Name	Aditi Nilesh Bhutada
UID no.	2021700009
Experiment No.	2

AIM:	Experiment based on divide and conquer approach. Finding the running time of an algorithm. The understanding of running time of algorithms is explored by implementing two basic sorting algorithms namely merge sort and quick sort.
PROGRAM	
THEORY:	**Quicksort:** It picks an element called as pivot, and then it partitions the given array around the picked pivot element. It then arranges the entire array in two sub-array such that one array holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.The Divide and Conquer steps of Quicksort perform following functions. Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element. Conquer: Recursively, sort two subarrays with Quicksort Combine: Combine the already sorted array. **Merge Sort:** Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the merge() function to perform the merging. The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into twoelement lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

INPUT:	We have to generate random 100000 numbers using rand() function and use this input as 1000 blocks of 100 integer numbers to merge and quick sorting algorithms.
ALGORITHM:	**Merge sort:** 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 **Quick Sort:** Step 1: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element. Step 2: Conquer: Recursively, sort two subarrays with Quicksort. Step 3: Finally combine two already sorted arrays **Main function:** Step 1: In the main function, import the file where 100000 random numbers are stored. Step 2: Initialize the block size from 1 to 1000 and size of the block=100 so that with every increment in block, 100 numbers are added to it i.e. 100,200,300…1000. Step 3: Use clock() function and CLOCKS_PER_SEC to calculate the runtime of every block

PROGRAM:

PROGRAM 1: Merge Sort

```
#include <stdio.h>
#include<stdlib.h>
#include<time.h>
int count=0;

// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 = r - m;
    /* create temp arrays */
    int L[n1], R[n2];
    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];
    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Copy the remaining elements of L[], if there
    are any */
```

	```
	    while (i < n1) {
	        arr[k] = L[i];
	        i++;
	        k++;
	    }
	    /* Copy the remaining elements of R[], if there
	    are any */
	    while (j < n2) {
	        arr[k] = R[j];
	        j++;
	        k++;
	    }
	}

	/* l is for left index and r is right index of the
	sub-array of arr to be sorted */
	void mergeSort(int arr[], int l, int r)
	{
	    if (l < r) {
	        // Same as (l+r)/2, but avoids overflow for
	        // large l and h
	        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);
	count++;
	    }
	}


	int main()
	{
	   FILE* ptr;
	            int arr[100000];
	    // file in reading mode
	    ptr = fopen("inputFile.txt", "r");

	    if (NULL == ptr)
	```

```
    {
        printf("file can't be opened \n");
    }

    int block=1;
    int size=100;
    while(block<=1000)
    {
      int data[size];
      for(int i=0;i<size;i++)
     {
        fscanf(ptr,"%d ",&data[i]);
        //printf("%d ",data[i]);
     }

     clock_t t;
     t = clock();
     mergeSort(data,0,size-1);

    t = clock() - t;
    double time_taken = ((double)t)/CLOCKS_PER_SEC;
    printf("\n %d   %f   %d",size,time_taken,count);

    size=size+100;
    block++;
    fseek(ptr,0,SEEK_SET);

    }

     fclose(ptr);

}
```

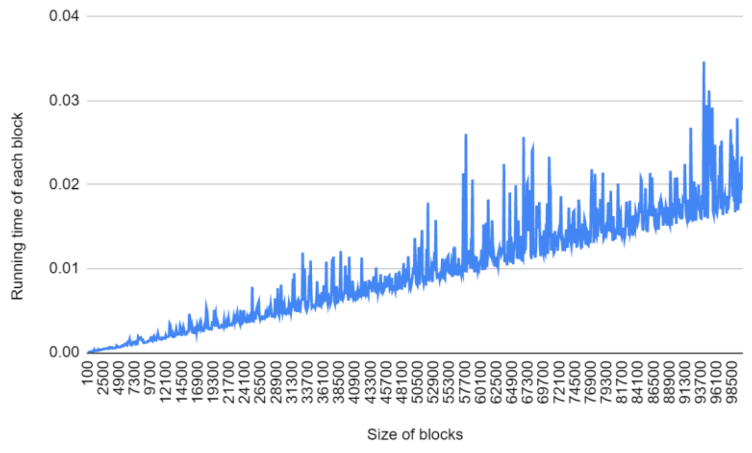
PROGRAM 2: Quick Sort

```
#include <stdio.h>
#include<stdlib.h>
#include<time.h>
int count=0;
```

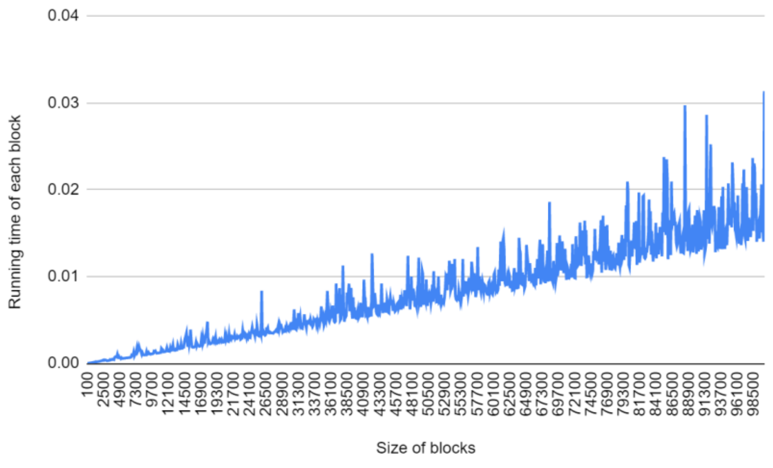
```
void swap(int *a, int *b) {
  int t = *a;
  *a = *b;
  *b = t;
}

// function to find the partition position
int partition(int array[], int low, int high) {

  // select the rightmost element as pivot
  int pivot = array[high];

  // pointer for greater element
  int i = (low - 1);

  // traverse each element of the array
  // compare them with the pivot
  for (int j = low; j < high; j++) {
    if (array[j] <= pivot) {

      // if element smaller than pivot is found
      // swap it with the greater element pointed by i
      i++;

      // swap element at i with element at j
      swap(&array[i], &array[j]);
    }
  }

  // swap the pivot element with the greater element at i
  swap(&array[i + 1], &array[high]);
  count++;
  // return the partition point
  return (i + 1);
}

void quickSort(int array[], int low, int high) {
  if (low < high) {
    // find the pivot element such that
    // elements smaller than pivot are on left of pivot
    // elements greater than pivot are on right of pivot
    int pi = partition(array, low, high);

    // recursive call on the left of pivot
    quickSort(array, low, pi - 1);

    // recursive call on the right of pivot
    quickSort(array, pi + 1, high);
  }
}

int main()
{
   FILE* ptr;
            int arr[100000];
    // file in reading mode
    ptr = fopen("inputFile.txt", "r");

    if (NULL == ptr)
    {
        printf("file can't be opened \n");
    }

    int block=1;
    int size=100;
    while(block<=1000)
    {
      int data[size];
      for(int i=0;i<size;i++)
     {
        fscanf(ptr,"%d ",&data[i]);

     }

     clock_t t;
     t = clock();
     quickSort(data,0,size-1);
    t = clock() - t;
    double time_taken = ((double)t)/CLOCKS_PER_SEC;
    printf("\n %d   %f   %d",size,time_taken,count);
    size=size+100;
    block++;
    fseek(ptr,0,SEEK_SET);
    }
     fclose(ptr);

}
```

GRAPHS:

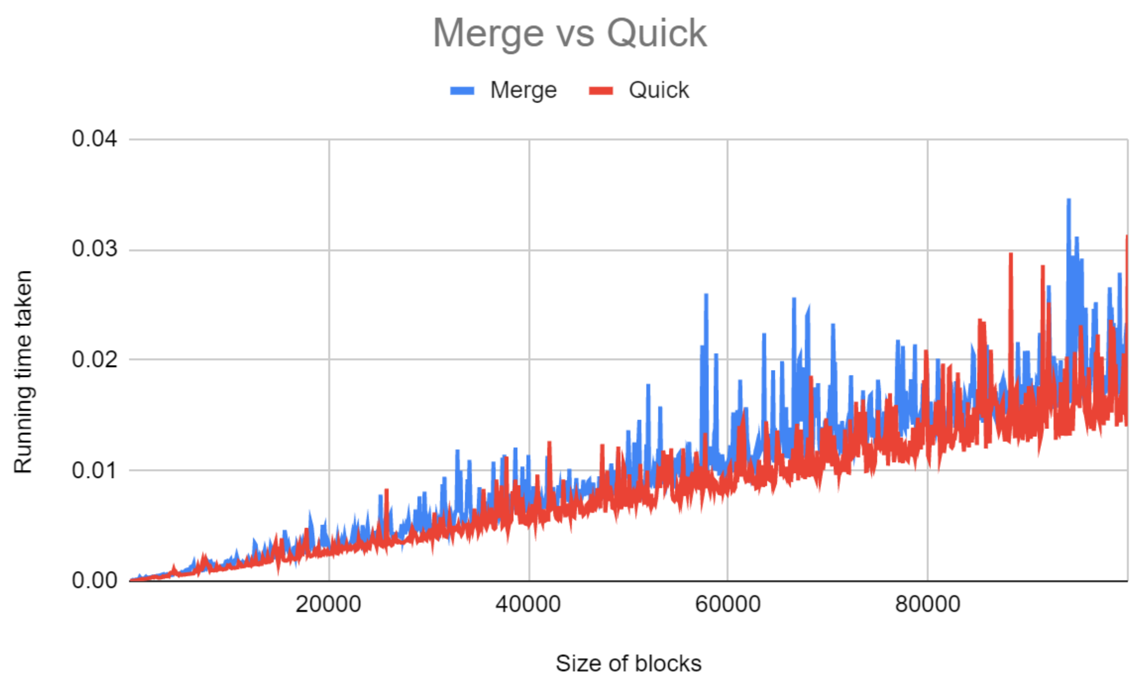
- **Merge sort:**

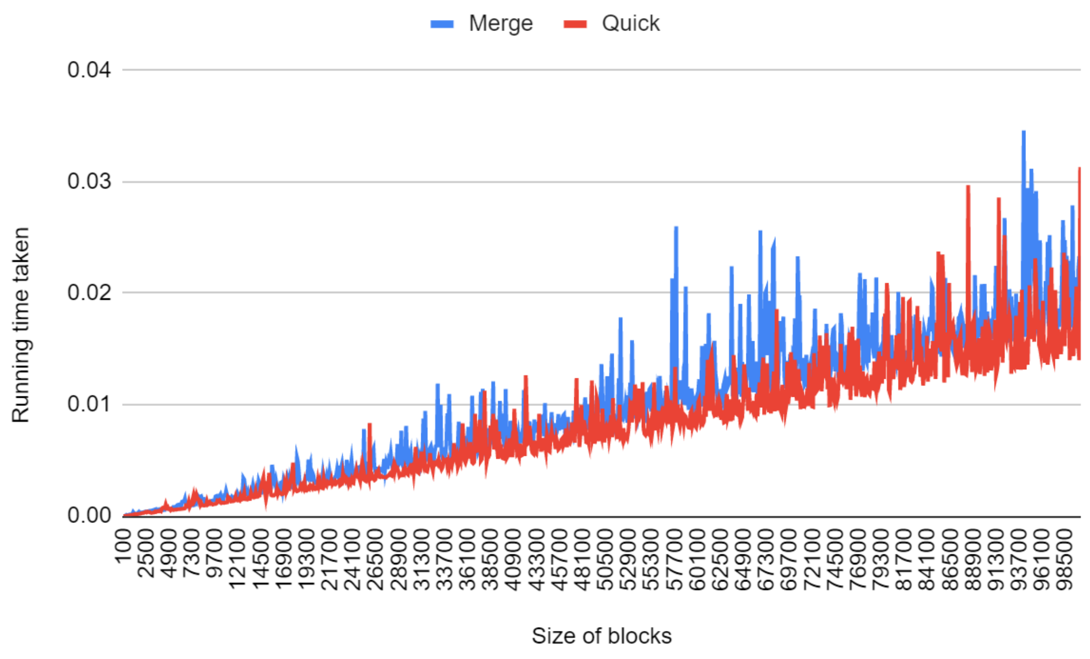


- **Quick Sort:**

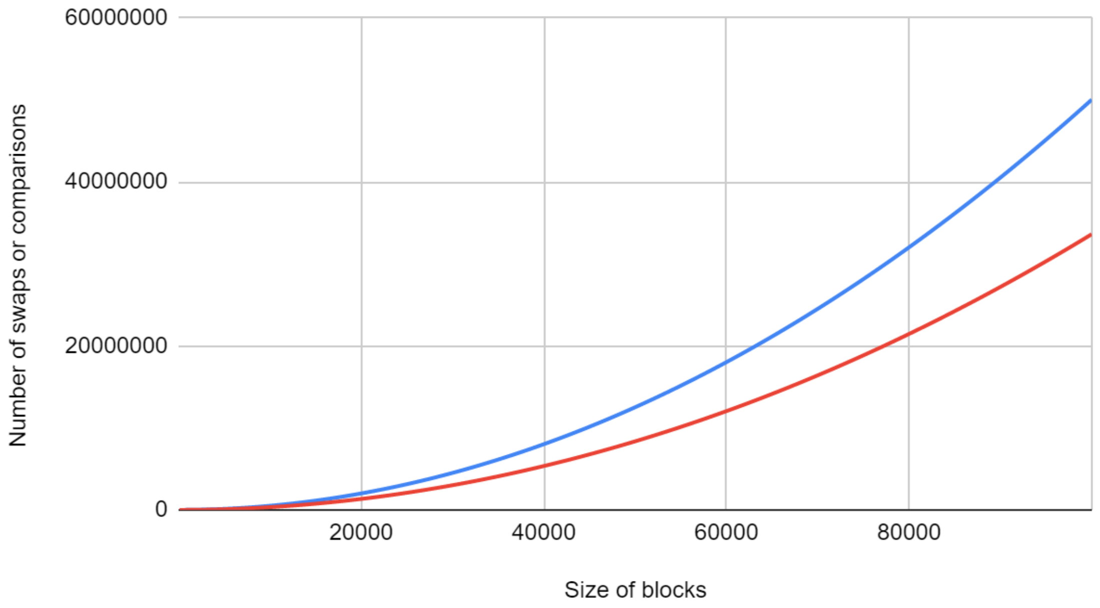


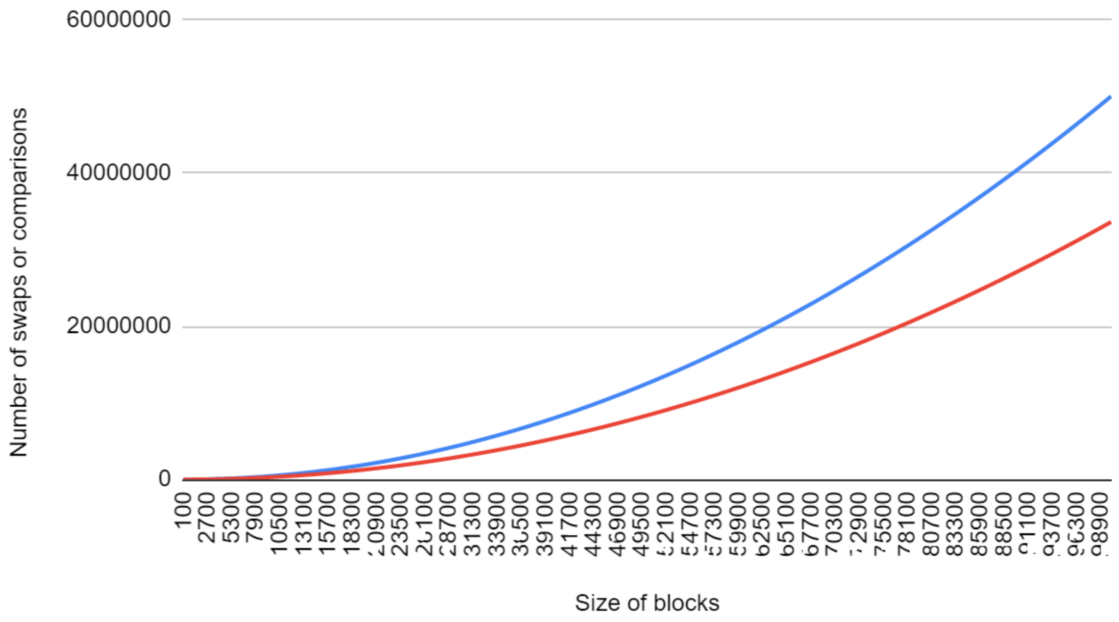
- **Quick vs Merge sort:**





- **Number of comparisons:**
-





- Quick
- Merge

RESULT ANALYSIS:	Merge sort generally performs fewer comparisons than quicksort both in the worst-case and on average. If performing a comparison is costly, merge sort will have the upper hand in terms of speed. Quicksort is found to be little faster than the merge sort as the runtime is less in quicksort. Initially both had around same runtimes for blocks but as the size of block increases merge sort algorithm takes more time to sort the array. • Time complexity: Merge: Best case: O[n log n] Worst case: O[n log n] Quick: Best case: O[n log n] Worst case: O[n^2] • Space Complexity: Merge sort: In merge sort, all elements are copied into an auxiliary array of size N, where N is the number of elements present in the unsorted array. Hence, the space complexity for Merge Sort is O[n] Quick Sort: Since quicksort calls itself on the order of log(n) times (in the average case, worst case number of calls is O(n)), at each recursive call a new stack frame of constant size must be allocated. Hence the O(log(n)) space complexity.
CONCLUSION:	In this experiment I understood about two sorting algorithms i.e. merge and quick sort and their implementation in c programming language. Also I learned about a new function which is high_resolution_clock::now() to calculate the running time.