NAME:	Shubham Solanki
UID:	2022301015
SUBJECT	Design and Analysis of Algorithm
EXPERIMENT NO:	2
AIM:	Experiment based on divide and conquer approach
Algorithm	**1. Quick sort-** **QUICKSORT(A, p, r)** **1.** if p < r **2.** // Partition the subarray around the pivot, which ends up in A[q]. **3.** q = PARTITION(A, p, r) **4.** QUICKSORT(A, p, q – 1) // recursively sort the low side **5.** QUICKSORT(A, q + 1, r) **PARTITION(A, p, r)** **1.** x = A[r] // the pivot **2.** i = p – 1 // highest index into the low side **3.** for j = p to r – 1 // process each element other than the pivot **4.** if A[j] ≤ x // does this element belong on the low side? **5.** i = i + 1 // index of a new slot in the low side **6.** exchange A[i] with A[j] // put this element there

7. exchange A[i + 1] with A[r]// pivot goes just to the right of the low side

8. return i + 1 // new index of the pivot

2. merge sort-

```
MERGE(A, p, q, r)
1 nL = q – p + 1 // length of A[p : q]
2 nR = r – q // length of A[q + 1 : r]
3 let L[0 : nL – 1] and R[0 : nR – 1] be new arrays
4 for i = 0 to nL – 1 // copy A[p : q] into L[0 : nL –1]
5 L[i] = A[p + i]
6 for j = 0 to nR – 1 // copy A[q + 1 : r] into R[0 : nR – 1]
7 R[j] = A[q + j + 1]
8 i = 0 // i indexes the smallest remaining element in L
9 j = 0 // j indexes the smallest remaining element in R
10k = p // k indexes the location in A to fill
11 // As long as each of the arrays L and R contains an unmerged element,
//  copy the smallest unmerged element back into A[p : r].
12 while i < nL and j < nR
13 if L[i] ≤ R[j]
14 A[k] = L[i]
15 i = i + 1
16 else A[k] = R[j]
17 j = j + 1
18 k = k + 1
19 // Having gone through one of L and R entirely, copy the
/ /remainder of the other to the end of A[p : r].
20while i < nL
21 A[k] = L[i]
22 i = i + 1
23 k = k + 1
24while j < nR
25 A[k] = R[j]
26 j = j + 1
```

	27 k = k + 1

PROGRAM:	mainProgram.cpp
	
	`#include <iostream>`
	`#include <fstream>`
	`#include <cstdlib>`
	`#include <ctime>`
	`#include <bits/stdc++.h>`
	
	`using namespace std;`
	
	`int listt[100000];`
	
	`void read()`
	`{`
	`  ifstream fin("values.txt", ios::binary);`
	`  for (long i = 0; i < 100000; i++)`
	`  {`
	`  fin.read((char *)&listt[i], sizeof(int));`
	`  }`
	`  fin.close();`
	`}`
	
	`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);
    }
}

long partition(long left, long right)
{
    int pivot_element = listt[left];
    int lb = left, ub = right;
    int temp;

    while (left < right)
    {
        while (listt[left] <= pivot_element)
            left++;
        while (listt[right] > pivot_element)
            right--;
        if (left < right)
        {
            temp = listt[left];
            listt[left] = listt[right];
            listt[right] = temp;
        }
    }
    listt[lb] = listt[right];
    listt[right] = pivot_element;
    return right;
}

void quickSort(long left, long right)
{
    if (left < right)
    {
        long pivot = partition(left, right);
        quickSort(left, pivot - 1);
        quickSort(pivot + 1, right);
    }
```

```
}

int main()
{
    clock_t t1, t2, t3, t4;
    read();
    int num = 100;
    ofstream output("./output2.csv");
    output << "block_size,insertion,selection\n";
    for (int i = 0; i < 1000; i++)
    {
        t1 = clock();
        mergeSort(listt, 0, num - 1);
        t2 = clock();
        t3 = clock();
        quickSort(0, num - 1);
        t4 = clock();
        double mergetime = double(t2 - t1) /
double(CLOCKS_PER_SEC);
        double quicktime = double(t4 - t3) /
double(CLOCKS_PER_SEC);
        cout << endl;
        output << i+1 <<","<<fixed << mergetime <<
setprecision(5) << '\t';
        output << fixed << ","  <<quicktime <<
setprecision(5) <<"\n";
        cout << i + 1 << " " << fixed << mergetime
<< "\t";
        cout << fixed << quicktime;
        num += 100;
    }

    return 0;
}
```

randomValues.cpp

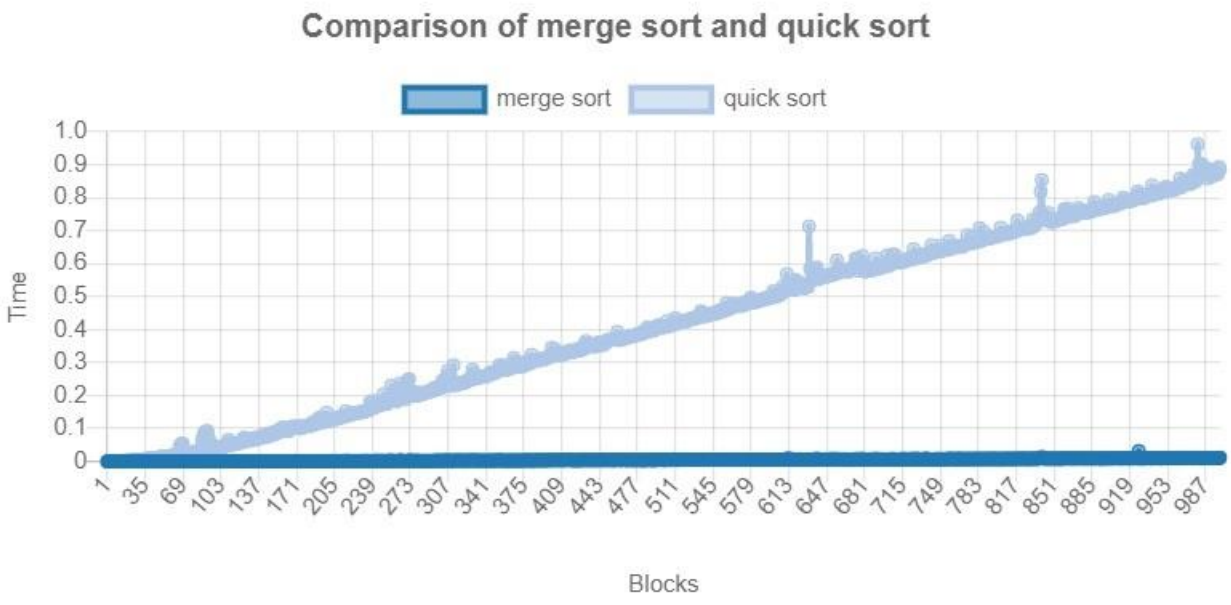
```cpp
#include <iostream>
#include <cstdlib>

using namespace std;

int main() {

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

Observation (SNAPSHOT)



Observation

For quick sort

the graph of the running time of quick sort reveals that as the size of the input data increases, the

running time also increases at a steady and consistent rate. The graph exhibits a characteristic upward trend, with the slope of the curve becoming steeper as the size of the input data increases.

For merge sort

the graph of the running time of merge sort shows that it also increases as the size of the input data increases, similar to insertion sort. However, the rate of increase is very very slow and very very minute increase in time takes place which is very slow compared to merge sort, which is evident from the less steep slope of the graph. This suggests that merge sort is more efficient.

Conclusion

The experiment on finding the running time of quick sort and merge sort shows that the running time of both algorithms is dependent on the size of the data. However we observed that the merge sort takes very very less time to sort the random numbers and thus it is proved to be more efficient.