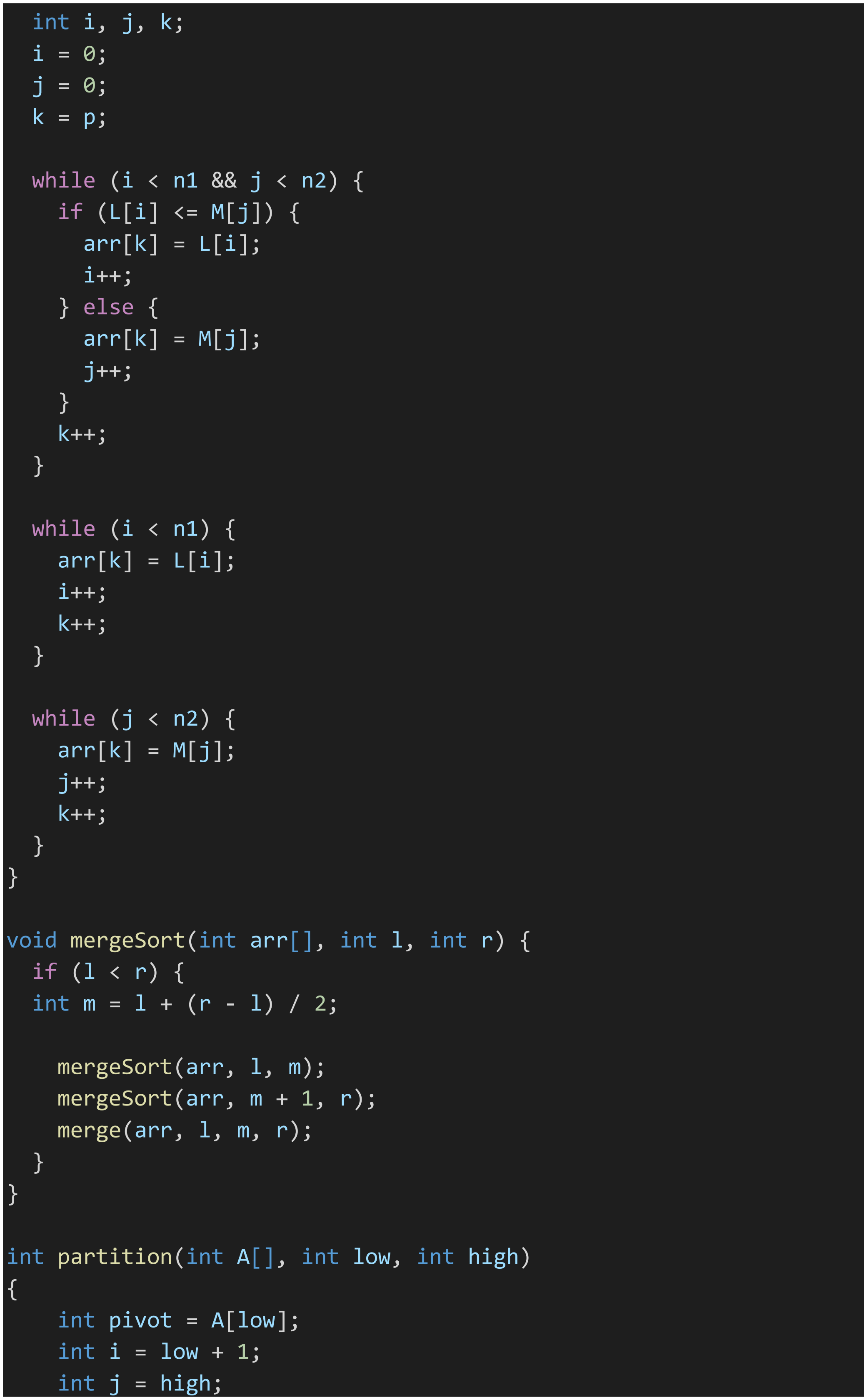
Name	Siddiqui Hawaiza Sabeel
UID no.	2022701010
Experiment No.	2
SUBJECT	DAA

AIM:	Experiment based on divide and conquer approach.
Program 1	
PROBLEM STATEMENT :	For this experiment, you need to implement two sorting algorithms namely Quicksort and Merge sort methods. Compare these algorithms based on time and space complexity. Time required for sorting algorithms can be performed using high_resolution_clock::now() under namespace std::chrono. You have to generate 1,00,000 integer numbers using C/C++ Rand function and save them in a text file. Both the sorting algorithms uses these 1,00,000 integer numbers as input as follows. Each sorting algorithm sorts a block of 100,200,300,...,100000 integer numbers with array indexes numbers A[0..99], A[100..199], A[200..299],…, A[99900..99999]. You need to use high_resolution_clock::now() function to find the time required for 100, 200, 300…. 100000 integer numbers. Finally, compare two algorithms namely Quicksort and Merge sort by plotting the time required to sort integers using LibreOffice Calc/MS Excel. The x-axis of 2-D plot represents the block no. of 1000 blocks. The y-axis of 2-D plot represents the tunning time to sort 1000 blocks of 100,200,300,...,100000 integer numbers. Note – You have to use C/C++ file processing functions for reading and writing randomly generated 100000 integer numbers
ALGORITHM/ THEORY:	1. **Merge Sort:** **Algorithm:** a. Start b. If the input array has length less than or equal to 1, return the array as it is already sorted. c. Divide the input array into two halves. d. Recursively sort the left half by calling merge sort on it. e. Recursively sort the right half by calling merge sort on it. f. Merge the two sorted halves into a single sorted array. g. Return the final sorted array. h. End

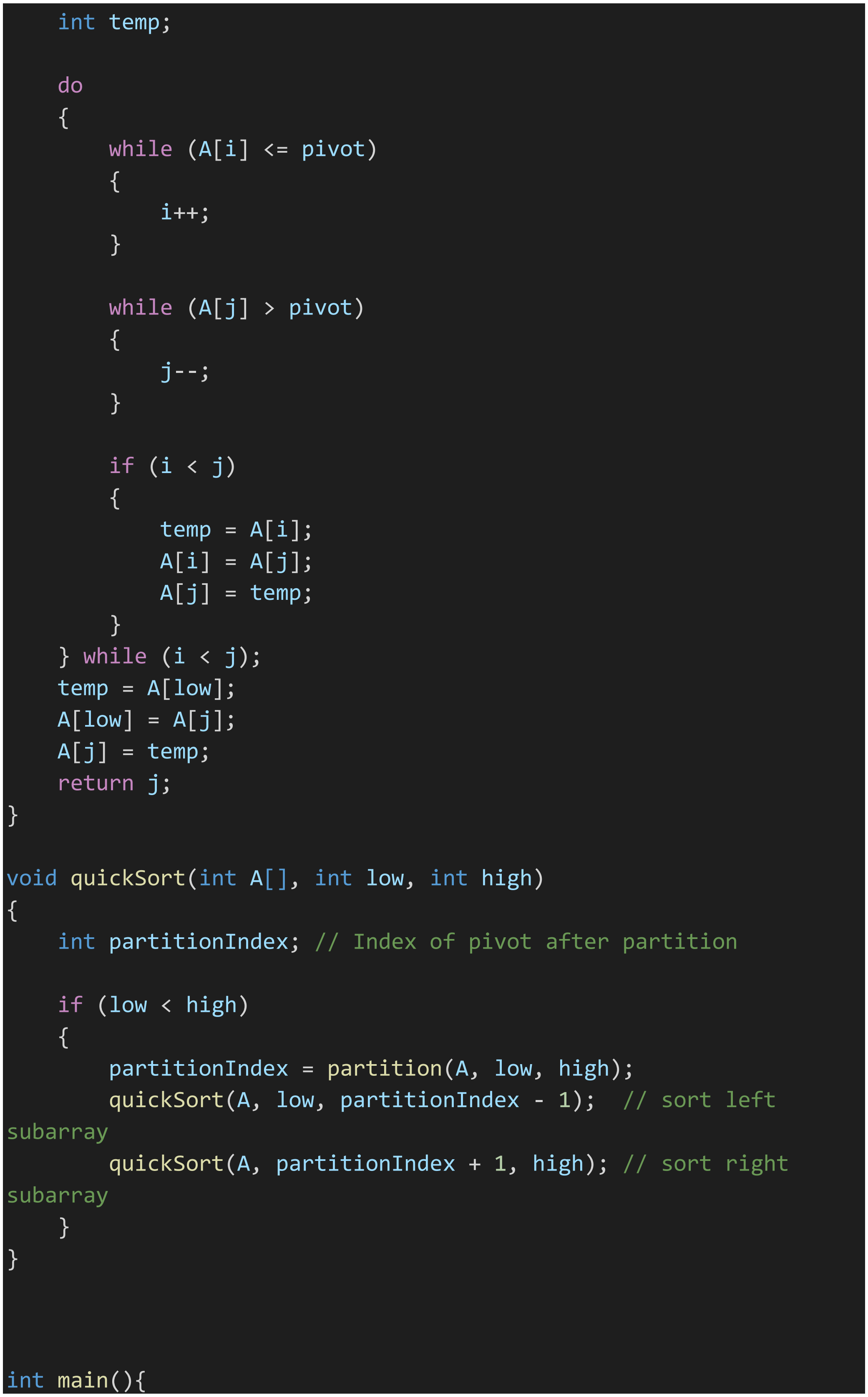
2. **Quick Sort:**
   Algorithm:
   a. Start
   b. If the input array has length less than or equal to 1, return the array as it is already sorted.
   c. Select a pivot element from the array.
   d. Partition the other elements into two sub-arrays, based on whether they are less than or greater than the pivot.
   e. Recursively sort the left sub-array by calling QuickSort on it.
   f. Recursively sort the right sub-array by calling QuickSort on it.
   g. Concatenate the left sub-array, the pivot, and the right sub-array to get the final sorted array.
   h. Return the final sorted array..
   i. End

PROGRAM:

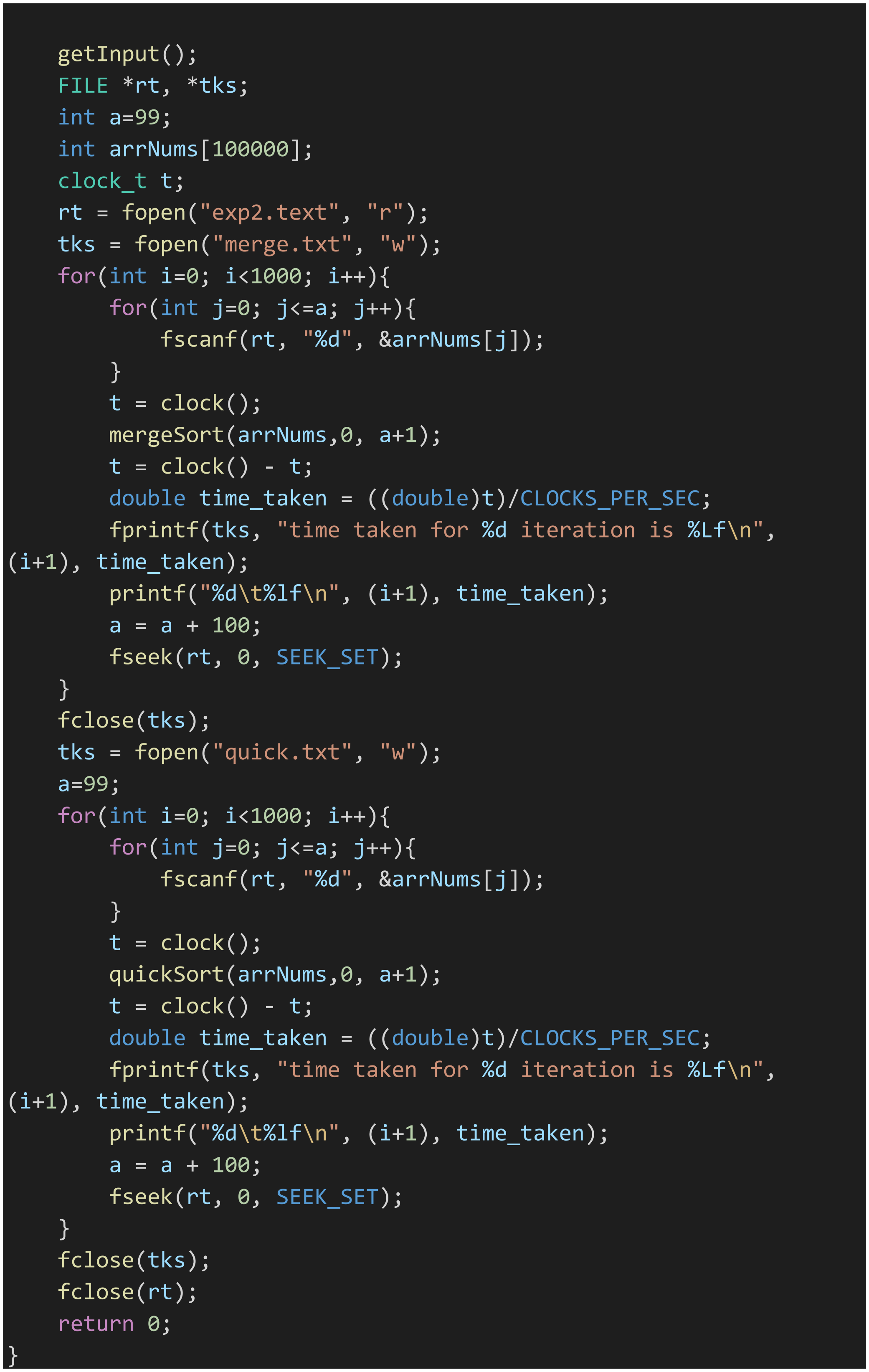
```
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>

void getInput()
{
  FILE *fp;
  fp = fopen("inputexp2.text","w");
  for(int i=0;i<100000;i++)
  fprintf(fp,"%d ",rand()%100000);
  fclose(fp);
}
void merge(int arr[], int p, int q, int r) {

  // Create L ← A[p..q] and M ← A[q+1..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];

  // Maintain current index of sub-arrays and main array
```



```
  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);
  }
}

int partition(int A[], int low, int high)
{
    int pivot = A[low];
    int i = low + 1;
    int j = high;
```

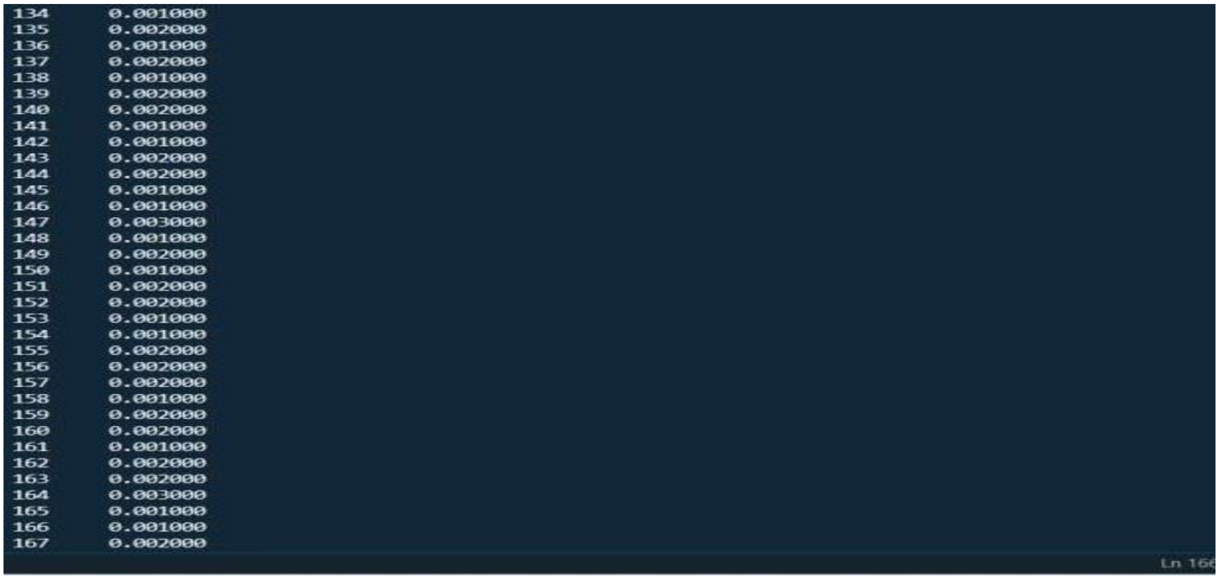


```
    int temp;

    do
    {
        while (A[i] <= pivot)
        {
            i++;
        }

        while (A[j] > pivot)
        {
            j--;
        }

        if (i < j)
        {
            temp = A[i];
            A[i] = A[j];
            A[j] = temp;
        }
    } while (i < j);
    temp = A[low];
    A[low] = A[j];
    A[j] = temp;
    return j;
}

void quickSort(int A[], int low, int high)
{
    int partitionIndex; // Index of pivot after partition

    if (low < high)
    {
        partitionIndex = partition(A, low, high);
        quickSort(A, low, partitionIndex - 1);  // sort left
subarray
        quickSort(A, partitionIndex + 1, high); // sort right
subarray
    }
}



int main(){
```

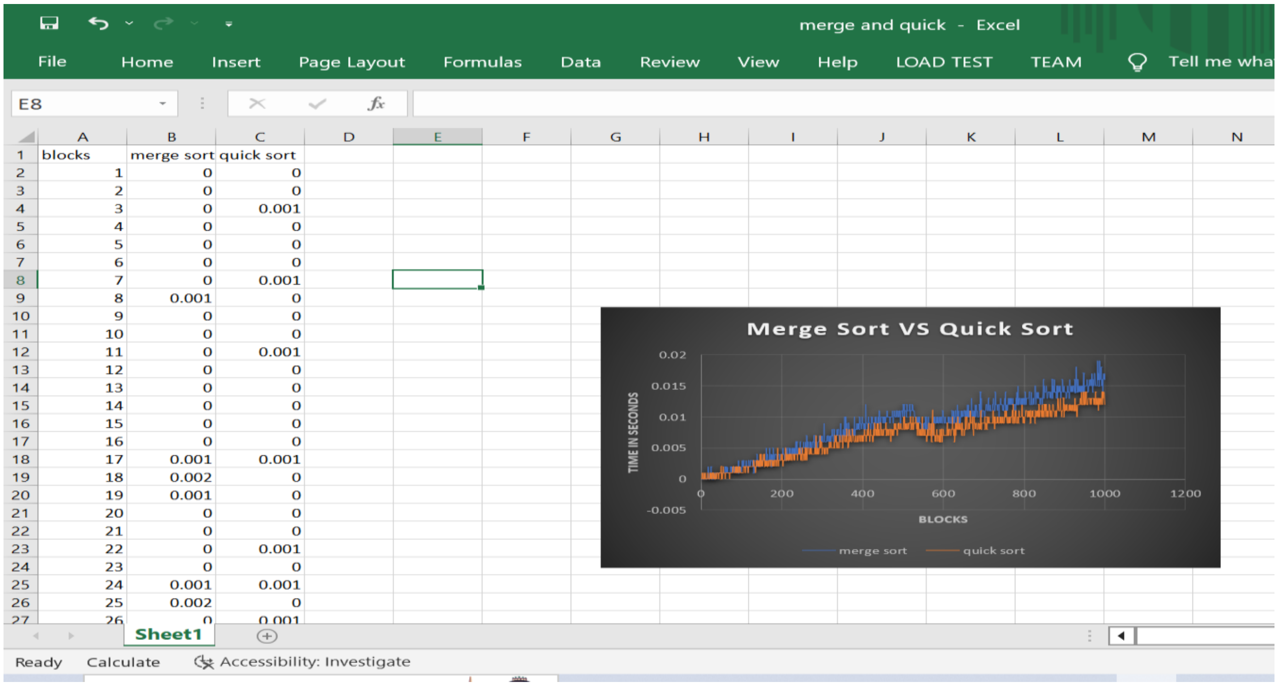


```c
    getInput();
    FILE *rt, *tks;
    int a=99;
    int arrNums[100000];
    clock_t t;
    rt = fopen("exp2.text", "r");
    tks = fopen("merge.txt", "w");
    for(int i=0; i<1000; i++){
        for(int j=0; j<=a; j++){
            fscanf(rt, "%d", &arrNums[j]);
        }
        t = clock();
        mergeSort(arrNums,0, a+1);
        t = clock() - t;
        double time_taken = ((double)t)/CLOCKS_PER_SEC;
        fprintf(tks, "time taken for %d iteration is %Lf\n",
(i+1), time_taken);
        printf("%d\t%lf\n", (i+1), time_taken);
        a = a + 100;
        fseek(rt, 0, SEEK_SET);
    }
    fclose(tks);
    tks = fopen("quick.txt", "w");
    a=99;
    for(int i=0; i<1000; i++){
        for(int j=0; j<=a; j++){
            fscanf(rt, "%d", &arrNums[j]);
        }
        t = clock();
        quickSort(arrNums,0, a+1);
        t = clock() - t;
        double time_taken = ((double)t)/CLOCKS_PER_SEC;
        fprintf(tks, "time taken for %d iteration is %Lf\n",
(i+1), time_taken);
        printf("%d\t%lf\n", (i+1), time_taken);
        a = a + 100;
        fseek(rt, 0, SEEK_SET);
    }
    fclose(tks);
    fclose(rt);
    return 0;
}
```

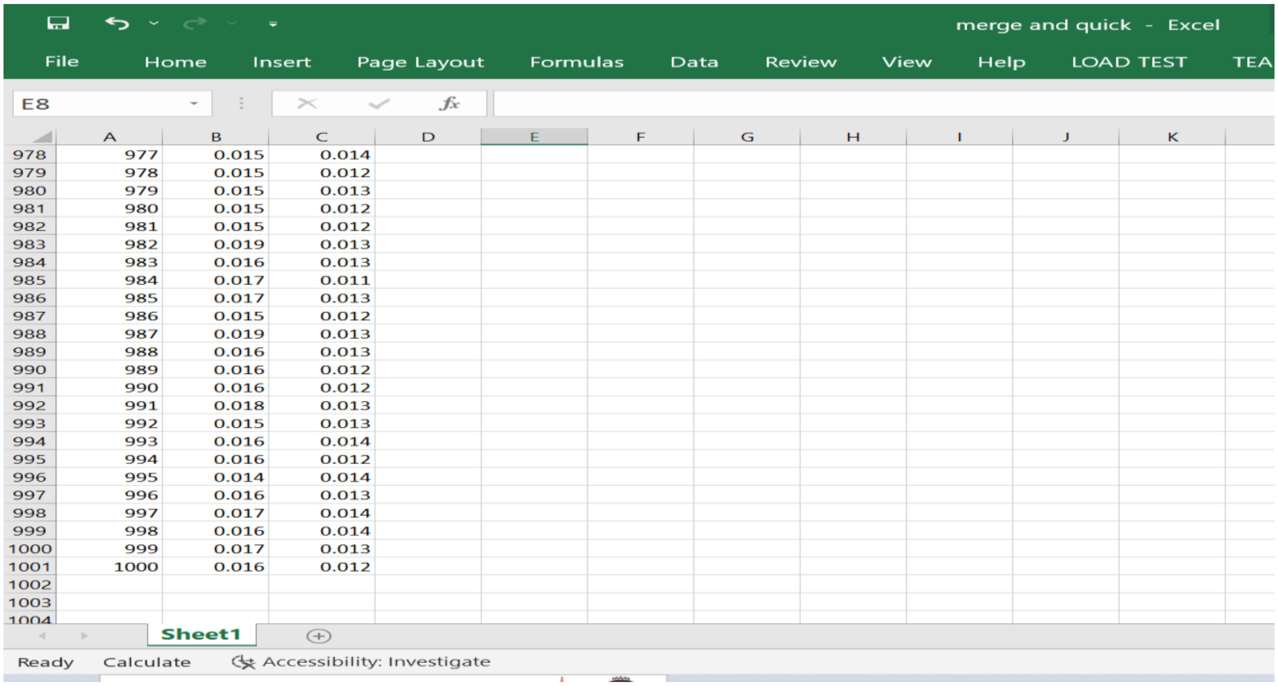
RESULT:



```
134      0.001000
135      0.002000
136      0.001000
137      0.002000
138      0.001000
139      0.002000
140      0.002000
141      0.001000
142      0.001000
143      0.002000
144      0.002000
145      0.001000
146      0.001000
147      0.003000
148      0.001000
149      0.002000
150      0.001000
151      0.002000
152      0.002000
153      0.001000
154      0.001000
155      0.002000
156      0.002000
157      0.002000
158      0.001000
159      0.002000
160      0.002000
161      0.001000
162      0.002000
163      0.002000
164      0.003000
165      0.001000
166      0.001000
167      0.002000
```

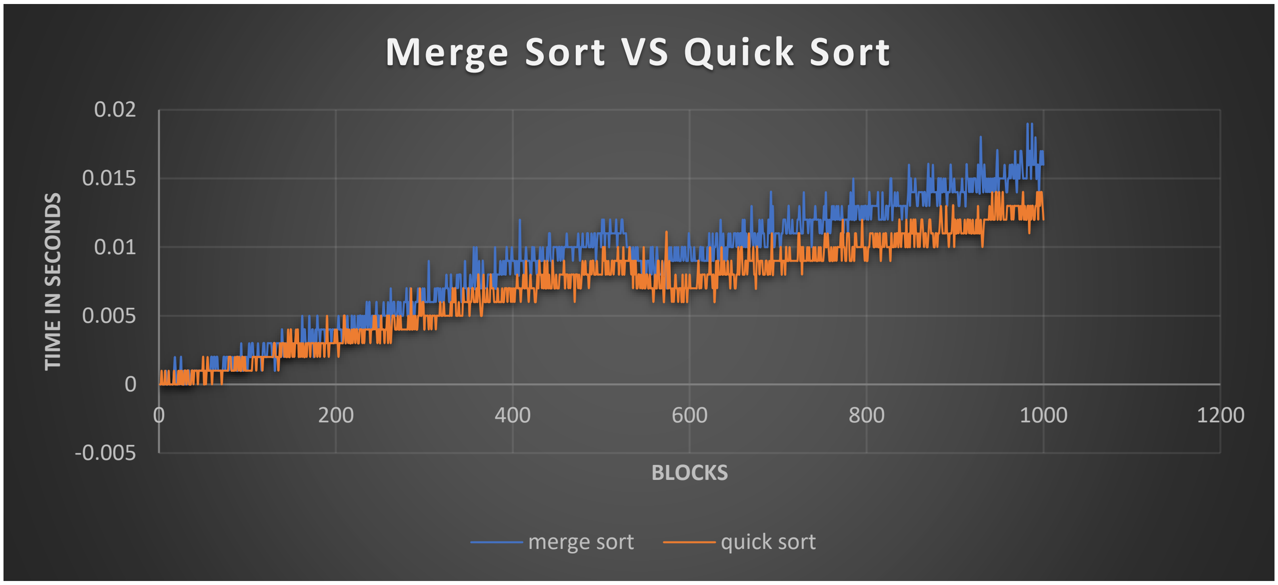


blocks	merge sort	quick sort
1	0	0
2	0	0
3	0	0.001
4	0	0
5	0	0
6	0	0
7	0	0.001
8	0.001	0
9	0	0
10	0	0
11	0	0.001
12	0	0
13	0	0
14	0	0
15	0	0
16	0	0
17	0.001	0.001
18	0.002	0
19	0.001	0
20	0	0
21	0	0
22	0	0.001
23	0	0
24	0.001	0.001
25	0.002	0
26	0	0.001



	A	B	C
978	977	0.015	0.014
979	978	0.015	0.012
980	979	0.015	0.013
981	980	0.015	0.012
982	981	0.015	0.012
983	982	0.019	0.013
984	983	0.016	0.013
985	984	0.017	0.011
986	985	0.017	0.013
987	986	0.015	0.012
988	987	0.019	0.013
989	988	0.016	0.013
990	989	0.016	0.012
991	990	0.016	0.012
992	991	0.018	0.013
993	992	0.015	0.013
994	993	0.016	0.014
995	994	0.016	0.012
996	995	0.014	0.014
997	996	0.016	0.013
998	997	0.017	0.014
999	998	0.016	0.014
1000	999	0.017	0.013
1001	1000	0.016	0.012
1002			
1003			
1004			

GRAPH:



CONCLUSION:	we have used two algorithm techniques i.e mergesort and quicksort to sort the random no.s . both the algorithms have less time complexity. i have seen behaviour of the algorithms with time using of graph . it is seen that quick has better time complexity than merge sort.