# DESIGN AND ANALYSIS OF ALGORITHMS

## EXPERIMENT 2

**Name:** Janhavi R Deshmukh

**Branch:** CSE(DS) D1

**UID:** 2021700017

**Aim: – Experiment based on divide and conquer approach.**

**Theory:**

Merge Sort:
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.

Quick Sort:

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the 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.

**Program:**

Merge Sort:

```
#include <stdio.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;

    /* create temp arrays */
    int L[n1], R[n2];

    /* Copy data to temp arrays L[] and R[] */
```

```c
    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);
    }
}

int main()
{
   //to generate 1000000 random numbers
    /*int numbers[100000];
    for(int i=0;i<100000;i++)
    {

         numbers[i]=rand()%100000;
         printf("%d \n",numbers[i]);
    }*/

   FILE* ptr;

    // file in reading mode
    ptr = fopen("input.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);
    /* if(block<3)  //this prints sorted first 2 blocks only
     {
       printf("\n\nafter sorting block %d:\n",block);
        for(int i=0;i<size;i++)
        {
          printf("%d \n",data[i]);
         }
      }*/


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

    size=size+100;
    block++;
    fseek(ptr,0,SEEK_SET); //moving cursor again to start pointer of txt file

    }

    // fclose(ptr);

}
```

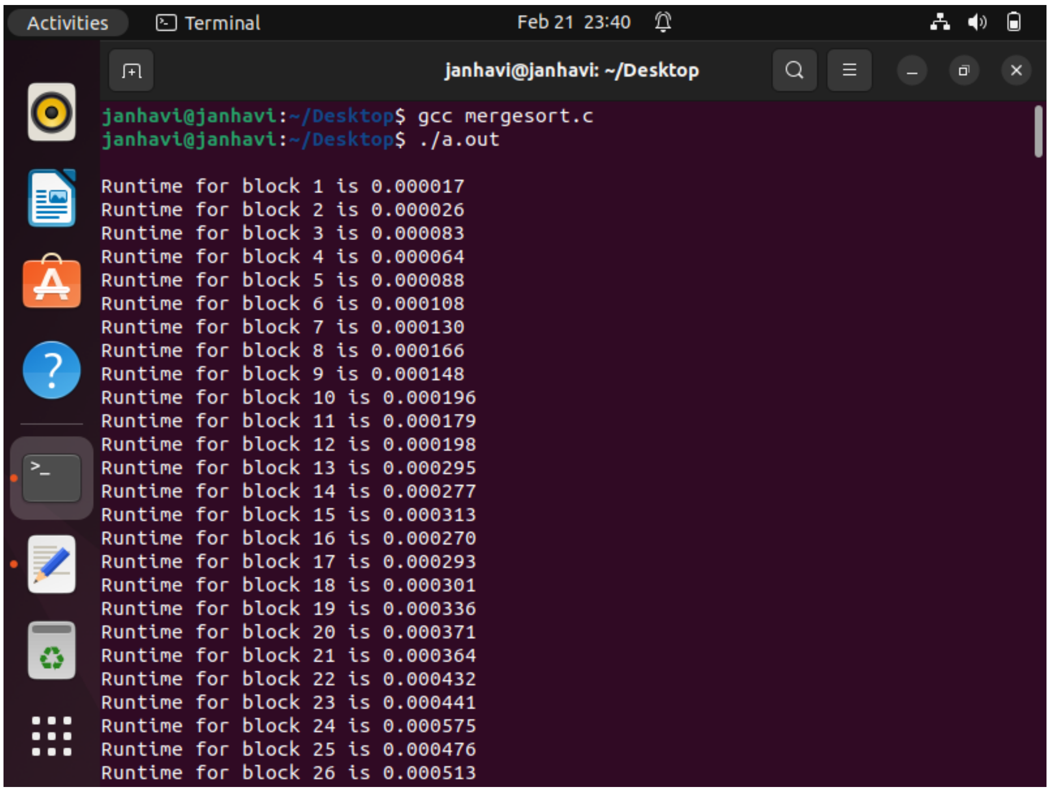
Quick sort:

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


void quicksort(int number[],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);
   }
```

```c
}

int main()
{
   //to generate 1000000 random numbers
    /*int numbers[100000];
    for(int i=0;i<100000;i++)
    {

         numbers[i]=rand()%100000;
         printf("%d \n",numbers[i]);
    }*/

   FILE* ptr;

    // file in reading mode
    ptr = fopen("input.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();
    quicksort(data,0,size-1);
    /* if(block<3)  //this prints sorted first 2 blocks only
     {
       printf("\n\nafter sorting block %d:\n",block);
        for(int i=0;i<size;i++)
        {
          printf("%d \n",data[i]);
        }
     }*/


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

    size=size+100;
    block++;
    fseek(ptr,0,SEEK_SET); //moving cursor again to start pointer of txt file

    }

    // fclose(ptr);

}
```

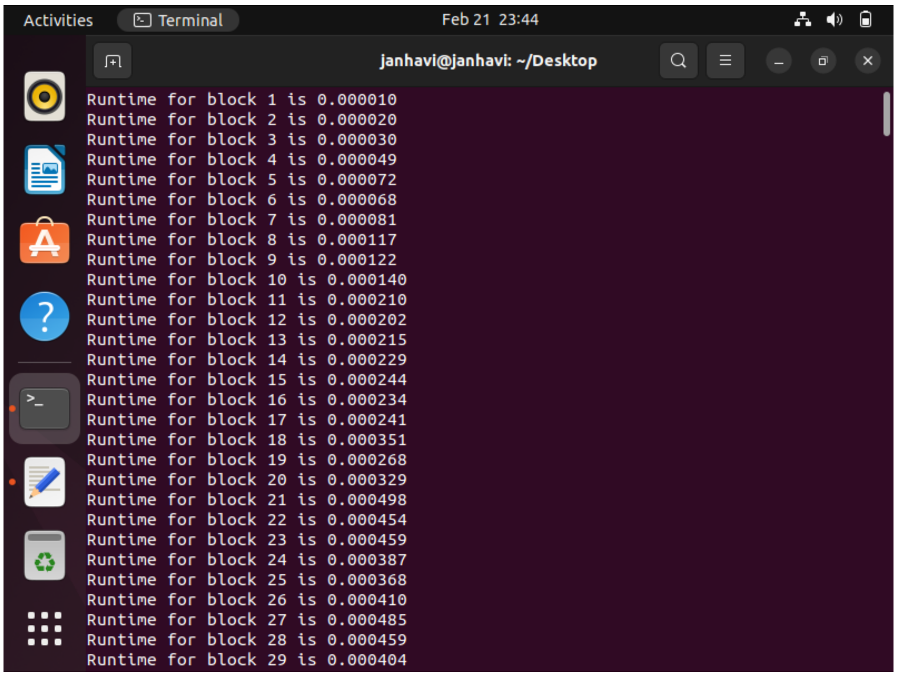
**Output:**

Merge sort:



```
janhavi@janhavi:~/Desktop$ gcc mergesort.c
janhavi@janhavi:~/Desktop$ ./a.out

Runtime for block 1 is 0.000017
Runtime for block 2 is 0.000026
Runtime for block 3 is 0.000083
Runtime for block 4 is 0.000064
Runtime for block 5 is 0.000088
Runtime for block 6 is 0.000108
Runtime for block 7 is 0.000130
Runtime for block 8 is 0.000166
Runtime for block 9 is 0.000148
Runtime for block 10 is 0.000196
Runtime for block 11 is 0.000179
Runtime for block 12 is 0.000198
Runtime for block 13 is 0.000295
Runtime for block 14 is 0.000277
Runtime for block 15 is 0.000313
Runtime for block 16 is 0.000270
Runtime for block 17 is 0.000293
Runtime for block 18 is 0.000301
Runtime for block 19 is 0.000336
Runtime for block 20 is 0.000371
Runtime for block 21 is 0.000364
Runtime for block 22 is 0.000432
Runtime for block 23 is 0.000441
Runtime for block 24 is 0.000575
Runtime for block 25 is 0.000476
Runtime for block 26 is 0.000513
```

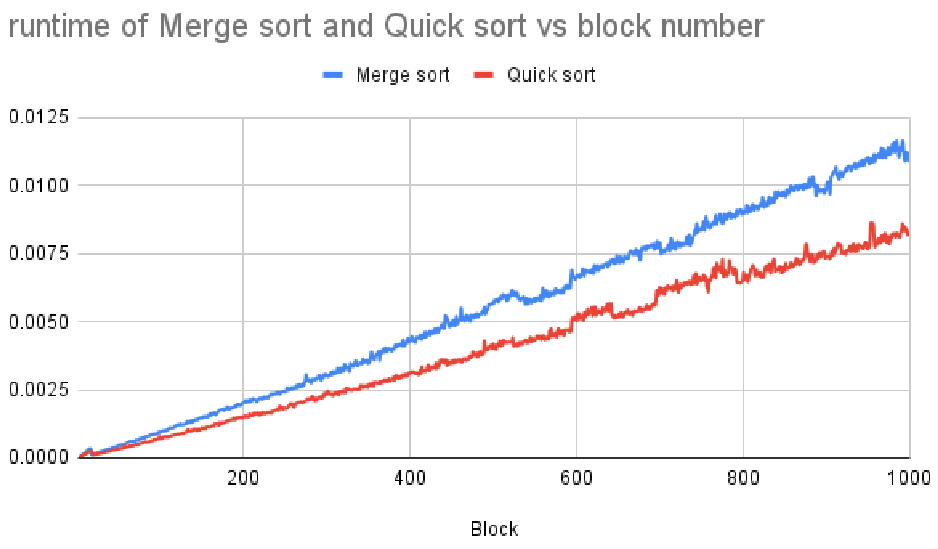
Quick sort:



```
Runtime for block 1 is 0.000010
Runtime for block 2 is 0.000020
Runtime for block 3 is 0.000030
Runtime for block 4 is 0.000049
Runtime for block 5 is 0.000072
Runtime for block 6 is 0.000068
Runtime for block 7 is 0.000081
Runtime for block 8 is 0.000117
Runtime for block 9 is 0.000122
Runtime for block 10 is 0.000140
Runtime for block 11 is 0.000210
Runtime for block 12 is 0.000202
Runtime for block 13 is 0.000215
Runtime for block 14 is 0.000229
Runtime for block 15 is 0.000244
Runtime for block 16 is 0.000234
Runtime for block 17 is 0.000241
Runtime for block 18 is 0.000351
Runtime for block 19 is 0.000268
Runtime for block 20 is 0.000329
Runtime for block 21 is 0.000498
Runtime for block 22 is 0.000454
Runtime for block 23 is 0.000459
Runtime for block 24 is 0.000387
Runtime for block 25 is 0.000368
Runtime for block 26 is 0.000410
Runtime for block 27 is 0.000485
Runtime for block 28 is 0.000459
Runtime for block 29 is 0.000404
```

- runtime for all blocks

**Graph:**



**Observation:**

From above graph ,it is clearly visible that the running time for merge sort algorithm representing blue line is more than the running time for quick sort algorithm.

In quick sort , the splitting of a array of elements is in any ratio not necessarily divided into half while in merge the array is divided into two halves. Worst case complexity for quick sort is O(n^2) and for merge is O(nlogn). Quicksort does not require additional array space while For merging of sorted sub-arrays, it needs a temporary array with the size equal to the number of input elements.Quick sort is inefficient for larger arrays .Among both of the sorting algorithm, the merge sort is fast, efficient, stable while quick sort only works efficiently when the small set of elements is involved.

**Conclusion:**

From this experiment I learnt and understood the implementation of selection and insertion sort bt performing it on 100000 randomly generated numbers by using rand() function. Runtime for quick sort is less than that of merge sort.