Name: Yash Ravindra Burad

UID: 2022301004

Experiment no.: 2

Class: Comps A

Subject: DAA

Aim: To find out running time of 2 sorting algorithms like Merge sort and Quick sort.

Theory:

Merge Sort:

Merge sort is a comparison-based sorting algorithm that employs the divide-and-conquer strategy. In the divide-and-conquer approach, the problem is divided into multiple subproblems, solved individually, and finally, the result of the subproblems are combined to form the final solution.

In merge sort, we divide the array into two smaller subarrays of equal size or with a size difference of one, depending on the parity of the array's length. Each subarray is further divided into two smaller subarrays again and again recursively until we get subarrays of size one. We then sort the subarrays and merge them to produce the sorted array.

Quicksort is a comparison-based sorting algorithm. Like merge sort, this is also based on the divide-and-conquer strategy. The algorithm has two basic operations — swapping items in place and partitioning a section of the array.

Quicksort sorts an array by choosing a pivot element and then partitioning the rest of the elements around the pivot. All the elements less than the pivot are moved to the left side of the pivot (called left partition), and the elements greater than or equal to the pivot are moved to the right of the pivot (called right partition).

The sorting is continued on left and right partitions separately and recursively by choosing pivot points and breaking down the partitions into single-element subarrays before combining them to form one sorted list.

Algorithm:

Main function:
step 1: start

Step2: call generate_numbers() function

Step 2: call operation()function

Step 3: end

generate_numbers() 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

operation function():
step 1: start

step 2: open the file in reading mode

step 3: start the loop from 0 to 100000 and increment it with 100

step 4: create two arrays

step 5: start the loop from 0 to j and scan the data from file

step 6: before sorting store the time

step 7: perform selection sort

step 8: check the time after after the sorting

step 9: calculate the time taken by the algorithm

step 10: before sorting store the time

step 11: perform selection sort

step 12: check the time after after the sorting

step 13: calculate the time taken by the algorithm

Merge Sort:

```
MERGE_SORT(arr, beg, end)

if beg < end
set mid = (beg + end)/2
MERGE_SORT(arr, beg, mid)
MERGE_SORT(arr, mid + 1, end)
MERGE (arr, beg, mid, end)
end of if

END MERGE_SORT
```

```
void merge(int a[], int beg, int mid, int end)
{
    int i, j, k;
    int n1 = mid - beg + 1;
    int n2 = end - mid;

    int LeftArray[n1], RightArray[n2]; //temporary arrays
         /* copy data to temp arrays */
   for (int i = 0; i < n1; i++)
    LeftArray[i] = a[beg + i];
    for (int j = 0; j < n2; j++)
    RightArray[j] = a[mid + 1 + j];

    i = 0, /* initial index of first sub-array */
    j = 0; /* initial index of second sub-array */
    k = beg;  /* initial index of merged sub-array */
```

```
  while (i < n1 && j < n2)
    {
        if(LeftArray[i] <= RightArray[j])
        {
            a[k] = LeftArray[i];
            i++;
        }
        else
        {
            a[k] = RightArray[j];
            j++;
        }
        k++;
    }
    while (i<n1)
    {
        a[k] = LeftArray[i];
        i++;
        k++;
    }

    while (j<n2)
    {
        a[k] = RightArray[j];
        j++;
        k++;
    }
}
```

Quick Sort:

QUICKSORT (array A, start, end)

```
{
  if (start < end)
  {
 p = partition(A, start, end)
 QUICKSORT (A, start, p - 1)
 QUICKSORT (A, p + 1, end)
 }
}
PARTITION (array A, start, end)
{
  pivot ? A[end]
  i ? start-1
  for j ? start to end -1 {
  do if (A[j] < pivot) {
  then i ? i + 1
  swap A[i] with A[j]
   }}
  swap A[i+1] with A[end]
  return i+1
}
```

Program:

```
#include<stdio.h>

#include<math.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[]
    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]
    // Initial index of first subarray
    i = 0;

    // Initial index of second subarray
    j = 0;

    // Initial index of merged subarray
    k = l;
    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++;
    }
}

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);
    }
```

```
}


void quicksort(int number[100000],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);
   }
}


void gen_arr()
{

  FILE *ptr;
  ptr=fopen("number.txt","w");
  for(int i=0;i<100000;i++)
  {
     fprintf(ptr,"%d\n",rand() % 100000);
  }
  fclose(ptr);
}
```

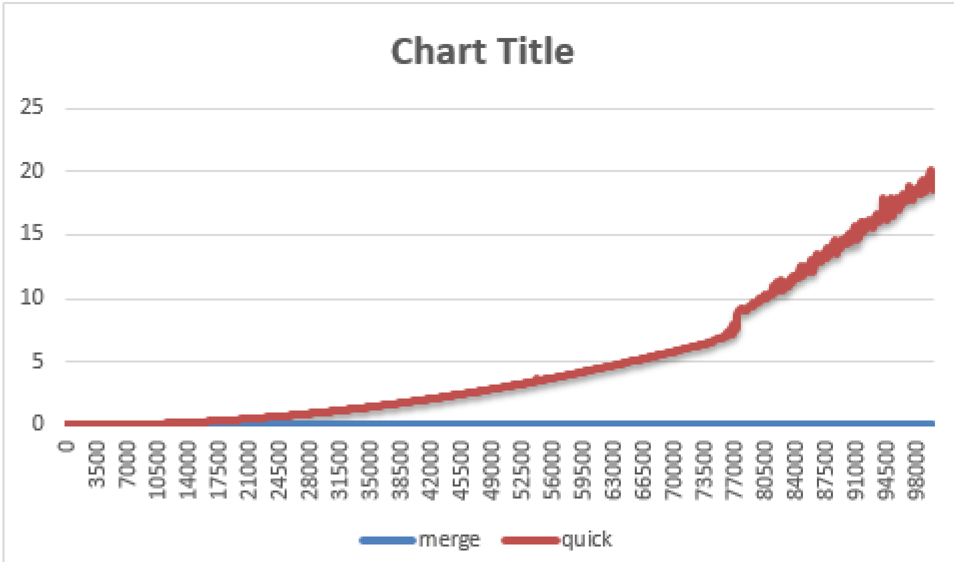
```
void operation()
{
  FILE *ptr;
  ptr=fopen("number.txt","r");
  for(int j=0;j<100000;j+=100)
  {
      int arr1[j];
      int arr2[j];
      for(int i=0;i<j;i++)
      {
          fscanf(ptr,"%d\n",&arr1[i]);
      }
      for(int i=0;i<j;i++)
      {
         arr2[i]=arr1[i];
      }
      clock_t s=clock();
      mergeSort(arr1, 0,j);
      double currm=(double)(clock()-s)/CLOCKS_PER_SEC;


      clock_t i=clock();
      quicksort(arr2, 0, j);
      double currq=(double)(clock()-i)/CLOCKS_PER_SEC;
      printf("\n%d %f %f",j,currm,currq);
  }
}
int main()
{

   gen_arr();
   operation();
   return 0;
}
```

Output:

Graph:



Observation:
In above graph, It is seen that Merge sort is more efficient than Quick sort. 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. Time complexities:

Merge sort: O(n log n)
Quick sort O (n^2).

If input size is less, then quick sort can be efficient because, It is in place sorting algorithm, It does not take extra space while sorting.

Conclusion:

In this practical, I learnt that merge sort is more efficient than Quick sort, in the long run, Merge sort takes very less time than quick sort, and merge sort also takes lesser comparisons than quick sort. I draw the graph for time taken by both algorithm for the numbers 0 to 1 lakh random numbers with 1000 blocks. Ans from the graph it is also seen that merge sort takes low time than quick sort.