NAME:	Shubham Vishwakarma
UID No.	2021700071
BRANCH:	S.Y CSE-DS
BATCH:	D
SUBJECT	Design and Analysis of Algorithms
EXPERIMENT No.	2
Date of Performance	13/02/2023
Date of Submission	18/02/2023

AIM:

Experiment based on divide and conquer approach.

Program 1

PROBLEM STATEMENT :

Each student have to generate random 100000 numbers using rand() function and use this input as 1000
blocks of 100,200,300,...,100000 integer numbers to Quicksort and Merge sorting algorithms.

ALGORITHM/ 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.

Algorithm:

```
step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.
    if left > right
        return
    mid= (left+right)/2
    mergesort(array, left, mid)
    mergesort(array, mid+1, right)
    merge(array, left, mid, right)

step 4: Stop
```

Worst Case Time Complexity [Big-O]: **O(n*log n)**

Best Case Time Complexity [Big-omega]: **O(n*log n)**

Average Time Complexity [Big-theta]: **O(n*log n)**

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(n^2), respectively.

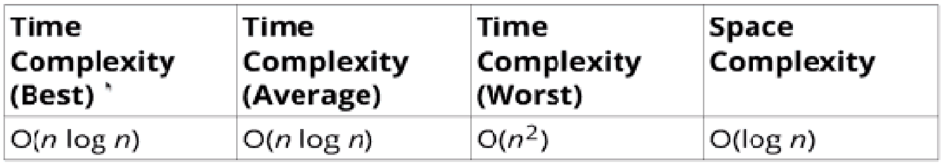
Algorithm:

```
QUICKSORT (array A, start, end)
{
 if (start < end)
 {
    p = partition(A, start, end)
    QUICKSORT (A, start, p - 1)
    QUICKSORT (A, p + 1, end)
 }
}
```

Partition Algorithm:

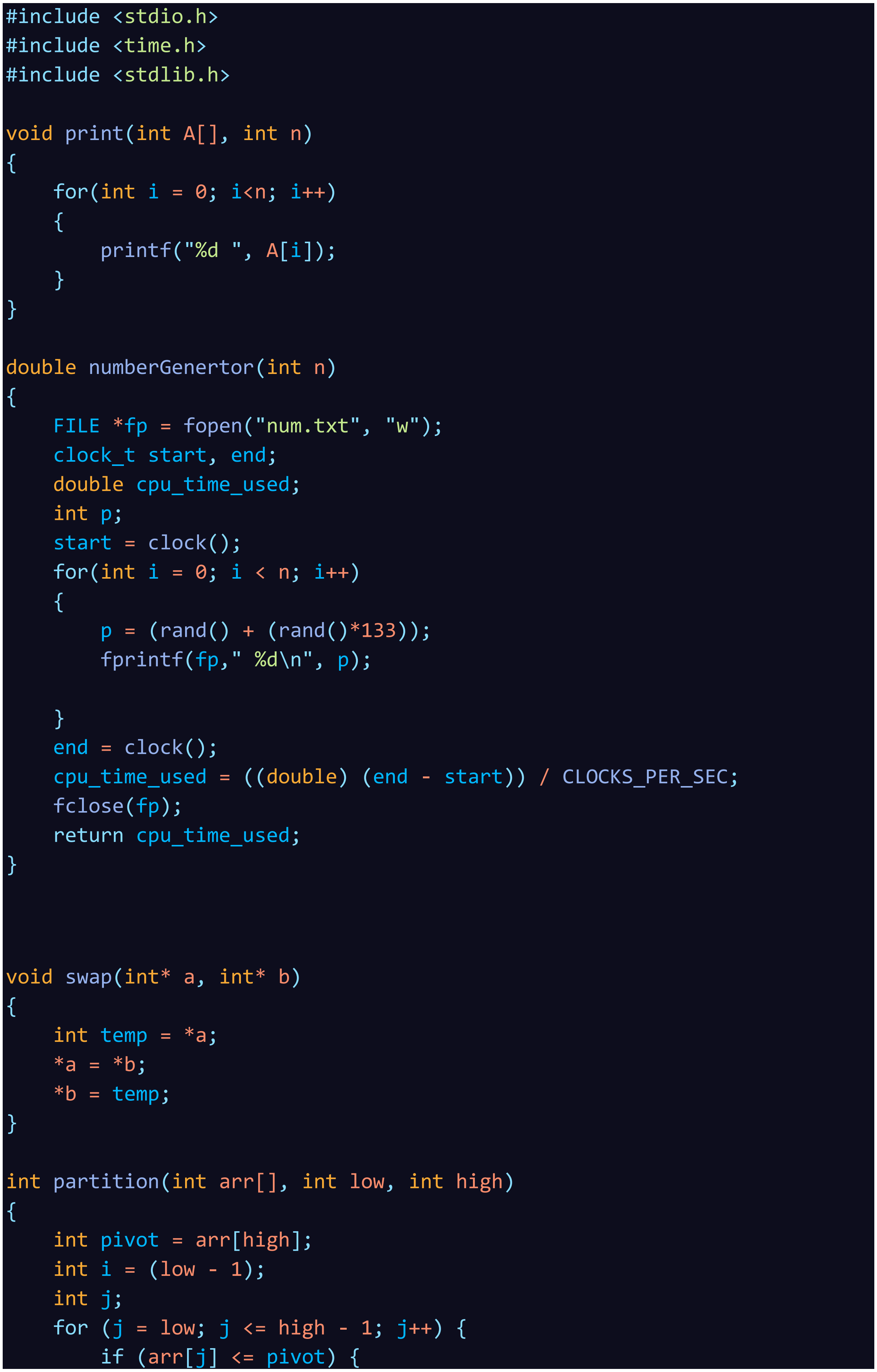
The partition algorithm rearranges the sub-arrays in a place.

```
PARTITION (array A, start, end)
{
 pivot ? A[end]
 i ? start-1
 for j ? start to end -1 {
    do if (A[j] < pivot) {
       then i ? i + 1
       swap A[i] with A[j]
 }}
 swap A[i+1] with A[end]
 return i+1
}
```

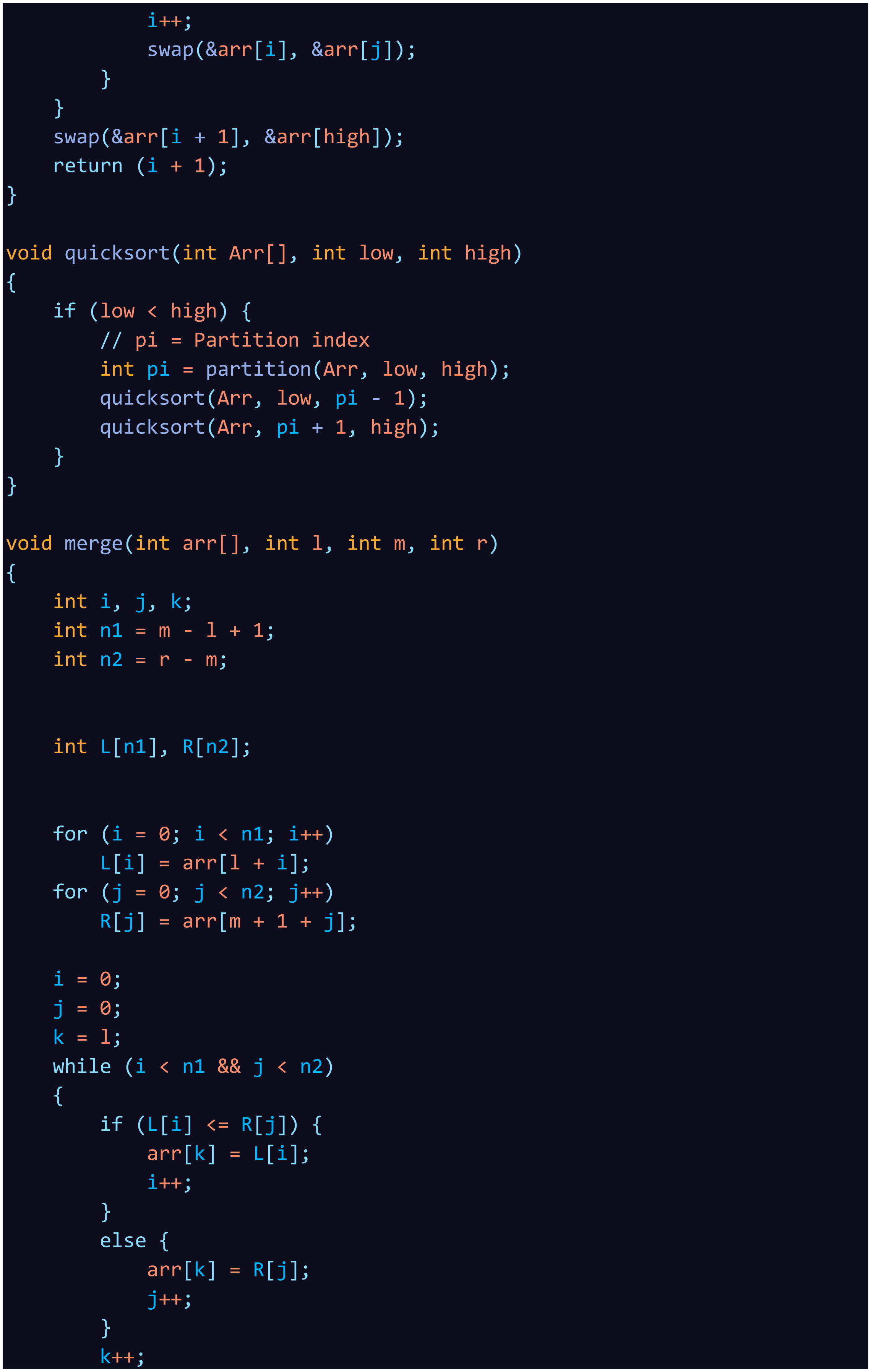


Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
O(*n* log *n*)	O(*n* log *n*)	O(n^2)	O(log *n*)

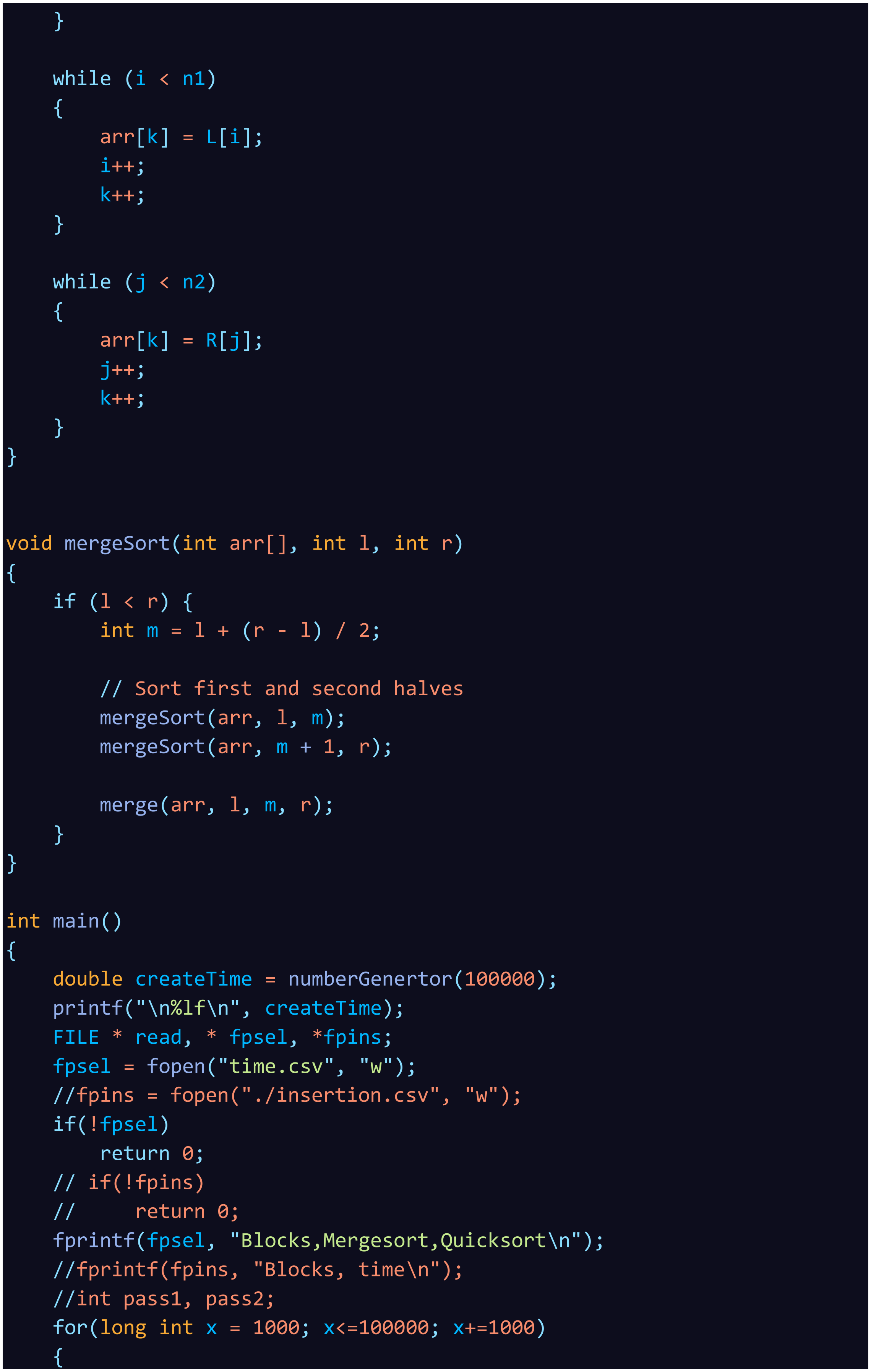
PROGRAM:



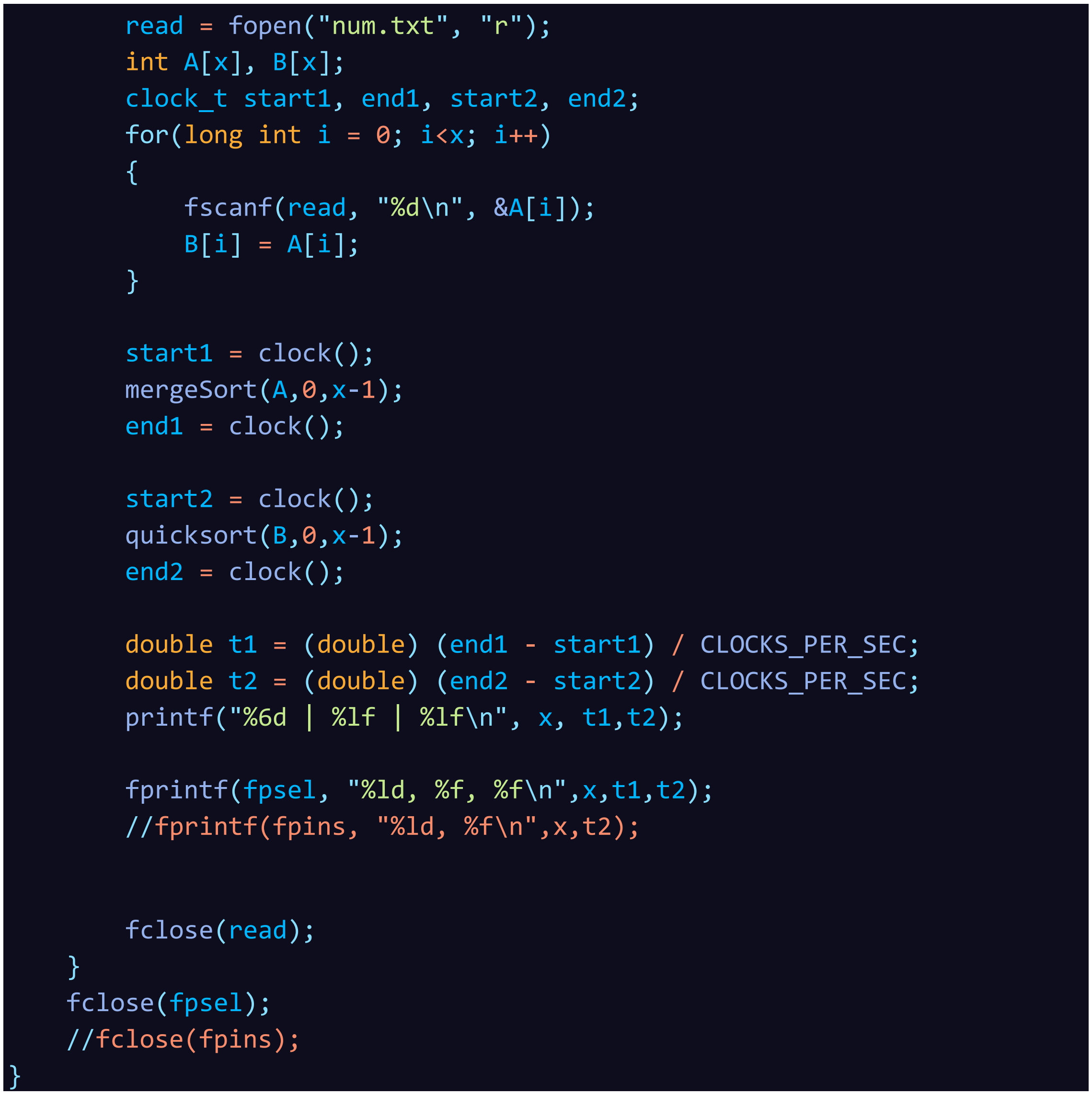
```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

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

double numberGenertor(int n)
{
    FILE *fp = fopen("num.txt", "w");
    clock_t start, end;
    double cpu_time_used;
    int p;
    start = clock();
    for(int i = 0; i < n; i++)
    {
        p = (rand() + (rand()*133));
        fprintf(fp," %d\n", p);

    }
    end = clock();
    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    fclose(fp);
    return cpu_time_used;
}



void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

int partition(int arr[], int low, int high)
{
    int pivot = arr[high];
    int i = (low - 1);
    int j;
    for (j = low; j <= high - 1; j++) {
        if (arr[j] <= pivot) {
```



```
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void quicksort(int Arr[], int low, int high)
{
    if (low < high) {
        // pi = Partition index
        int pi = partition(Arr, low, high);
        quicksort(Arr, low, pi - 1);
        quicksort(Arr, pi + 1, high);
    }
}

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++;
```

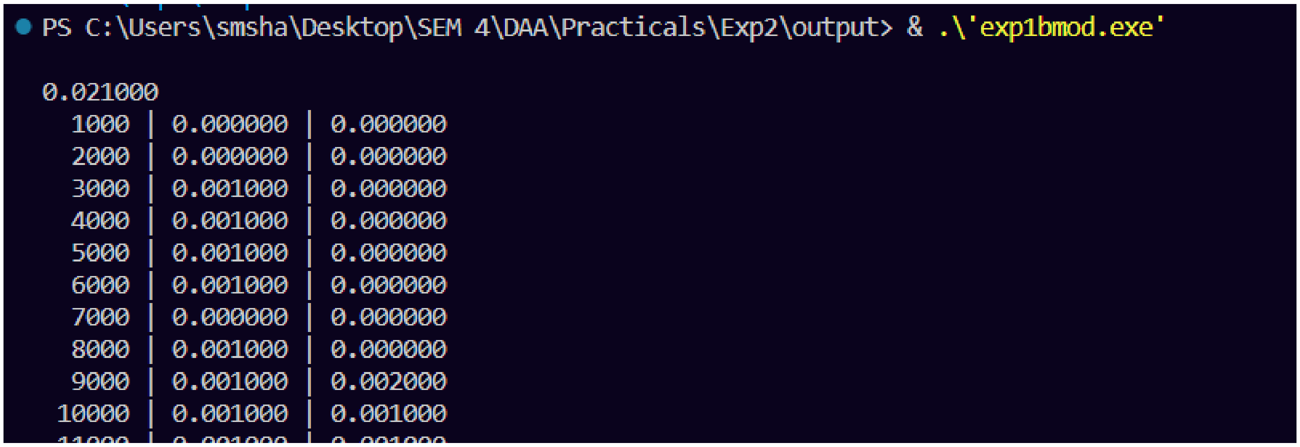


```
    }

    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;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}

int main()
{
    double createTime = numberGenertor(100000);
    printf("\n%lf\n", createTime);
    FILE * read, * fpsel, *fpins;
    fpsel = fopen("time.csv", "w");
    //fpins = fopen("./insertion.csv", "w");
    if(!fpsel)
        return 0;
    // if(!fpins)
    //     return 0;
    fprintf(fpsel, "Blocks,Mergesort,Quicksort\n");
    //fprintf(fpins, "Blocks, time\n");
    //int pass1, pass2;
    for(long int x = 1000; x<=100000; x+=1000)
    {
```

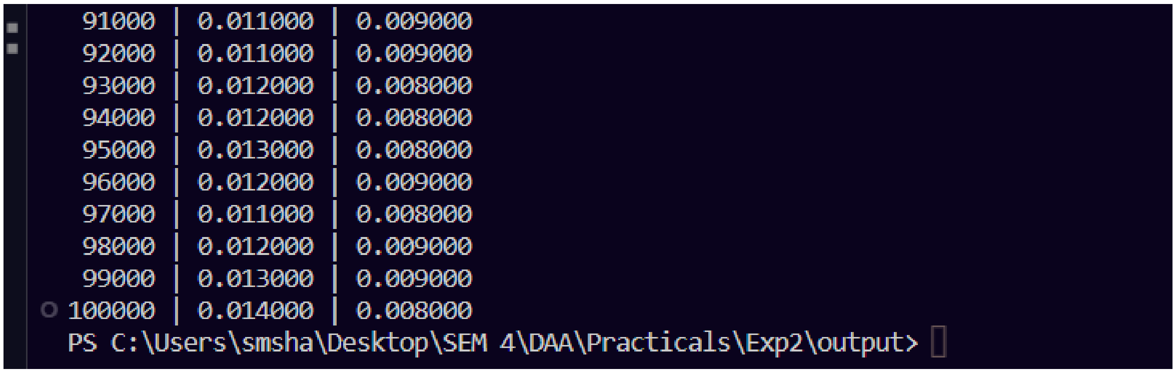


```c
        read = fopen("num.txt", "r");
        int A[x], B[x];
        clock_t start1, end1, start2, end2;
        for(long int i = 0; i<x; i++)
        {
            fscanf(read, "%d\n", &A[i]);
            B[i] = A[i];
        }

        start1 = clock();
        mergeSort(A,0,x-1);
        end1 = clock();

        start2 = clock();
        quicksort(B,0,x-1);
        end2 = clock();

        double t1 = (double) (end1 - start1) / CLOCKS_PER_SEC;
        double t2 = (double) (end2 - start2) / CLOCKS_PER_SEC;
        printf("%6d | %lf | %lf\n", x, t1,t2);

        fprintf(fpsel, "%ld, %f, %f\n",x,t1,t2);
        //fprintf(fpins, "%ld, %f\n",x,t2);


        fclose(read);
    }
    fclose(fpsel);
    //fclose(fpins);
}
```

RESULT:

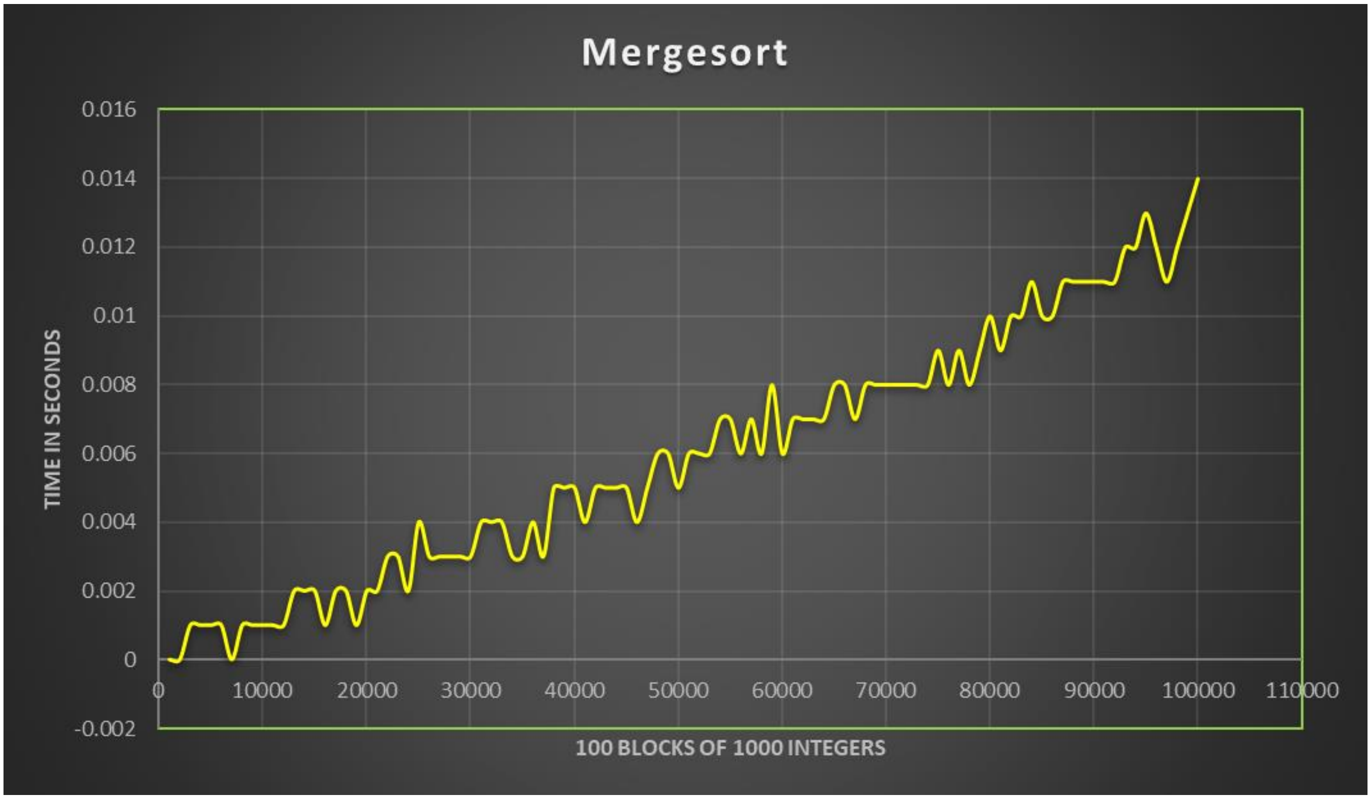


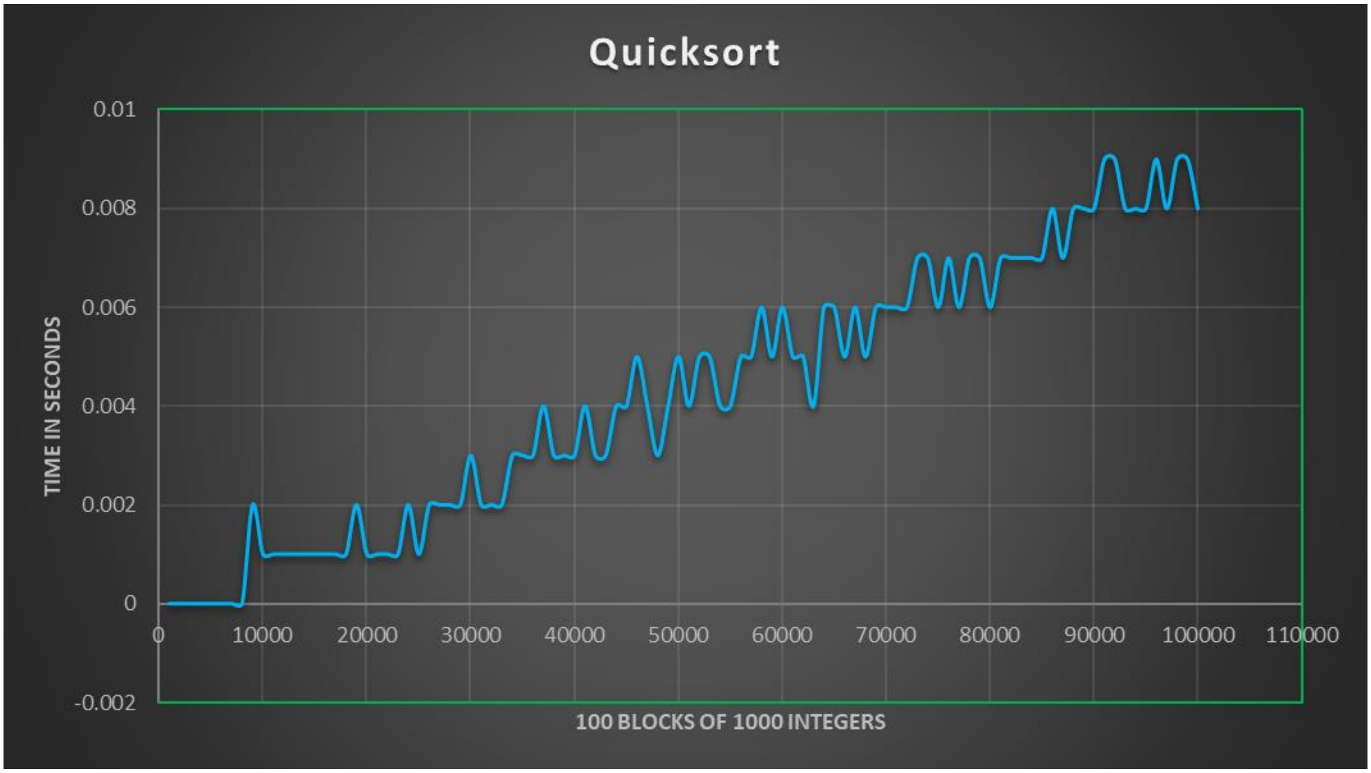
```
● PS C:\Users\smsha\Desktop\SEM 4\DAA\Practicals\Exp2\output> & .\'exp1bmod.exe'

0.021000
  1000 | 0.000000 | 0.000000
  2000 | 0.000000 | 0.000000
  3000 | 0.001000 | 0.000000
  4000 | 0.001000 | 0.000000
  5000 | 0.001000 | 0.000000
  6000 | 0.001000 | 0.000000
  7000 | 0.000000 | 0.000000
  8000 | 0.001000 | 0.000000
  9000 | 0.001000 | 0.002000
 10000 | 0.001000 | 0.001000
 11000 | 0.001000 | 0.001000
```

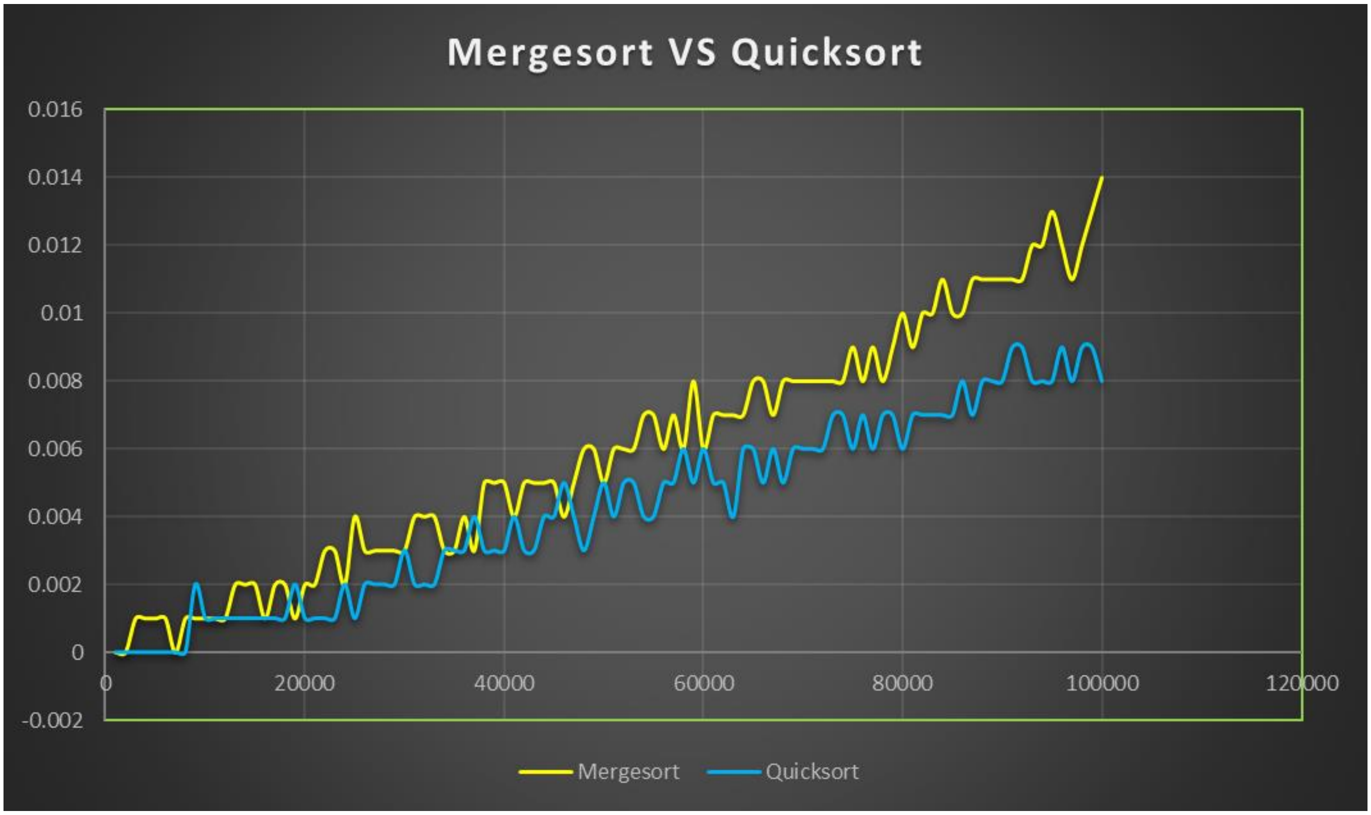


```
  91000 | 0.011000 | 0.009000
  92000 | 0.011000 | 0.009000
  93000 | 0.012000 | 0.008000
  94000 | 0.012000 | 0.008000
  95000 | 0.013000 | 0.008000
  96000 | 0.012000 | 0.009000
  97000 | 0.011000 | 0.008000
  98000 | 0.012000 | 0.009000
  99000 | 0.013000 | 0.009000
 100000 | 0.014000 | 0.008000
PS C:\Users\smsha\Desktop\SEM 4\DAA\Practicals\Exp2\output>
```

GRAPH:




Quicksort
TIME IN SECONDS
0.01
0.008
0.006
0.004
0.002
0
-0.002
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
110000
100 BLOCKS OF 1000 INTEGERS


Mergesort VS Quicksort
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
-0.002
0
20000
40000
60000
80000
100000
120000
Mergesort
Quicksort

From the graph we can see that Quick sort have better running time than Merge sort
The spikes in the graph due to following reasons

1. Due to a greater number of internal swaps at that given range
2. Due to CPU over heating (Machine Dependent)
3. Due to nature of random number (large difference between consecutive numbers)
4. Due to other process running in the background during the execution of program
5. Merge sort uses external sorting while Quick sort uses internal sorting

CONCLUSION :

In this experiment, we wrote a program which given 100000 integers in 100 blocks of 1000 numbers, uses the selection andinsertion sort algorithms to sort them in ascending order.

In merge sort, during every recursive step, we divide the problem into 2 subproblems and so on. We then sort those subproblems and merge the 2 of them to form the result of the current step and so on.

We analyzed the algorithm in which we found that the time complexity for all 3 cases of merge sort is O (n log(n)) which is very efficient.

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n^2)

Thus, we successfully accomplished the aim of this experiment.