NAME:	Prithvi Singh
UID:	2022301014
SUBJECT	DAA
EXPERIMENT NO:	2
AIM:	To implement and compare merge and quick sort.
ALGORITHM	• Merge Sort Algorithm: 1. If the array has only one element or is empty, it is already sorted. 2. Divide the array into two halves: left and right. 3. Recursively sort the left half using Merge Sort. 4. Recursively sort the right half using Merge Sort. 5. Merge the two sorted halves by comparing the first elements of each half and adding the smaller one to a new temporary array, repeating until one of the halves is completely merged. 6. Copy the remaining elements from the other half into the temporary array. 7. Copy the temporary array back into the original array. • Quick Sort Algorithm: 1. Pick a pivot element from the array (usually the first or last element). 2. Partition the array around the pivot element by rearranging the elements so that all elements smaller than the pivot come before it, and all elements larger than the pivot come after it. 3. Recursively apply steps 1 and 2 to the sub-array of elements smaller than the pivot, and the sub-array of elements larger than the pivot. 4. The base case of the recursion is when the sub-array has fewer than two elements, in which case it is already sorted.

PROGRAM:

1. 2exp.cpp

```cpp
#include<iostream>
using namespace std;

int main()
{
    for(int i =0; i<100000; i++)
    {
        cout<<rand() << "\n";
    }
}
```

2. Main.cpp

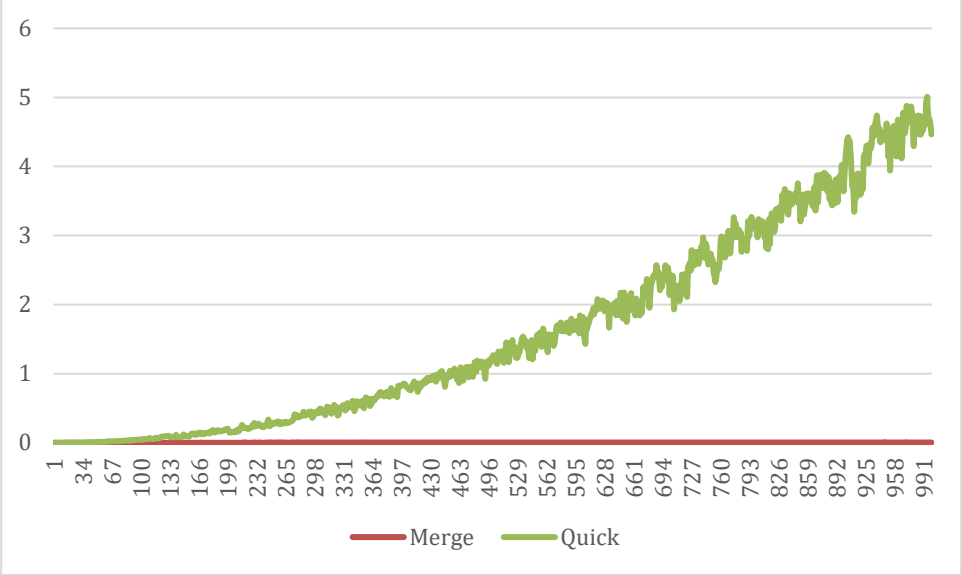
```cpp
#include <iostream>
#include<bits/stdc++.h>
using namespace std;

void merge(int arr[], int p, int q, int r)
{
  int n1 = q - p + 1;
  int n2 = r - q;

  int L[n1], M[n2];

  for (int i = 0; i < n1; i++)
    L[i] = arr[p + i];
  for (int j = 0; j < n2; j++)
    M[j] = arr[q + 1 + j];


  int i, j, k;
  i = 0;
  j = 0;
  k = p;

  while (i < n1 && j < n2) {
    if (L[i] <= M[j]) {
      arr[k] = L[i];
      i++;
```

```
    } else {
      arr[k] = M[j];
      j++;
    }
    k++;
  }

  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }

  while (j < n2) {
    arr[k] = M[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);
  }
}

void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int divide(int array[], int low, int high) {


  int pivot = array[high];
```

```
  int i = (low - 1);


  for (int j = low; j < high; j++) {
    if (array[j] <= pivot) {

      i++;


      swap(&array[i], &array[j]);
    }
  }


  swap(&array[i + 1], &array[high]);


  return (i + 1);
}

void quickSort(int array[], int low, int high) {
  if (low < high) {


    int pi = divide(array, low, high);


    quickSort(array, low, pi - 1);


    quickSort(array, pi + 1, high);
  }
}




int main()
{
     int i,i1;
     int arr[100000];
```

```
    fp = fopen("2nd.txt", "r");


        for ( i = 0; i < 100000; i++)
        {
            fscanf(fp,"%d\n",&arr[i]);
        }



    int num = 100;
    for (int i = 0; i <1000; i++)
    {
        t1 = clock();
        mergeSort(arr,0, num);
        t2 = clock();
        t3 = clock();
        quickSort(arr,0, num);
        t4 = clock();
        double mergetime = double(t2 - t1) / double(CLOCKS_PER_SEC);
        double quicktime = double(t4 - t3) / double(CLOCKS_PER_SEC);
        cout << endl;
        cout << i+1 <<" "<<fixed << mergetime << setprecision(5) <<
'\t';
        cout << fixed << quicktime << setprecision(5);

        num += 100;
    }
    fin.close();
    return 0;
}
```

Graphs and Observation:



- **Merge Sort:**

Here as we can see in the graph for merge sort that as the values change time required for sorting changes too. In this sorting algorithm we can see that according to values there isn't any drastic changes throughout execution of the program. Majorly the whole sorting algorithm is executed within 1 second. It takes less time as compared to sorting the same values using quick sort algorithm. Also, the running time differs from system to system as each system has different characteristics and specifications.

- **Quick Sort:**

Here as we can see in the graph for quick sort that as the values change time required for sorting changes too. This can be understood by seeing throttling at different stages in the execution. Sometimes also due to different values in each block the time to sort those blocks also differs. We can also acknowledge from the graph that quick sort for the most amount of running time takes more time than compared to merge sort. Despite of throttling we can observe that graph moves almost evenly for most amount of running time. Also, the running

	time differs from system to system as each system has different characteristics and specifications.
Conclusion:	The experiment on finding the running time of merge sort and quick sort shows that the running time of both algorithms is dependent on the size of the input data. Merge sort is basically way quicker than quick sort.