Experiment No.	2
Aim	To compare two sorting algorithms- Merge sort and Quick sort, based on their running time.
Name	Allen Andrew
UID No.	2021300006
Class & Division	SE COMPS A

Algorithms:

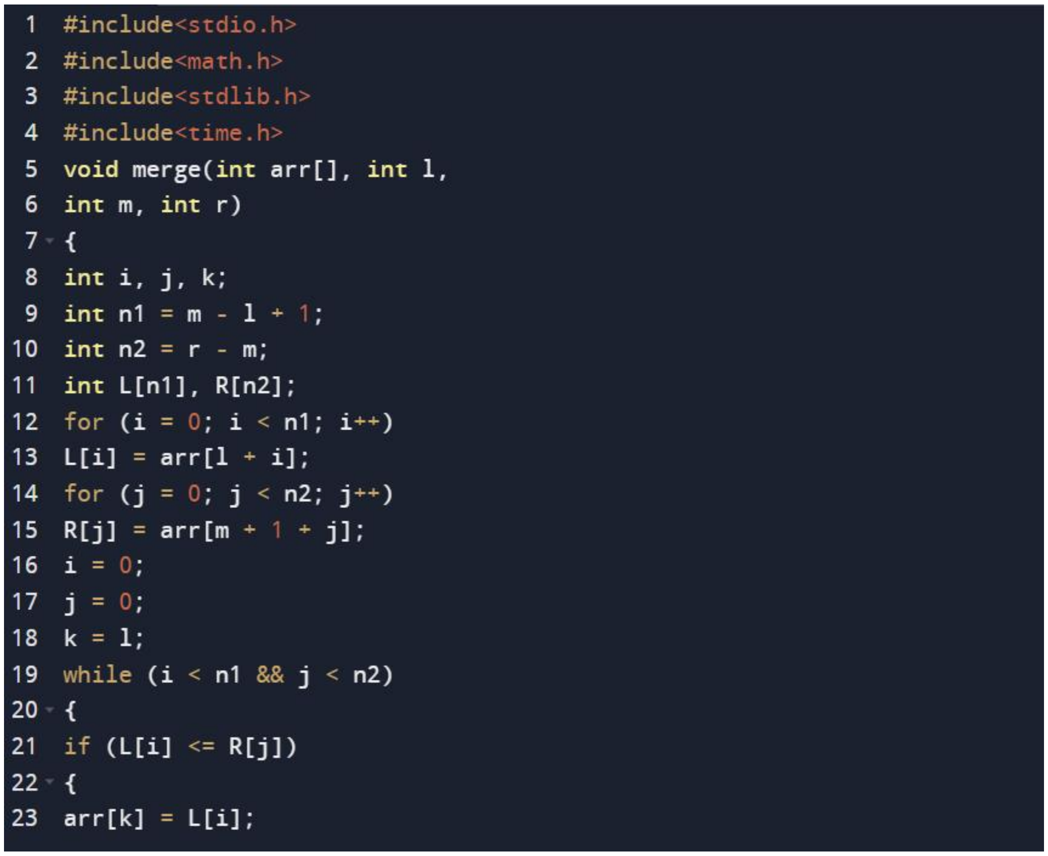
[A] For merge sort

I. Calculate the middle index of the array that is to be sorted.
II. Call the mergeSort function for the two groups of elements formed in the concerned array.
III. Create a temporary array to store the elements of the two groups formed in order to sort them.
IV. Using two pointers, one on the beginning of each group, start traversing the groups and comparing the elements pointed.
V. Pick the smallest of the two elements, move it to the main array, and increment the concerned pointer by 1.
VI. Repeat till both the groups in the temporary array are empty.

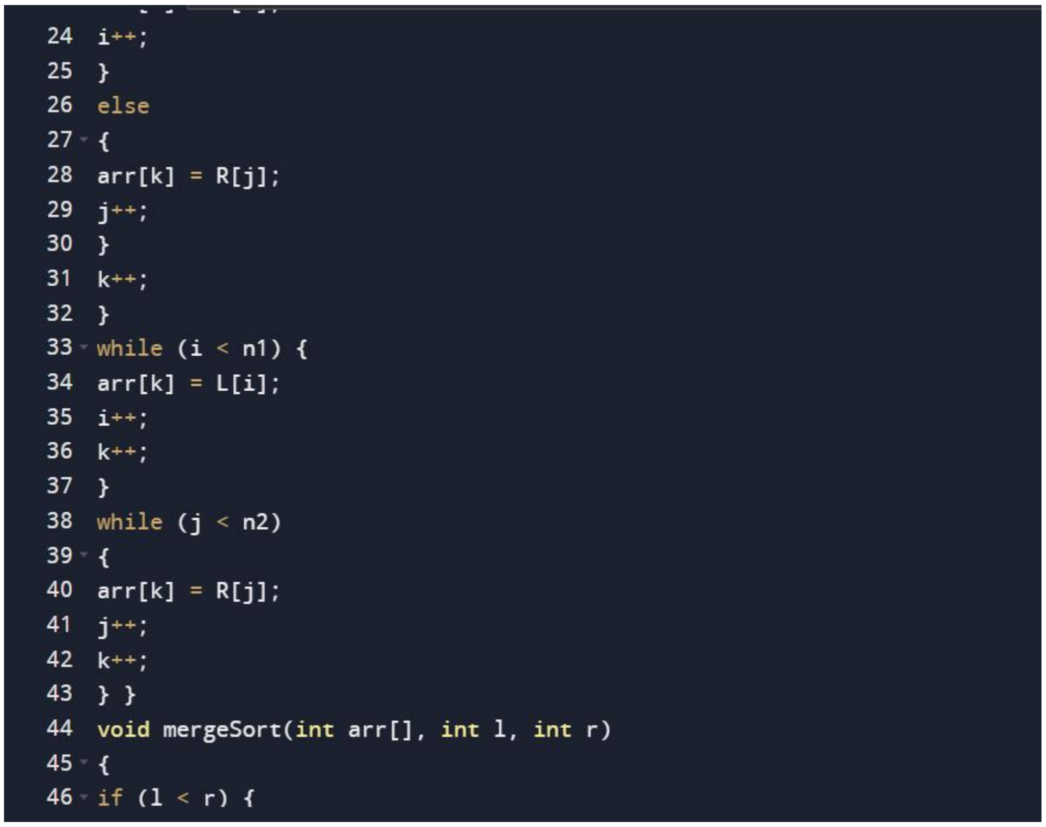
[B] For quick sort

I. Fix the pivot, the low and high pointers on the array.
II. Increment the low pointer till an element greater than the pivot is found, stop if the pointer reaches end of array.
III. Decrement the high pointer till an element lesser than the pivot is found, stop if the pointer reaches beginning of array.
IV. If high pointer is ahead of the low pointer, swap the elements pointed by them.
V. Repeat steps 2 to 4 till high is ahead of low pointer.
VI. Swap the pivot and the element pointed by high.
VII. Using the partitioning index represented by high pointer, call the quickSort function for the left and right partition

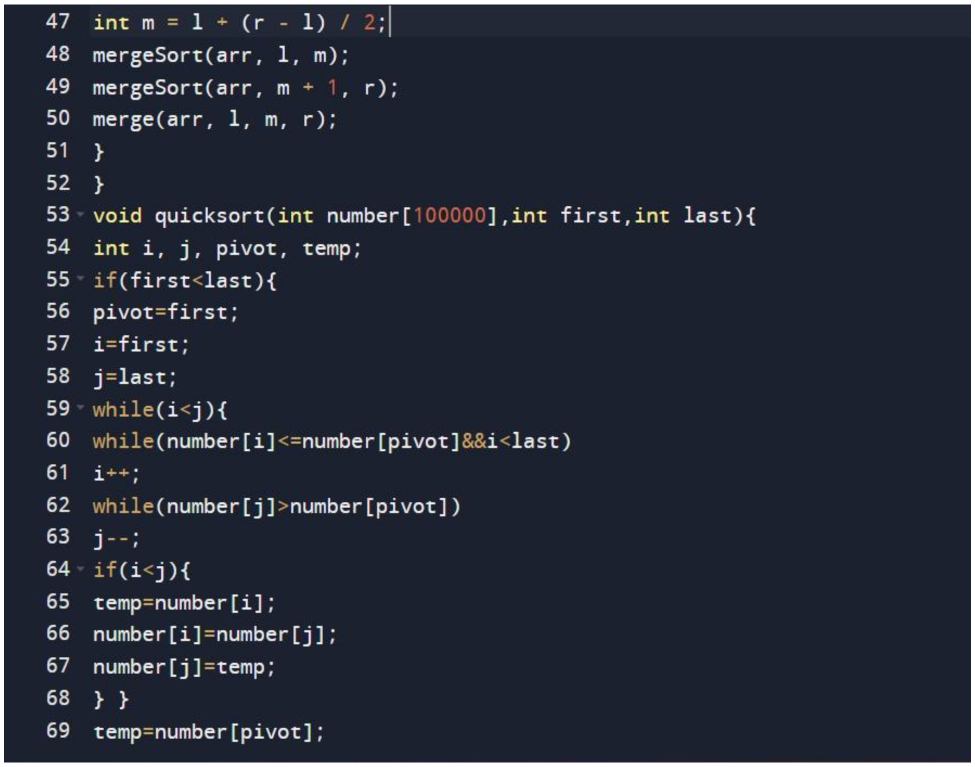
Program:



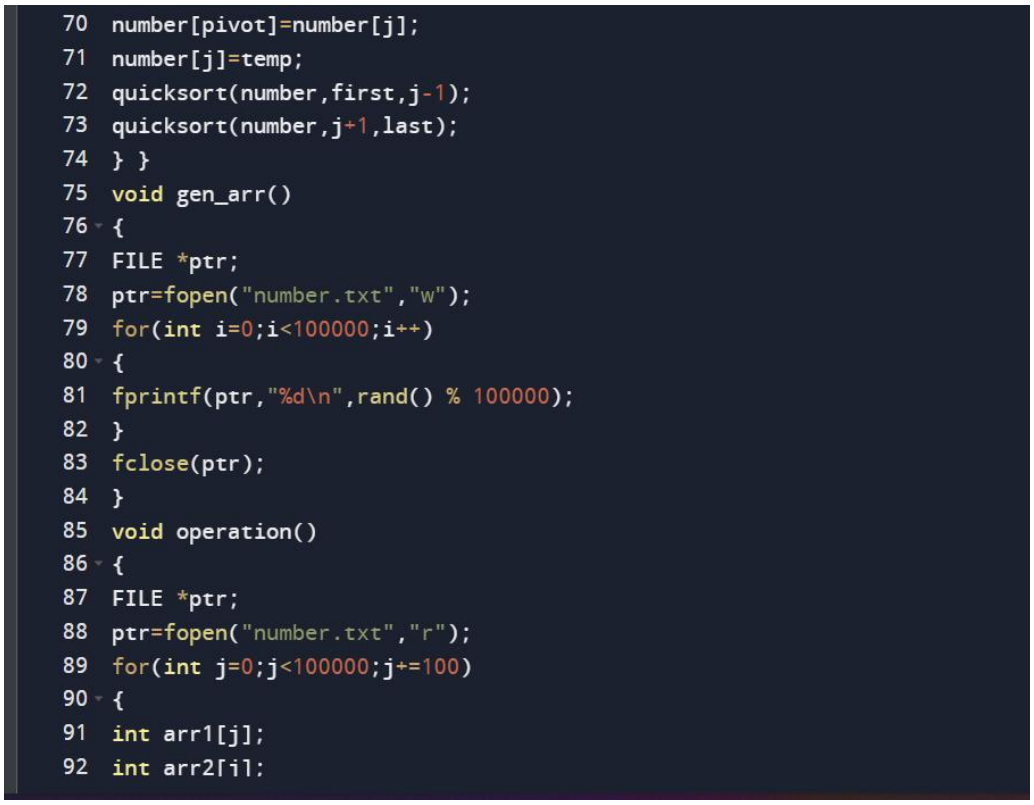
```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<time.h>
void merge(int arr[], int l,
int m, int r)
{
int i, j, k;
int n1 = m - l + 1;
int n2 = r - m;
int L[n1], R[n2];
for (i = 0; i < n1; i++)
L[i] = arr[l + i];
for (j = 0; j < n2; j++)
R[j] = arr[m + 1 + j];
i = 0;
j = 0;
k = l;
while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



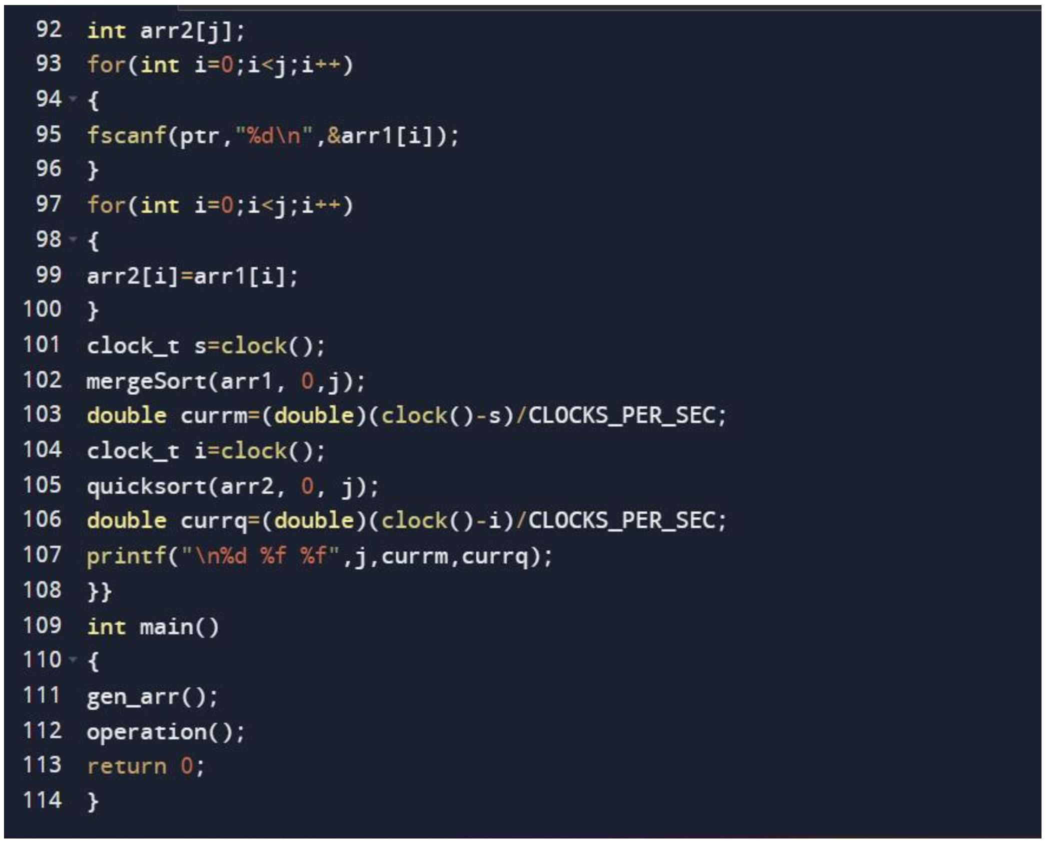
```
i++;
}
else
{
arr[k] = R[j];
j++;
}
k++;
}
while (i < n1) {
arr[k] = L[i];
i++;
k++;
}
while (j < n2)
{
arr[k] = R[j];
j++;
k++;
} }
void mergeSort(int arr[], int l, int r)
{
if (l < r) {
```



```
int m = l + (r - l) / 2;
mergeSort(arr, l, m);
mergeSort(arr, m + 1, r);
merge(arr, l, m, r);
}
}
void quicksort(int number[100000],int first,int last){
int i, j, pivot, temp;
if(first<last){
pivot=first;
i=first;
j=last;
while(i<j){
while(number[i]<=number[pivot]&&i<last)
i++;
while(number[j]>number[pivot])
j--;
if(i<j){
temp=number[i];
number[i]=number[j];
number[j]=temp;
} }
temp=number[pivot];
```

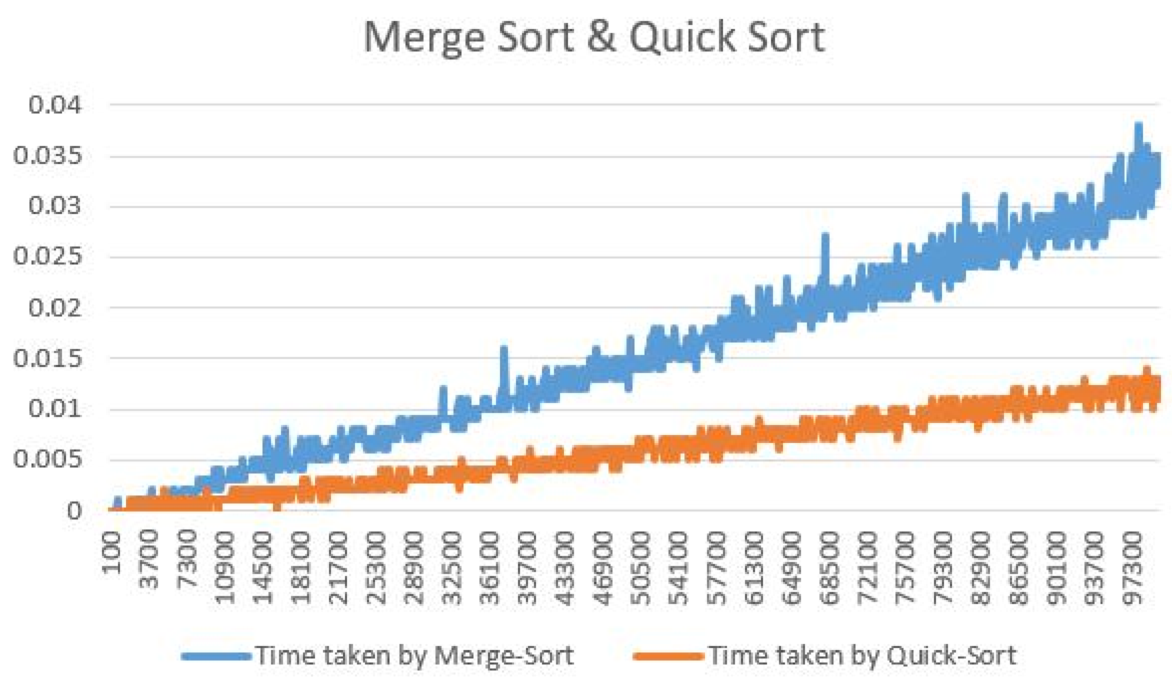


```
number[pivot]=number[j];
number[j]=temp;
quicksort(number,first,j-1);
quicksort(number,j+1,last);
} }
void gen_arr()
{
FILE *ptr;
ptr=fopen("number.txt","w");
for(int i=0;i<100000;i++)
{
fprintf(ptr,"%d\n",rand() % 100000);
}
fclose(ptr);
}
void operation()
{
FILE *ptr;
ptr=fopen("number.txt","r");
for(int j=0;j<100000;j+=100)
{
int arr1[j];
int arr2[i];
```



```c
int arr2[j];
for(int i=0;i<j;i++)
{
fscanf(ptr,"%d\n",&arr1[i]);
}
for(int i=0;i<j;i++)
{
arr2[i]=arr1[i];
}
clock_t s=clock();
mergeSort(arr1, 0,j);
double currm=(double)(clock()-s)/CLOCKS_PER_SEC;
clock_t i=clock();
quicksort(arr2, 0, j);
double currq=(double)(clock()-i)/CLOCKS_PER_SEC;
printf("\n%d %f %f",j,currm,currq);
}}
int main()
{
gen_arr();
operation();
return 0;
}
```

Observation:



Merge Sort-BLUE
Qucik Sort- RED

In conclusion, both quicksort and merge sort have their advantages and disadvantages, and their performance can vary depending on the input data. Quick sort is generally faster than merge sort, but merge sort is more stable and can handle larger arrays more efficiently. Ultimately, the choice between the two depends on the specific requirements of the sorting task and the characteristics of the data being sorted.
We notice that initially, when the elements are less in number, both the sorting algorithms work equally well; however, upon the increase of elements further, these two deviate in terms of time taken for the process. Nevertheless, both of these algorithms are very efficient and take much less time than insertion or selection sort algorithms.

Conclusion:

By performing this experiment, I was able to understand the process of determining the time an algorithm implementation takes to perform the required task. Using this, I was able to compare merge and quick sort algorithms which led me to the conclusion that both these algos are very efficient and take less time than insertion and selection sort.