Experiment No. 2

Name: Om Bhutki

UID: 2022301003

Subject: DAA

Experiment No: 2

Aim – Experiment based on divide and conquers approach.

Objective: To understand the running time of algorithms by implementing two sorting algorithms based on divide and conquers approach namely Merge and Quick sort.

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.

Quick sort– Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked 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.

It is mainly divided in three basic steps:

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.

Algorithm:

step 1: start

Step2: call rand_num() function

Step 2: create rand_num file and store the random numbers in it.

Step3: open rand_num file in reading mode

Step 4: Store all random numbers in an array

Step5: Traverse all elements using for loop take n as 100

Step6: Perform insertion and selection sort on each block of 100 numbers

Step7: Calculate time required to perform insertion and selection sort at each iteration

Step8: Increment n by 100

Step 9: If n reaches 1000 then end else go to step 6 rand_num() function:

step 1: start

step 2: crate the file pointer

step 3: open the file in writing mode

step 3: starts the loop from 0 to 100000

step 4: insert the 100000 random numbers in the file

step 5: close the file handle

step 6: end

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:

Algorithm for choosing pivot:

Step 1 − Choose the highest index value as pivot

Step 2 − Take two variables to point left and right of the list excluding pivot

Step 3 − left points to the low index

Step 4 − right points to the high

Step 5 − while value at left is less than pivot move right

Step 6 − while value at right is greater than pivot move left

Step 7 − if both step 5 and step 6 does not match swap left and right

Step 8 − if left ≥ right, the point where they met is new pivot

Algorithm for quick sort:

Step 1 − Make the right-most index value pivot

Step 2 − partition the array using pivot value

Step 3 − quicksort left partition recursively

Step 4 − quicksort right partition recursively

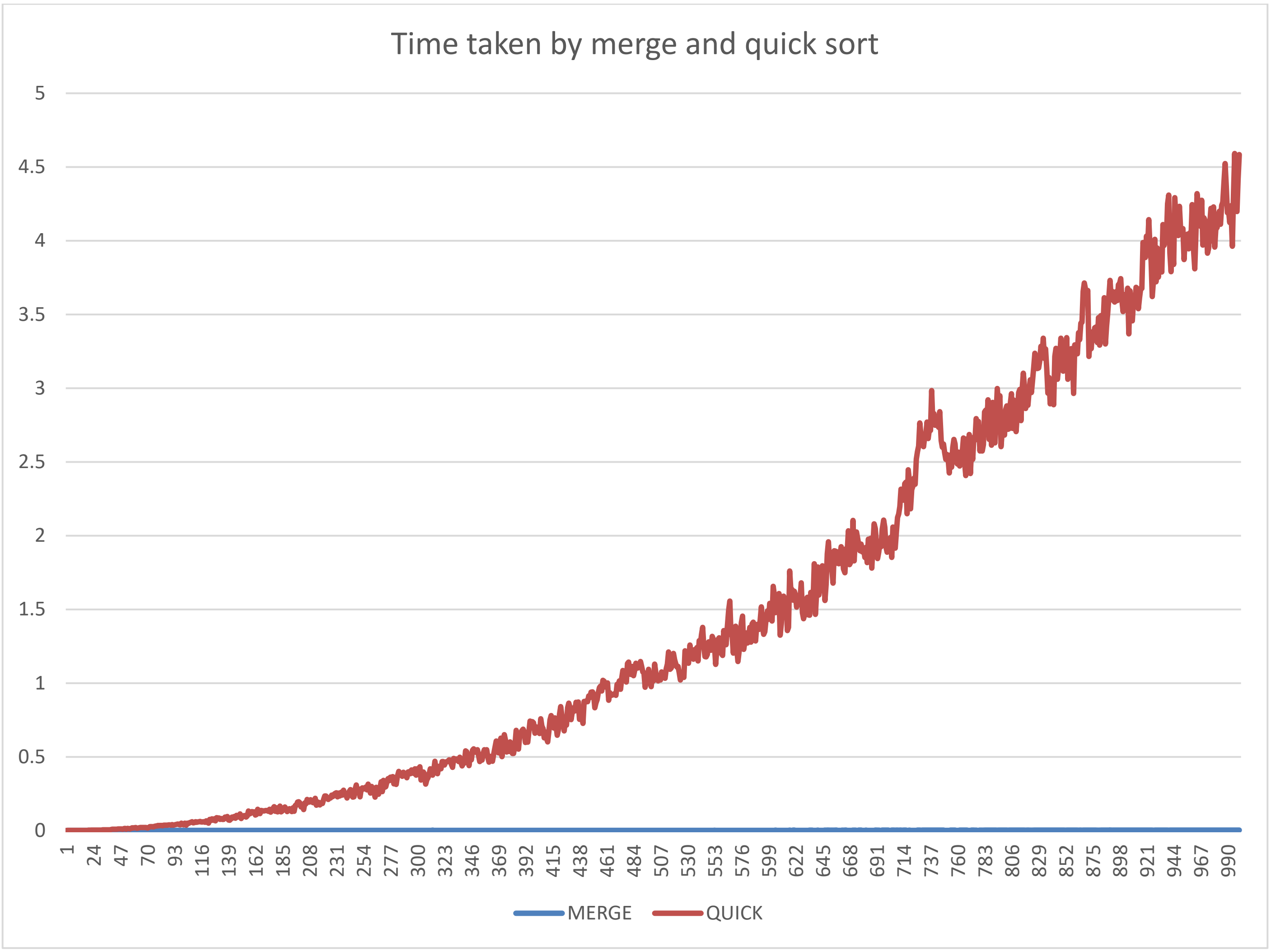
Program:

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

int * rand_num(){
   static int arr[100000];
   int i;
   for (i = 0; i < 100000; ++i){
      arr[i] = rand();
   }

   return arr;
}

void merge(int arr[], int left,
            int middle, int right)
{
 int i, j, k;
 int n1 = middle - left + 1;
 int n2 = right - middle;

 int Left[n1], Right[n2];

 for (i = 0; i < n1; i++)
            Left[i] = arr[left + i];
 for (j = 0; j < n2; j++)
            Right[j] = arr[middle + 1 + j];


 i = 0;


 j = 0;


 k = left;
 while (i < n1 && j < n2)
 {
            if (Left[i] <= Right[j])
            {
                     arr[k] = Left[i];
                     i++;
```

```
        }
        else
        {
                arr[k] = Right[j];
                j++;
        }
        k++;
 }


 while (i < n1) {
        arr[k] = Left[i];
        i++;
        k++;
 }


 while (j < n2)
 {
        arr[k] = Right[j];
        j++;
        k++;
 }
}

void mergeSort(int arr[],
                int left, int right)
{
 if (left < right)
 {

        int m = left + (right - left) / 2;
        mergeSort(arr, left, m);
        mergeSort(arr, m + 1, right);

        merge(arr, left, m, right);
 }
}

void swap(int *xp, int *yp)
{
   int temp = *xp;
   *xp = *yp;
   *yp = temp;
}
```

```
void printArray(int arr[], int n)
{
    int i;
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

int divide(int array[], int low, int high) {

  int pivot = array[high];

  int i = (low - 1);
  for (int j = low; j < high; j++) {
    if (array[j] <= pivot) {

      i++;
      swap(&array[i], &array[j]);
    }
  }

  swap(&array[i + 1], &array[high]);
  return (i + 1);
}

void quickSort(int array[], int low, int high) {
  if (low < high) {


    int pi = divide(array, low, high);


    quickSort(array, low, pi - 1);
    quickSort(array, pi + 1, high);
  }
}
int main()
{
     int *x1;
     int i,i1;
     int arr[100000];
     x1 = rand_num();
     FILE *fp;
    int ch;
     fp = fopen("rand_num.txt","w");
     for (i1 = 0; i1 < 100000; ++i1){
          fprintf(fp,"%d\n",*(x1 + i1));
        }

        fp = fopen("rand_num.txt", "r");



          for ( i = 0; i < 100000; i++)
          {
              fscanf(fp,"%d\n",&arr[i]);
          }

        FILE *file = fopen("output.txt","w");



        int num = 100;
        for ( i = 0; i < 1000; i++)
        {
          clock_t t1 = clock();
          mergeSort(arr,0,num);
          clock_t t2 = clock();
          clock_t t3 = clock();
          quickSort(arr, 0, num);
          clock_t t4 = clock();
          double mergeSort_time = (double)(t2-t1)/(double)CLOCKS_PER_SEC;
          double quickSort_time = (double)(t4-t3)/(double)CLOCKS_PER_SEC;
          fprintf(file,"%d\t",i+1);
          fprintf(file,"%f\t",mergeSort_time);
          fprintf(file,"%f\n",quickSort_time);

          num += 100;
        }
        fclose(fp);
        fclose(file);
        return 0;
}
```

Result:



Inference:

Merge sort: As we can see from the graph, merge sort takes significantly less time than quick sort. We can see some spikes in time towards the end of the graph but there aren't any drastic change of values during the course of its execution. Overall, it takes between 0 to 1 ms to sort the values during each iteration.

Quick sort: On the other hand, time taken to perform quick sort keeps increasing as we add more numbers to sort during each iteration. Also, there are a lots of throttles in the graph as the amount of numbers increases. Quick sort also reaches as high as almost 5 ms and despite all the throttling the graph has a relatively linear increase.

Also, these values differ from system to system depending upon hardware configuration, operating system, memory, cpu speed etc.

Conclusion:

Thus, after performing this experiment we conclude that merge sort is in fact the better divide and conquer algorithm than quick sort for sorting large data and the size of input matters a lot when measuring performance of a sorting algorithm.