DESIGN AND ANALYSIS OF ALGORITHMS LAB

NAME: Mohammed Saad Belgi

UID: 2021700005

BATCH: A

BRANCH: CSE DS

EXPT. NO.: 2

AIM: Experiment based on divide and conquers approach.

ALGORITHM:

Merge sort:

MERGE-SORT(ARR, lo, hi):

```
1.  if hi>lo
2.     mid ← (lo + hi) / 2
3.     MERGE-SORT(ARR, lo, mid)
4.     MERGE-SORT(ARR, mid+1, hi)
5.     MERGE(ARR, lo, mid, hi)
```

MERGE(A, p, q, r):

```
1.  n1 ← q-p+1
2.  n2 ← r-1
3.  let L[1…n1+1] and R[1…n2+1] be new arrays
4.  for i = 1 to n1
5.      L[i] ← A[p+i-1]
6.  for j = 1 to n2
7.      R[j] ← A[q+j]
8.  L[n1+1] ← inf
9.  R[n2+1] ← inf
10. i ← 1
11. j ← 1
12. for k = p to r
13.    if L[i] <= R[j]
14.        A[k] ← L[i]
15.        i ← i + 1
16.    else A[k] ← R[j]
17.          j ← j + 1
```

Quick sort:

QUICKSORT(A,p,r):

```
1.     If p<r
```

```
2.     q ← PARTITION(A,p,r)
3.     QUICKSORT(A, p, q-1)
4.     QUICKSORT(A, q, r)
```

PARTITON(A,p,r):

```
1. x ← A[r]
2. i ← p – 1
3. for j = p to r – 1:
4.     i ← i+1
5.     exchange A[i] with A[j]
6. exchange A[i+1] with A[r]
7. return i+1
```

CODE:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>    // required to use clock function if executing program on linux
#ifdef _WIN32        // to check if OS is windows (_WIN32 is a macro defined on every windows based gcc compiler)
#include <windows.h> // required to use windows api for time measurement if executing on windows
#endif

// prototypes:
void mergeSort(int *, int);
void quickSort(int *, int);
void mergeSortActual(int *, int, int, int *);
void merge(int *, int, int, int, int *);
void quickSortActual(int *, int, int);
int partition(int *, int, int);
void deepCopy(int *, int *, int);
void printArray(int *, int);
void generateRandomNumbers();
void LinuxMain();
#ifdef _WIN32
void WindowsMain();
#endif

int compareCounter; // global variable to keep count of comparison operations

int main()
```

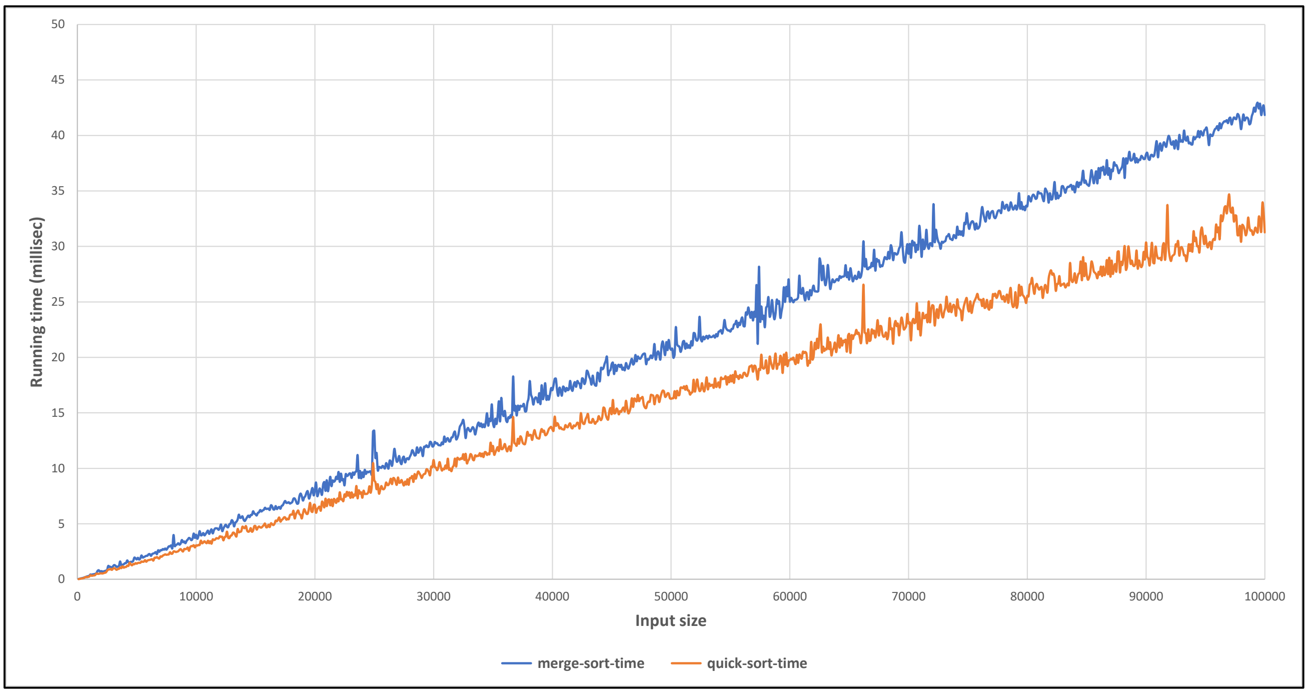
```c
{
    // generateRandomNumbers();
#ifdef _WIN32
    WindowsMain();
#else
    LinuxMain();
#endif
    return 0;

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

void mergeSort(int *arr, int len)
{
    compareCounter = 0;
    int holder[len];
    int mid = (len - 1) / 2;
    mergeSortActual(arr, 0, mid, holder);
    mergeSortActual(arr, mid + 1, len - 1, holder);
    merge(arr, 0, mid, len - 1, holder);
}

void mergeSortActual(int *arr, int low, int high, int
*holder)
{
    if (low < high)
    {
        int mid = (low + high) / 2;
        mergeSortActual(arr, low, mid, holder);
        mergeSortActual(arr, mid + 1, high, holder);
        merge(arr, low, mid, high, holder);
    }
}

void merge(int *arr, int low, int mid, int high, int
*holder)
{
    int len = high - low + 1;
    int i = low, j = mid + 1;
    int temp_idx = i;
    while (i <= mid && j <= high)
    {
        compareCounter++;
        if (arr[i] < arr[j])
            holder[temp_idx++] = arr[i++];
        else
            holder[temp_idx++] = arr[j++];
    }
    if (i > mid)
    {
        while (j <= high)
            holder[temp_idx++] = arr[j++];
    }
    else if (j > high)
    {
        while (i <= mid)
            holder[temp_idx++] = arr[i++];
    }
    for (int i = low; i <= high; i++)
        arr[i] = holder[i];
}

void quickSort(int *arr, int len)
{
    compareCounter = 0;
    int part = partition(arr, 0, len - 1);
    quickSortActual(arr, 0, part);
    quickSortActual(arr, part + 1, len - 1);
}

void quickSortActual(int *arr, int low, int high)
{
    if (low < high)
    {
        int part = partition(arr, low, high);
        quickSortActual(arr, low, part - 1);
        quickSortActual(arr, part + 1, high);
    }
}
```

```c
int partition(int *arr, int low, int high)
{
    int pivot = arr[low];

    // Hoare partition scheme:
    // int i = low + 1, j = high, temp;
    // while (j > i)
    // {
    //     compareCounter++;
    //     while (i < high && pivot > arr[i])
    //         i++;
    //     compareCounter++;
    //     while (j > low && pivot <= arr[j])
    //         j--;
    //     if (i < j)
    //     {
    //         temp = arr[i];
    //         arr[i] = arr[j];
    //         arr[j] = temp;
    //     }
    // }
    // arr[low] = arr[j];
    // arr[j] = pivot;
    // return j;

    // Lomuto partition scheme:
    int i = low, temp;
    for (int j = low + 1; j <= high; j++)
    {
        compareCounter++;
        if (arr[j] < pivot)
        {
            i++;
            temp = arr[j];
            arr[j] = arr[i];
            arr[i] = temp;
        }
    }
    arr[low] = arr[i];
    arr[i] = pivot;
    return i;
}
```

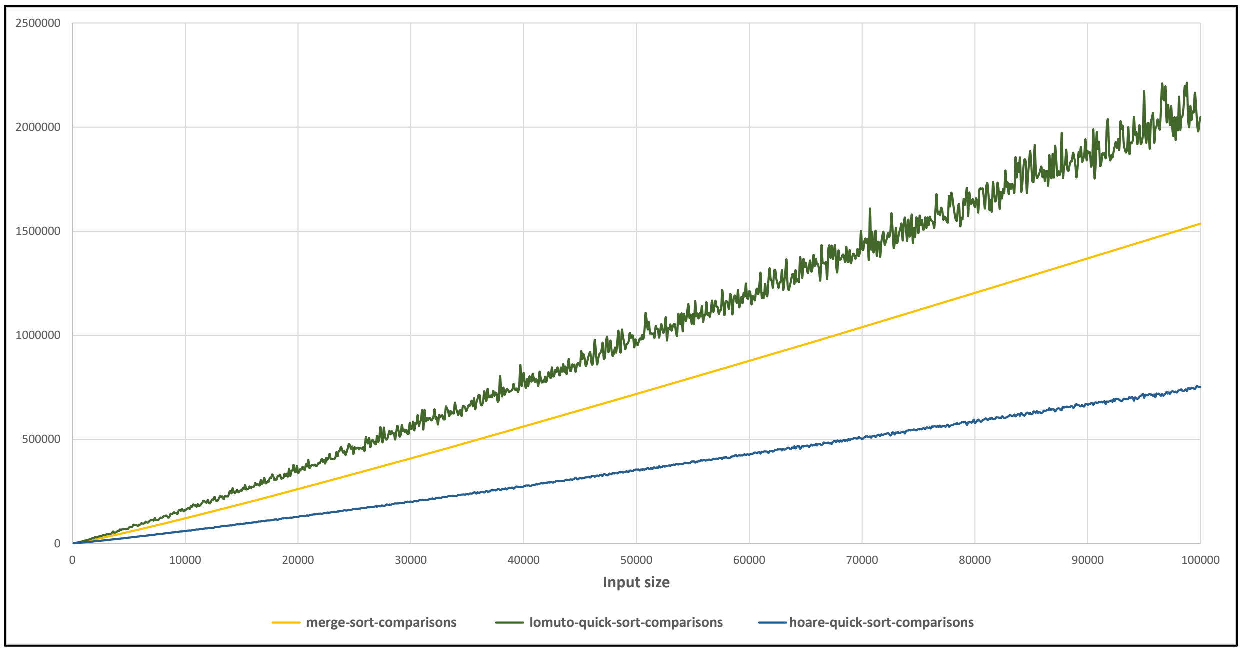
```c
void deepCopy(int *source, int *dest, int len)
{
    for (int i = 0; i < len; i++)
        dest[i] = source[i];
}

void generateRandomNumbers()
{
    FILE *fptr = fopen("rand_num.txt", "w");
    time_t cur_time;
    srand((unsigned int)time(&cur_time));
    for (int i = 0; i < 100000; i++)
        fprintf(fptr, "%d\n", rand());
    fclose(fptr);
}

#ifdef _WIN32
void WindowsMain()
{
    FILE *rand_num = fopen("rand_num.txt", "r");
    FILE *dest = fopen("output.txt", "w");
    fprintf(dest, "size    | merge-sort-time | quick-sort-
time | merge-sort-comparisons | quick-sort-
comparisons\n");
    double time1, time2;
    LARGE_INTEGER clock_freq, start, end;
    int mergeComparisons, quickComparisons;
    QueryPerformanceFrequency(&clock_freq);
    for (int size = 100; size <= 100000; size += 100)
    {
        int arr1[size];
        int arr2[size];
        for (int j = 0; j < size; j++)
            fscanf(rand_num, "%d", &arr1[j]);
        fseek(rand_num, 0, SEEK_SET);
        deepCopy(arr1, arr2, size);

        compareCounter = 0;
        QueryPerformanceCounter(&start);
        mergeSort(arr1, size);
        QueryPerformanceCounter(&end);
        time1 = (double)(end.QuadPart - start.QuadPart) *
1000.0 / clock_freq.QuadPart;
        mergeComparisons = compareCounter;

        QueryPerformanceCounter(&start);
        quickSort(arr2, size);
        QueryPerformanceCounter(&end);
        time2 = (double)(end.QuadPart - start.QuadPart) *
1000.0 / clock_freq.QuadPart;
        quickComparisons = compareCounter;

        fprintf(dest, "%6d | %15.4f | %15.4f | %22d |
%22d\n", size, time1, time2, mergeComparisons,
quickComparisons);
        printf("Size %d done!\n", size);
    }
    fclose(rand_num);
    fclose(dest);
}
#endif

void LinuxMain()
{
    FILE *rand_num = fopen("rand_num.txt", "r");
    FILE *dest = fopen("output.txt", "w");
    fprintf(dest, "size   | merge-sort-time | quick-sort-
time | merge-sort-comparisons | quick-sort-
comparisons\n");
    double time1, time2;
    clock_t start, end;
    int mergeComparisons, quickComparisons;
    for (int size = 100; size <= 100000; size += 100)
    {
        int arr1[size];
        int arr2[size];
        for (int j = 0; j < size; j++)
            fscanf(rand_num, "%d", &arr1[j]);
        fseek(rand_num, 0, SEEK_SET);
        deepCopy(arr1, arr2, size);

        start = clock();
        mergeSort(arr1, size);
        end = clock();
        time1 = (double)(end - start) * 1000.0 /
CLOCKS_PER_SEC;
        mergeComparisons = compareCounter;

        start = clock();
        quickSort(arr2, size);
        end = clock();
        time2 = (double)(end - start) * 1000.0 /
CLOCKS_PER_SEC;
        quickComparisons = compareCounter;

        fprintf(dest, "%6d | %15.4f | %15.4f | %22d |
%22d\n", size, time1, time2, mergeComparisons,
quickComparisons);
        printf("Size %d done!\n", size);
    }
    fclose(rand_num);
    fclose(dest);
}
```

PLOTTING THE DATA OBTAINED AFTER EXECUTION IN EXCEL:

Graph of running time of quick sort (Lomuto partitioning scheme) and merge sort vs input size:



Graph of comparison operations of quick sort (using Lomuto and Hoare partitioning schemes) and merge sort vs input size:



RESULT ANALYSIS:

Quick sort (implemented using Lomuto partitioning scheme) beats merge sort for almost all input sizes.

Number of comparison operations required by merge sort is exactly linearly proportional to the input size. When quick sort is implemented using Lomuto partitioning scheme, number of comparisons required is higher than merge sort; but if Hoare's partitioning scheme is used, fewer comparisons are required than merge sort.

CONCLUSION:

Quick sort performs better than merge sort for randomised input of size smaller than 100000.