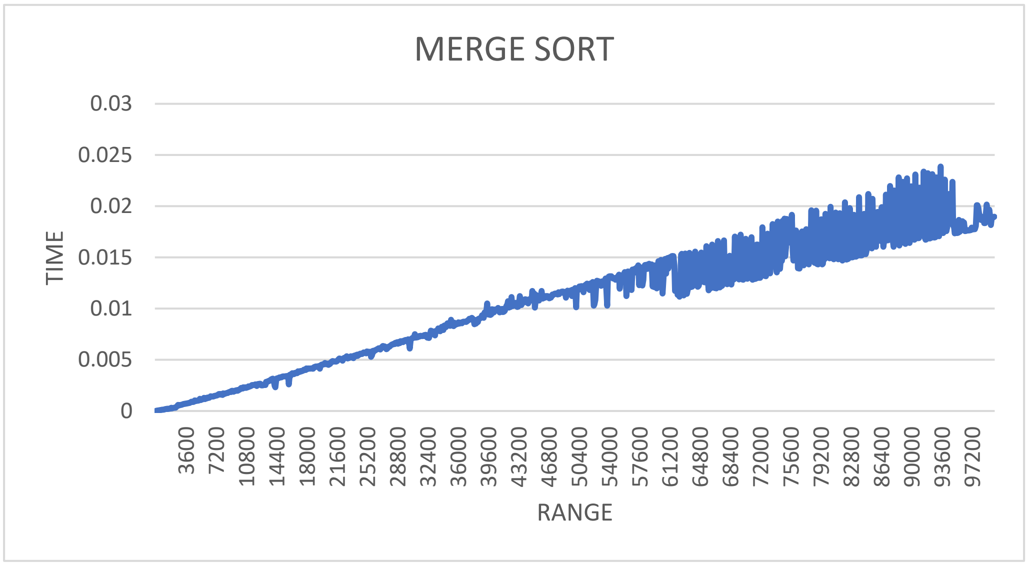
```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void
merge (int arr[], int p, int q, int r)
{
  int i, j, k;
```

```
0.014560        0.011001
0.014427        0.010846
0.014412        0.010659
0.014430        0.010605
0.014432        0.010525
0.014417        0.010976
0.014493        0.010680
0.014558        0.010829
0.014509        0.010742
0.014565        0.011140
0.014554        0.010813
0.014614        0.010909
0.014659        0.011001
0.014713        0.010900
0.014663        0.010962
0.014730        0.011153
0.014746        0.011008
0.014742        0.010933
0.015659        0.012616
0.016729        0.012889
0.015668        0.011062
```



```
98613
41391
7358
54505
54066
47998
865
53216
14413
63188
32547
21188
52931
29298
80669
93191
65961
61706
4092
23569
66125
1013
41054
77659
98840
69003
55132
8707
44991
```





We observe that quick sort beats merge sort for almost all input sizes i.e. quick sort is quicker and takes less time than merge sort. Time taken by quicksort ranged from 0 to 0.014 whereas for merge sort it ranged from 0 to 0.025.

CONCLUSION:	• Implemented Quick Sort and Merge Sort in c. • Quick Sort is a in-place sorting algorithm and has Space Complexity of O(1).

	• For merging of sorted sub-arrays, it needs a temporary array with the size equal to the number of input elements, therefore it has Space Complexity of O(n). • Time Complexity(Average case): Quick Sort – O(n logn) Merge Sort – O(n logn)