DESIGN AND ANALYSIS OF ALGORITHMS

EXPERIMENT 2

ANUSHKA ACHARYA
CSE DS D1 BATCH
UID: 2021700001

AIM: EXPERIMENT BASED ON DIVIDE AND CONQUER APPROACH

THEORY:

Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.
In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.Merge sort is a popular choice for sorting large datasets because it is relatively efficient and easy to implement. It is often used in conjunction with other algorithms, such as quicksort, to improve the overall performance of a sorting routine.

https://www.geeksforgeeks.org/merge-sort/

QuickSort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- Pick median as the pivot.

The key process in quickSort is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.
https://www.geeksforgeeks.org/quick-sort/

CODE:
MERGESORT:

```
#include<stdlib.h>
#include<stdio.h>
#include <time.h>

void merge(int a[], int l, int mid, int r){
int n1 = mid-l+1;
int n2 = r-mid;
int i,j,k;
int L[n1], R[n2];
for (i = 0; i < n1; i++)
L[i] = a[l + i];
```

```
for (j = 0; j < n2; j++)
R[j] = a[mid+ 1+ j];
i = 0;
j = 0;
k = l;
while (i < n1 && j < n2){
if (L[i] <= R[j]){
a[k] = L[i];
i++;
}
else{
a[k] = R[j];
j++;
}
k++;
}
while (i < n1){
a[k] = L[i];
i++;
k++;
}
while (j < n2){
a[k] = R[j];
j++;
k++;
}
}
void mergeSort(int a[], int l, int r){
if (l < r){
int mid= l+(r-l)/2;
mergeSort(a, l, mid);
mergeSort(a, mid+1, r);
merge(a, l, mid, r);
}
}
int main(){
int size = 100;
FILE *fp;
fp = fopen("anu.txt","r");
while(size<=100000){
double time_taken=0.0;
clock_t begin=clock();
int a[size];
for(int j=0;j<size;j++){
fscanf(fp,"%d ",&a[j]);
}
mergeSort(a, 0, size - 1);
for(int test1=1;test1<size;test1++){
printf("%d\t",a[test1]);
}
printf("\n");
size=size+100;
fseek(fp,0,SEEK_SET);
clock_t end=clock();
time_taken = (double)(end-begin)/CLOCKS_PER_SEC;
```

```
printf("Runtime: %f\n",time_taken);
}
return 0;
}
```

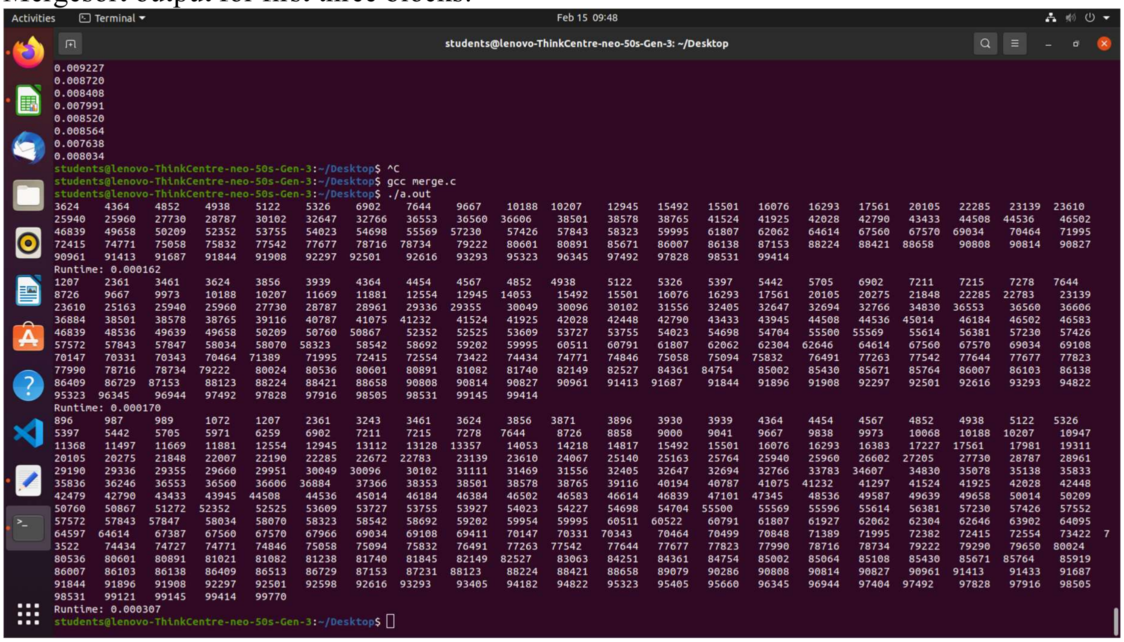
QUICKSORT:

```
#include<stdlib.h>
#include<stdio.h>
#include <time.h>

int partition (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);
}

void quickSort(int a[], int start, int end) {
    if (start < end)  {
        int p = partition(a, start, end);
        quickSort(a, start, p - 1);
        quickSort(a, p + 1, end);
    }
}
int main(){
int size = 100;
FILE *fp;
fp = fopen("anu.txt","r");
while(size<=100000){
double time_taken=0.0;
clock_t begin=clock();
int a[size];
for(int j=0;j<size;j++){
fscanf(fp,"%d ",&a[j]);
}
quickSort(a, 0, size - 1);
for(int test1=1;test1<size;test1++){
printf("%d\t",a[test1]);
}
printf("\n");
size=size+100;
```

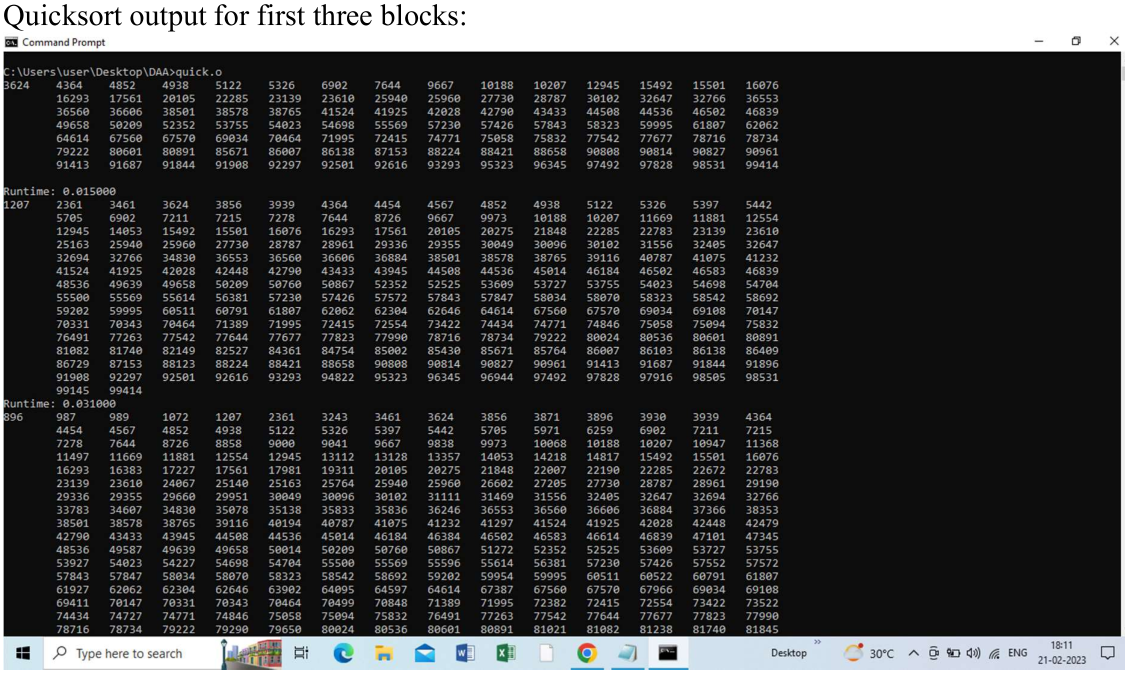
```
fseek(fp,0,SEEK_SET);
clock_t end=clock();
time_taken = (double)(end-begin)/CLOCKS_PER_SEC;
printf("Runtime: %f\n",time_taken);
}
return 0;
}
```

OUTPUT:

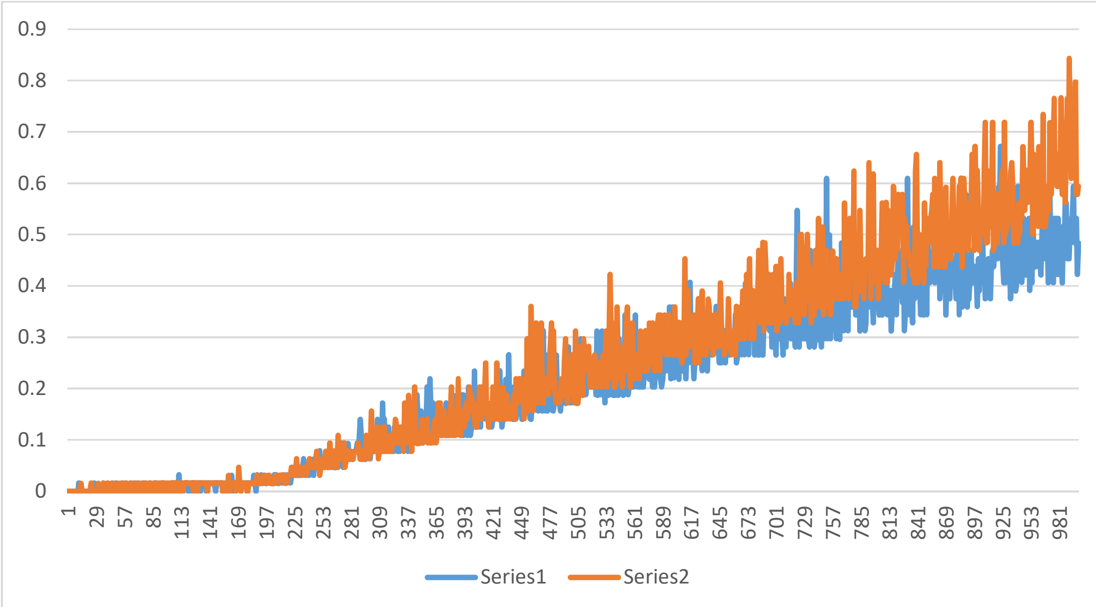
Mergesort output for first three blocks:



Quicksort output for first three blocks:



GRAPH:



OBSERVATION:
In this graph, the blue line represents merge sort whereas the orange line represents quicksort algorithm. It can be seen clearly then quicksort has a higher runtime than the merge sort algorithm. This is because in the merge sort, the array is parted into just 2 halves (i.e. n/2) where as In case of quick sort, the array is parted into any ratio. The worst case complexity of quick sort is O(n^2) as there is need of lot of comparisons in the worst condition. Where as In merge sort, worst case and average case has same complexities O(n log n). Thus, even though both algorithms work on divide and conquer strategy, merge sort is more efficient of the two when dealing with a huge set of data. Quicksort is usually preferred for small data sets.

CONCLUSION:
From this experiment I learnt how to perform merge sort and quicksort algorithms. I used the sorting techniques on 100000 random numbers and found out that runtime for merge sort is less than quicksort which helped me analyse that merge sort is the algorithm more efficient between the two, when working on a large set of data.