Name Khan Adnan Tanweer Alam
UID No. 2022301008
Class & Division COMPS A (BATCH C)
Experiment No. 2

Aim: Experiment on finding the running time of an algorithm(merge sort and quick sort)

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.

One thing that you might wonder is what is the specialty of this algorithm. We already have a number of sorting algorithms then why do we need this algorithm? One of the main advantages of merge sort is that it
has a time complexity of O(n log n), which means it can sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort.

Quick sort

Like Merge Sort, Quick sort 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.

The key process in quickSort is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Algorithm:

Merge sort:
1. If the array “b” has only one element, return the array as it is already sorted. 2. Calculate the middle index of the array “b” using "mid = (beg + end) / 2". 3. Call the "mergesort" function recursively for the first half of the array "a[beg, mid]". 4. Call the "mergesort" function recursively for the second half of the array "a[mid+1, end]" 5. Call the "merge" function to merge the two sorted arrays obtained from the previous steps back intothe original array “b”. 6. The "merge" function takes in two arrays, the first half and the second half, and sorts the elements inboth arrays and stores them back into the original array “b”. 7. Repeat the above steps until all elements of the array “b” are sorted in ascending order.

Quick sort:
1. If the array “b” has zero or one element, return the array as it is already sorted. 2. Choose the first element of the array “b” as the pivot. 3. Initialize two variables “low” and “high” to keep track of the elements to be swapped. Set “low” to the first position and “high” to the last position in the array “b.” 4. While “low” is less than “high,” repeat the following steps: a. Increment “low” while the element at “low” is less than or equal to the pivot. b. Decrement “high” while the element at “high” is greater than the pivot. c. If “low” is less than “high”, swap the elements at “low” and “high”. 5. Swap the pivot with the element at “high” to place the pivot in its correct position in the sorted array. 6. Call the quick sort algorithm recursively for the two sub-arrays "b[beg, high-1]" and "b[high+1, end]". 7. Repeat the above steps until all elements of the array “b” are sorted in ascending order

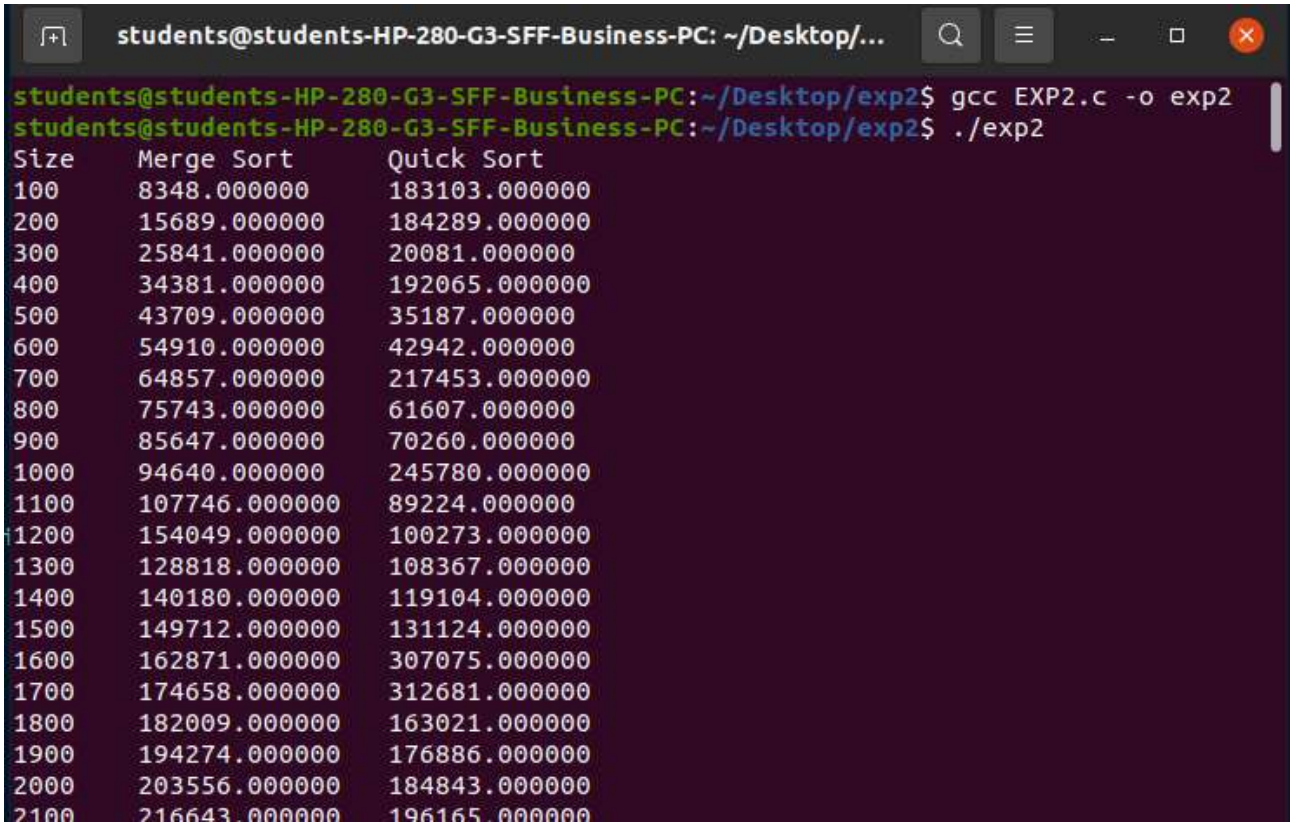
CODE :-

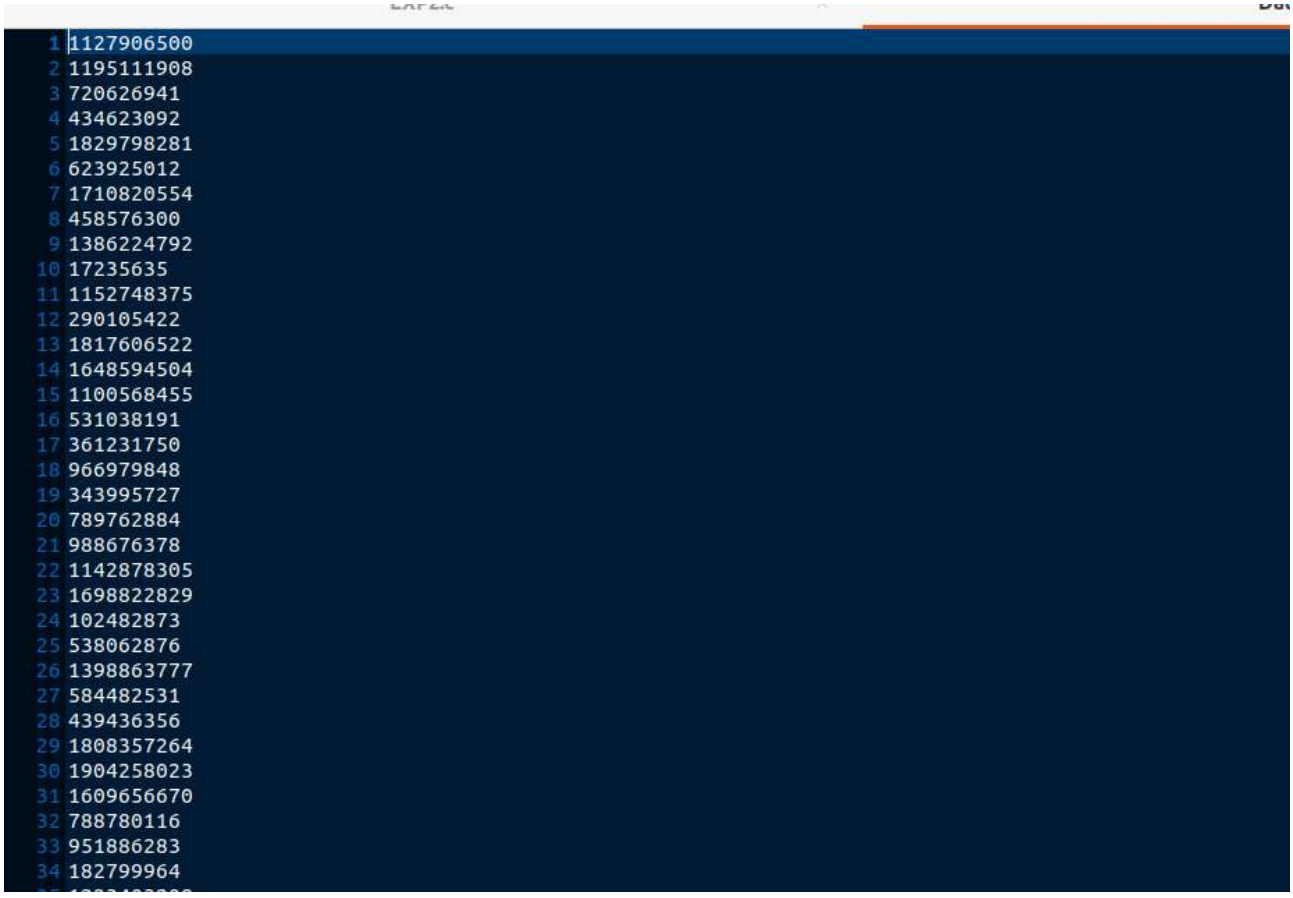
```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>

void dataInput() {
   //generate 100000 random numbers
   srand(time(NULL));
   for (int i=0;i<100000; i++)
   {
      int temp = rand();
      FILE *fptr;
      fptr = fopen("DataSet.txt", "a");
      fprintf(fptr, "%d\n", temp);
      fclose(fptr);
   }
}
//swap function
void swap(long *xp, long *yp) {
   long temp = *xp;
   *xp = *yp;
   *yp = temp;
}

//merge sort algorithm
void merge(long arr[],long temp[],int mid,int left,int right) {
   int i,left_end,size,temp_pos;
   left_end = mid-1;
   temp_pos = left;
   size = right-left+1;
   while((left<=left_end)&&(mid<=right)) {
      if(arr[left]<=arr[mid]) {
         temp[temp_pos] = arr[left];
         temp_pos = temp_pos+1;
         left = left+1;
      }
      else {
         temp[temp_pos] = arr[mid];
         temp_pos = temp_pos+1;
         mid = mid+1;
      }
   }
   while(left<=left_end) {
      temp[temp_pos] = arr[left];
      left = left+1;
      temp_pos = temp_pos+1;
   }
   while(mid<=right) {
```

```
        temp[temp_pos] = arr[mid];
        mid = mid+1;
        temp_pos = temp_pos+1;
    }
    for(i=0;i<=size;i++) {
        arr[right] = temp[right];
        right = right-1;
    }
}
void mergeSort(long arr[],long temp[],int left,int right) {
    int mid;
    if(right>left) {
        mid = (right+left)/2;
        mergeSort(arr,temp,left,mid);
        mergeSort(arr,temp,mid+1,right);
        merge(arr,temp,mid+1,left,right);
    }
}

//quick sort algorithm
int partition(long arr[], int low, int high) {
    int left, right, pivot_item = arr[low];
    left = low;
    right = high;
    while(left<right) {
        while(arr[left]<=pivot_item) {
            left++;
        }
        while(arr[right]>pivot_item) {
            right--;
        }
        if(left<right) {
            swap(&arr[left], &arr[right]);
        }
    }
    arr[low] = arr[right];
    arr[right] = pivot_item;
    return right;
}
void quickSort(long arr[], int low, int high) {
    int pivot;
    if (low<high) {
        pivot = partition(arr, low, high);
        quickSort(arr, low, pivot-1);
        quickSort(arr, pivot+1, high);
    }
}
int main(int argc, char const *argv[])
{
    //gen data
    dataInput();
    //read data from file
```

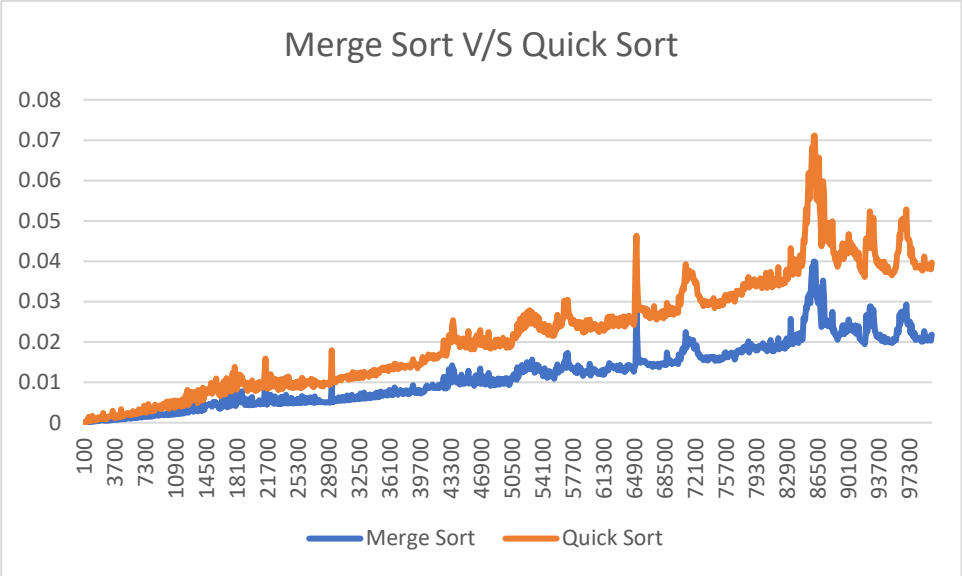
```
    FILE *fptr;
    fptr = fopen("DataSet.txt", "r");
    long arr[100000], arr1[100000], arr2[100000];
    for (int i = 0; i < 100000; i++)
    {
        fscanf(fptr, "%8ld", &arr[i]);
    }
    fclose(fptr);
    int s = 100;
    printf("Size\tMerge Sort\tQuick Sort\n");
    for(int i=0;i<1000;i++)
    {
        for(int j=0;j<s;j++)
        {
            arr1[j] = arr[j];
            arr2[j] = arr[j];
        }
        double diff1, diff2;
        struct timespec start, end;

        //merge sort
        clock_gettime(CLOCK_MONOTONIC, &start);
        long temp[s];
        mergeSort(arr1, temp, 0, s-1);
        clock_gettime(CLOCK_MONOTONIC, &end);
        diff1 = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec);

        //quick sort
        clock_gettime(CLOCK_MONOTONIC, &start);
        quickSort(arr2, 0, s-1);
        clock_gettime(CLOCK_MONOTONIC, &end);
        diff2 = (end.tv_sec - start.tv_sec) + (end.tv_nsec - start.tv_nsec);
        printf("%d\t%f\t%f\n", s, diff1, diff2);
        s += 100;
    }
    return 0;
}
```

Output:


students@students-HP-280-G3-SFF-Business-PC: ~/Desktop/...
students@students-HP-280-G3-SFF-Business-PC:~/Desktop/exp2$ gcc EXP2.c -o exp2
students@students-HP-280-G3-SFF-Business-PC:~/Desktop/exp2$ ./exp2
Size Merge Sort Quick Sort
100 8348.000000 183103.000000
200 15689.000000 184289.000000
300 25841.000000 20081.000000
400 34381.000000 192065.000000
500 43709.000000 35187.000000
600 54910.000000 42942.000000
700 64857.000000 217453.000000
800 75743.000000 61607.000000
900 85647.000000 70260.000000
1000 94640.000000 245780.000000
1100 107746.000000 89224.000000
1200 154049.000000 100273.000000
1300 128818.000000 108367.000000
1400 140180.000000 119104.000000
1500 149712.000000 131124.000000
1600 162871.000000 307075.000000
1700 174658.000000 312681.000000
1800 182009.000000 163021.000000
1900 194274.000000 176886.000000
2000 203556.000000 184843.000000


1127906500
1195111908
720626941
434623092
1829798281
623925012
1710820554
458576300
1386224792
17235635
1152748375
290105422
1817606522
1648594504
1100568455
531038191
361231750
966979848
343995727
789762884
988676378
1142878305
1698822829
102482873
538062876
1398863777
584482531
439436356
1808357264
1904258023
1609656670
788780116
951886283
182799964

Graph:



Observation: For the initial lower input numbers, both merge and quick sort provide results in almost similar runtimes. Quick sort is a tad bit faster than merge sort for higher number of inputs.

Conclusion: Quick sort takes less runtime and hence is a little more feasible than Merge sort for higher number of inputs.