DAA EXPERIMENT NO. 2

NAME: Glen Dsouza CSE DS (BATCH A)

UID: 2022701002

AIM: Experiment based on divide and conquer approach.

Problem Definition & Assumptions – 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.

ALGORITHM:

Quick Sort Function:

Step 1: Start.

Step 2: Check if the left index is less than the right index.

Step 3: Select the last element of the array (arr[right]) as the pivot element.

Step 4: Initialize a variable i to left - 1.

Step 5: Iterate over the sub-array from left to right-1. a. If the current element (arr[j]) is less than the pivot element, increment i and swap arr[i] and arr[j].

Step 6: Swap arr[i+1] and arr[right] to place the pivot element in its correct position.

Step 7: Set p to i + 1, the index of the pivot element.

Step 8: Recursively call quickSort() on the left sub-array, from left to p-1.

Step 9: Recursively call quickSort() on the right sub-array, from p+1 to right.

Step 10: Stop.

Merge Sort Function:

Step 1: Start.

Step 2: Declare an array and left, right, mid variable.

Step 3: Perform merge function.

mergesort(array,left,right)

```
mergesort (array, left, right)
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.

Main Function:

Step 1: Start

Step 3: In the main function, open a file "exp2.txt" for writing and initialize the random number generator with srand((unsigned int) time(NULL)).

Step 4: Generate 1000 blocks of 100 random numbers each and store them in the file.

Step 5: Close the file after writing.

Step 6: Open the file "exp2.txt" for reading.

Step 7: For each block of 100 elements, read the elements from the file into two arrays arr and arr1.

Step 8: Sort the elements in the arr using the quick_sort function.

Step 9: Measure the time taken for sorting using the clock() function and store it in the time_taken_quick_sort variable.

Step 10: Sort the elements in the arr1 using the merge_sort function.

Step 11: Measure the time taken for sorting using the clock() function and store it in the time_taken_merge_sort variable.

Step 12: Print the block number, time taken for quick sort, and time taken for merge sort.

Step 13: Repeat the process for 1000 blocks.

Step 14: Close the file after reading.

Step 15: Stop.

CODE:

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

void quickSort(int arr[], int left, int right) {
  if (left < right) {
    int pivot = arr[right];
    int i = left - 1;
    for (int j = left; j < right; j++) {
      if (arr[j] < pivot) {
        i++;
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
      }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[right];
    arr[right] = temp;
    int p = i + 1; // p is the pivot element
    quickSort(arr, left, p - 1);
    quickSort(arr, p + 1, right);
  }
}

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;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

void main() {
    FILE *fp;
    fp  = fopen ("exp2.txt", "w");
    srand((unsigned int) time(NULL));
    for(int block=0;block<1000;block++) {
        for(int i=0;i<100;i++) {
            int number = (int)(((float) rand() / (float)(RAND_MAX))*100000);
            fprintf(fp,"%d ",number);
        }
        fputs("\n",fp);
    }
    fclose (fp);
   fp = fopen("exp2.txt", "r");
   printf("Block\tQuick_sort\tMerge_sort\n");
   for(int block=0;block<1000;block++) {
    clock_t t,t1;
    int arr[(block+1)*100];
    int arr1[(block+1)*100];
    for(int i=0;i<(block+1)*100;i++){
        fscanf(fp, "%d", &arr[i]);
        arr1[i] = arr[i];
    }
    fseek(fp, 0, SEEK_SET);
    t = clock();
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    t = clock() - t;
    t1 = clock();
    n = sizeof(arr1) / sizeof(arr1[0]);
    mergeSort(arr1, 0, n - 1);
    t1 = clock() - t1;
    double time_taken_quick_sort = ((double)t)/CLOCKS_PER_SEC;
    double time_taken_merge_sort = ((double)t1)/CLOCKS_PER_SEC;
```

```
    printf("%d\t%f\t%f\n",(block+1),time_taken_quick_sort, time_taken_merge_sort);
   }
    fclose(fp);
}
```

OUTPUT:

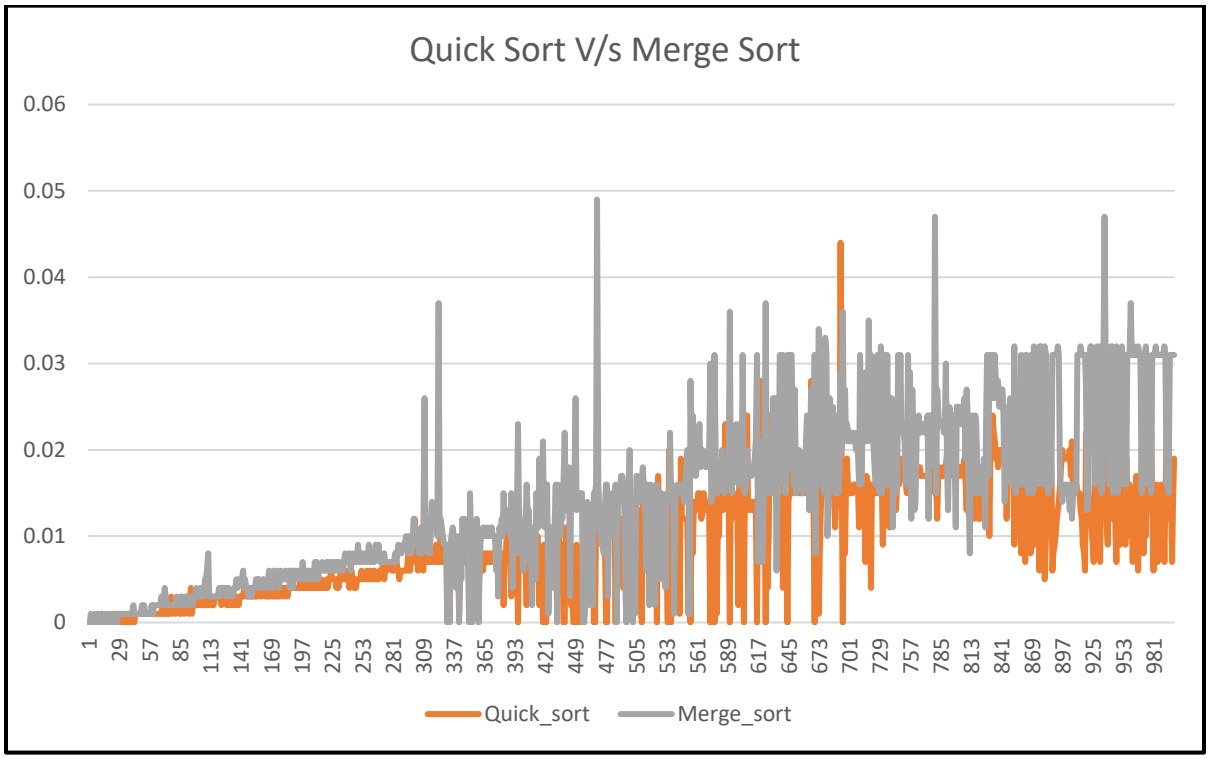
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS D:\Documents\Desktop\Glen\S.P.I.T\2nd Year\SEM IV\DAA\PRACS> cd "d:\Documents\Desktop\Glen\S.P.I.T\2nd Year\SEM IV\DAA\PRACS\" ; if ($?) { gcc EXP2.c -o EXP2 } ; if ($?) { .\EXP2
}
Block   Quick_sort      Merge_sort
1       0.000000        0.000000
2       0.000000        0.000000
3       0.000000        0.001000
4       0.000000        0.000000
5       0.000000        0.000000
6       0.000000        0.000000
7       0.001000        0.000000
8       0.000000        0.001000
9       0.000000        0.001000
10      0.000000        0.001000
11      0.000000        0.000000
12      0.000000        0.000000
13      0.001000        0.000000
14      0.000000        0.000000
15      0.001000        0.000000
16      0.000000        0.001000
17      0.000000        0.000000
18      0.000000        0.001000
19      0.000000        0.000000
20      0.000000        0.001000
21      0.000000        0.000000
22      0.000000        0.001000
23      0.001000        0.000000
24      0.001000        0.000000
25      0.001000        0.000000
26      0.000000        0.001000
27      0.001000        0.000000
28      0.000000        0.001000
29      0.000000        0.001000
30      0.001000        0.001000
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

970     0.020000        0.015000
971     0.021000        0.025000
972     0.008000        0.031000
973     0.016000        0.031000
974     0.010000        0.032000
975     0.019000        0.016000
976     0.013000        0.031000
977     0.020000        0.016000
978     0.022000        0.024000
979     0.022000        0.015000
980     0.014000        0.031000
981     0.006000        0.031000
982     0.016000        0.031000
983     0.008000        0.032000
984     0.016000        0.031000
985     0.007000        0.031000
986     0.015000        0.031000
987     0.008000        0.031000
988     0.016000        0.031000
989     0.008000        0.031000
990     0.015000        0.031000
991     0.007000        0.032000
992     0.014000        0.031000
993     0.022000        0.016000
994     0.022000        0.023000
995     0.021000        0.015000
996     0.012000        0.031000
997     0.017000        0.031000
998     0.007000        0.031000
999     0.014000        0.031000
1000    0.019000        0.031000
```

RESULT:



RESULT ANALYSIS:

The following graph is representation of amount of time (in seconds) required to sort block of integers using Quick sort & Merge sort algorithm.

In the above graph, time values of sorting algorithm are plotted on y-axis against no. of blocks on x-axis. The maximum no. of block is 1000 on X-axis.

Maximum amount of time required to sort 1000^{th} block using quick sort is approx. 0.019 seconds and using merge sort is 0.031 seconds.

Quick sort and Merge sort require almost similar with little variation of time. Comparatively, merge sort requires slightly more time than quick sort.

CONCLUSION: In this experiment quick sort & merge sort were implemented and their runtime across 1000 block of 100 integers was plotted on a graph.