DAA Experiment-2
(Batch-A/A1)

Name	Ansari Mohammed Shanouf Valijan
UID Number	2021300004
Class	SY B.Tech Computer Engineering(Div-A)
Experiment Number	2
Date of Performance	09-02-23
Date of Submission	15-02-23

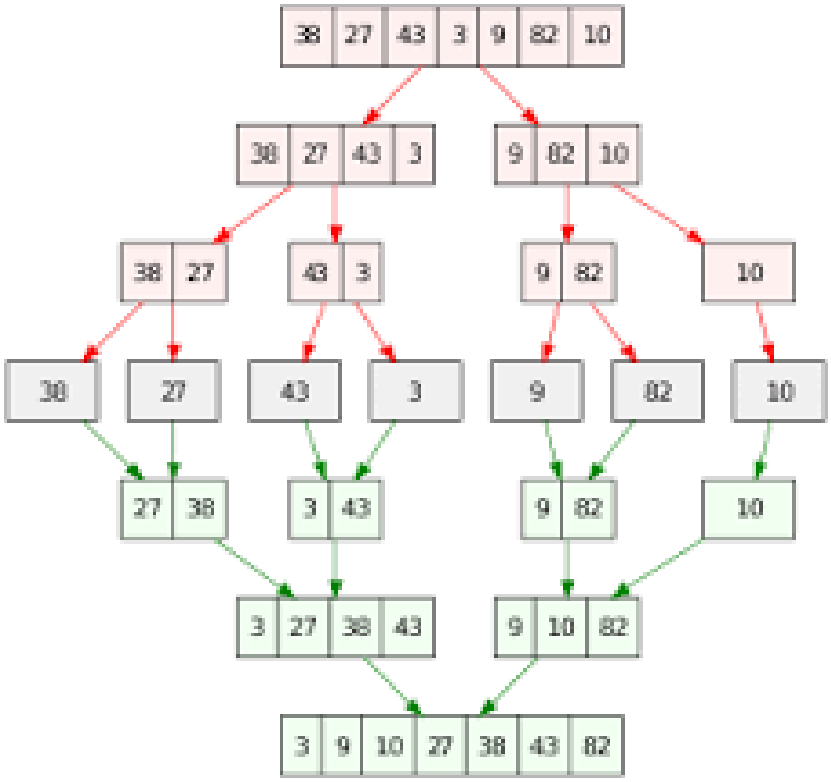
Aim:
To compare two sorting algorithms- Merge sort and Quick sort, based on their running time.

Theory:
Merge sort and quick sort are yet another crucial sorting algorithms that are comparatively much faster than the insertion or selection sort. These algorithms are based on the divide and conquer approach of solving a problem. In other words, recursion plays an important role in their implementation. Below is a brief description of these two sorting algorithms-

Merge Sort-

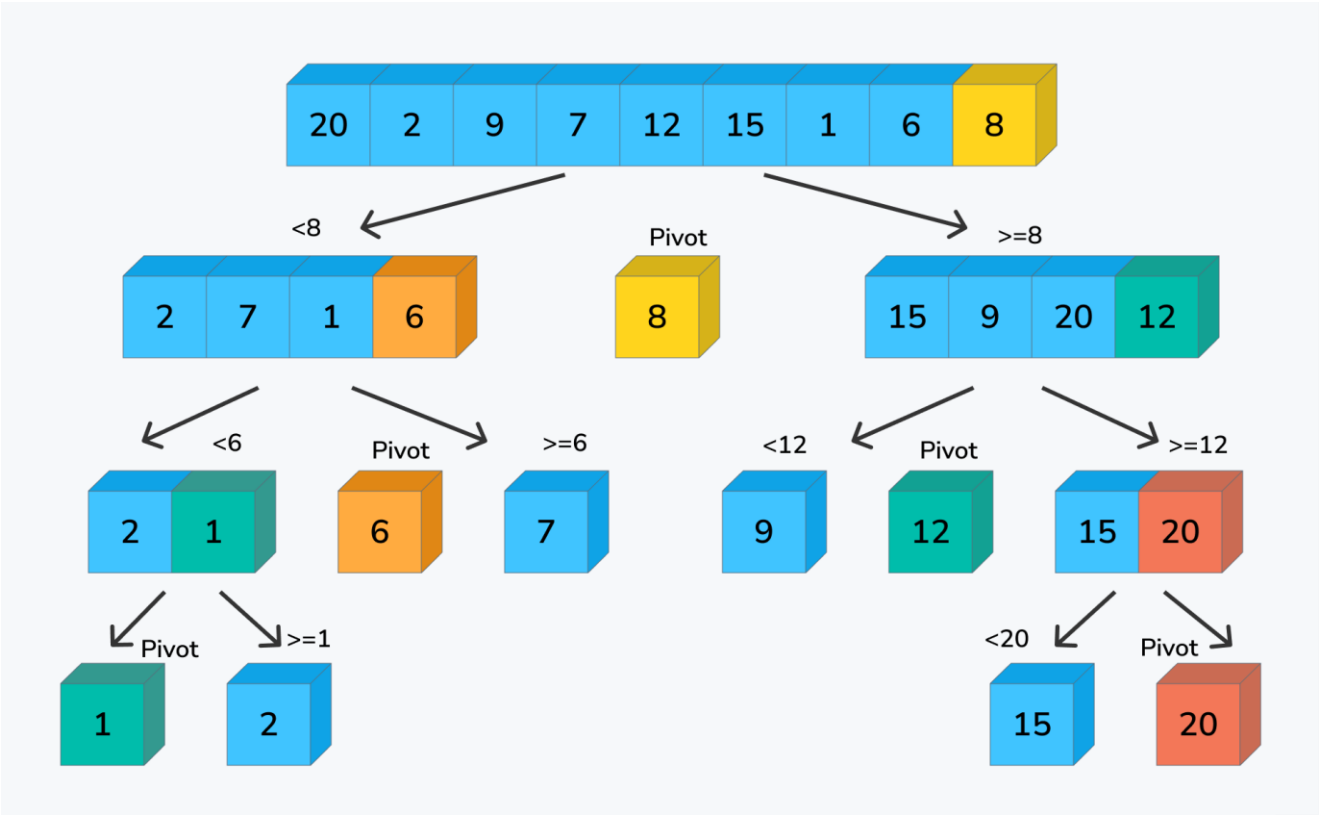
As mentioned earlier, it works on the divide and conquer approach. In this type of sorting, the array that is to be sorted is broken into smaller and smaller group of elements virtually till each group consists of a single element. Further, these groups are merged in such a fashion that each resulting group is a sorted group of elements. Doing this recursively ends up in sorting the entire array.

Below is an image demonstrating, conceptually, the above mentioned flow of the algorithm. Each arrow indicates a recursive call to the merge sort function. The first half of the image depicts the breaking of array in question into smaller and smaller groups while the second half shows the merging process to get the sorted array. The merging process requires an additional array so that the elements can be properly arranged.



Quick Sort-

In quick sort, the elements are placed at their respective places based on recursive partition. This partition happens around a reference element known as pivot. Considering the sorting being done for obtaining the ascending order of elements, the ones lesser than the pivot are placed to its left, while the others are placed to its right. For partitioning an array around a pivot, usually two pointers are used. Below is an image which depicts the process of quick sort-



Any element can be chosen to be the pivot. However, usually first or last element is selected to act as the pivot. When the process of partitioning the array is completed, we get the left unsorted array and the right unsorted array. These are also sorted recursively using the logic of quick sort.

Algorithm:

[A] For merge sort-

I. Calculate the middle index of the array that is to be sorted.
II. Call the mergeSort function for the two groups of elements formed in the concerned array.
III. Create a temporary array to store the elements of the two groups formed in order to sort them.
IV. Using two pointers, one on the beginning of each group, start traversing the groups and comparing the elements pointed.
V. Pick the smallest of the two elements, move it to the main array, and increment the concerned pointer by 1.
VI. Repeat till both the groups in the temporary array are empty.

[B] For quick sort-

I. Fix the pivot, the low and high pointers on the array.
II. Increment the low pointer till an element greater than the pivot is found, stop if the pointer reaches end of array.
III. Decrement the high pointer till an element lesser than the pivot is found, stop if the pointer reaches beginning of array.
IV. If high pointer is ahead of the low pointer, swap the elements pointed by them.
V. Repeat steps 2 to 4 till high is ahead of low pointer.
VI. Swap the pivot and the element pointed by high.
VII. Using the partitioning index represented by high pointer, call the quickSort function for the left and right partition.

Program:

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

//file for handling the input and output
FILE* data=NULL;
```

```c
FILE* output=NULL;

//functions for sorting
void swap(int arr[], int ix, int iy){
    int temp=arr[ix];
    arr[ix]=arr[iy];
    arr[iy]=temp;
}

//for merge-sort
void mergeSort(int arr[], int start, int end){
    if(start<end){
        int mid, ptr1, ptr2, tempInd;
        mid=(start+end)/2;
        mergeSort(arr, start, mid);
        mergeSort(arr, mid+1, end);
        int* tempArr=(int*)malloc((end+1)*sizeof(int));
        for(int i=start; i<=end; i++)
            tempArr[i]=arr[i];
        ptr1=start;
        ptr2=mid+1;
        tempInd=start;
        while(ptr1<=mid&&ptr2<=end){
            if(tempArr[ptr1]>=tempArr[ptr2]){
                arr[tempInd]=tempArr[ptr2];
                ptr2++;
                tempInd++;
            }
            else if(tempArr[ptr1]<tempArr[ptr2]){
                arr[tempInd]=tempArr[ptr1];
                ptr1++;
                tempInd++;
            }
        }
        while(ptr1<=mid){
            arr[tempInd]=tempArr[ptr1];
            ptr1++;
            tempInd++;
        }
        while(ptr2<=end){
            arr[tempInd]=tempArr[ptr2];
            ptr2++;
            tempInd++;
        }
        free(tempArr);
    }
}

//for quick-sort
int partition(int arr[], int low, int high){
    //considering first element as the pivot
    int pivot=arr[low], i=low+1, j=high;
    do{
        while(i<=high&&arr[i]<=pivot)
```

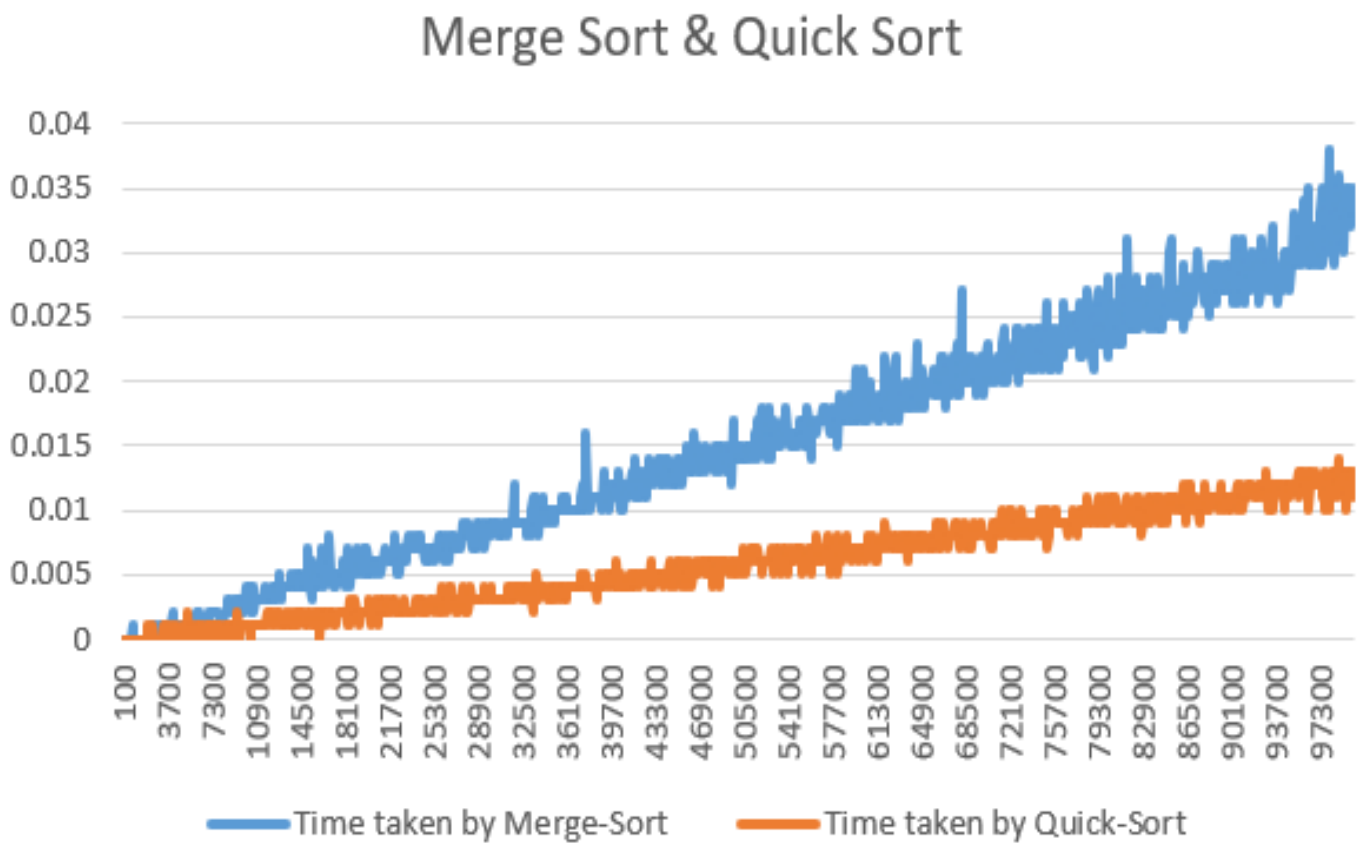
```
            i++;
        while(j>=low&&arr[j]>pivot)
            j--;
        if(i<j)
            swap(arr,i,j);
    }while(i<j);
    if(low!=j)
        swap(arr,low,j);
    return j;
}

void quickSort(int arr[], int low, int high){
    if(low<high){
        int partIndex=partition(arr,low,high);
        quickSort(arr,low,partIndex-1);
        quickSort(arr,partIndex+1,high);
    }
}

//for bringing the values into the array(for sorting)
void arrayLoading(int arr[],int endpoint){
    data=fopen("data.txt","r");
    for(int i=0; i<endpoint; i++){
        fscanf(data,"%d",&arr[i]);
    }
    fclose(data);
}

void main(){

    srand(time(NULL));
    clock_t t;
    int tempArr[100000];

    //intiating the data file
    data=fopen("data.txt","w");
    for(int i=0; i<100000; i++){
        fprintf(data,"%d\n",rand());
    }
    fclose(data);

    //sorting and printing out the time taken
    output=fopen("output.txt","w");
    double timeI, timeS;

    for(int i=100; i<=100000; i+=100){
        arrayLoading(tempArr,i);
        t=clock();
        mergeSort(tempArr,0,i-1);
        t=clock()-t;
        timeI=((double)t)/CLOCKS_PER_SEC;

        arrayLoading(tempArr,i);
        t=clock();
```

```
        quickSort(tempArr,0,i-1);
        t=clock()-t;
        timeS=((double)t)/CLOCKS_PER_SEC;
        fprintf(output,"%d\t\t%lf\t\t%lf\n",i,timeI,timeS);
        printf("Block count -----> %d\n",i);
    }
    fclose(output);
}
```

Implementation:

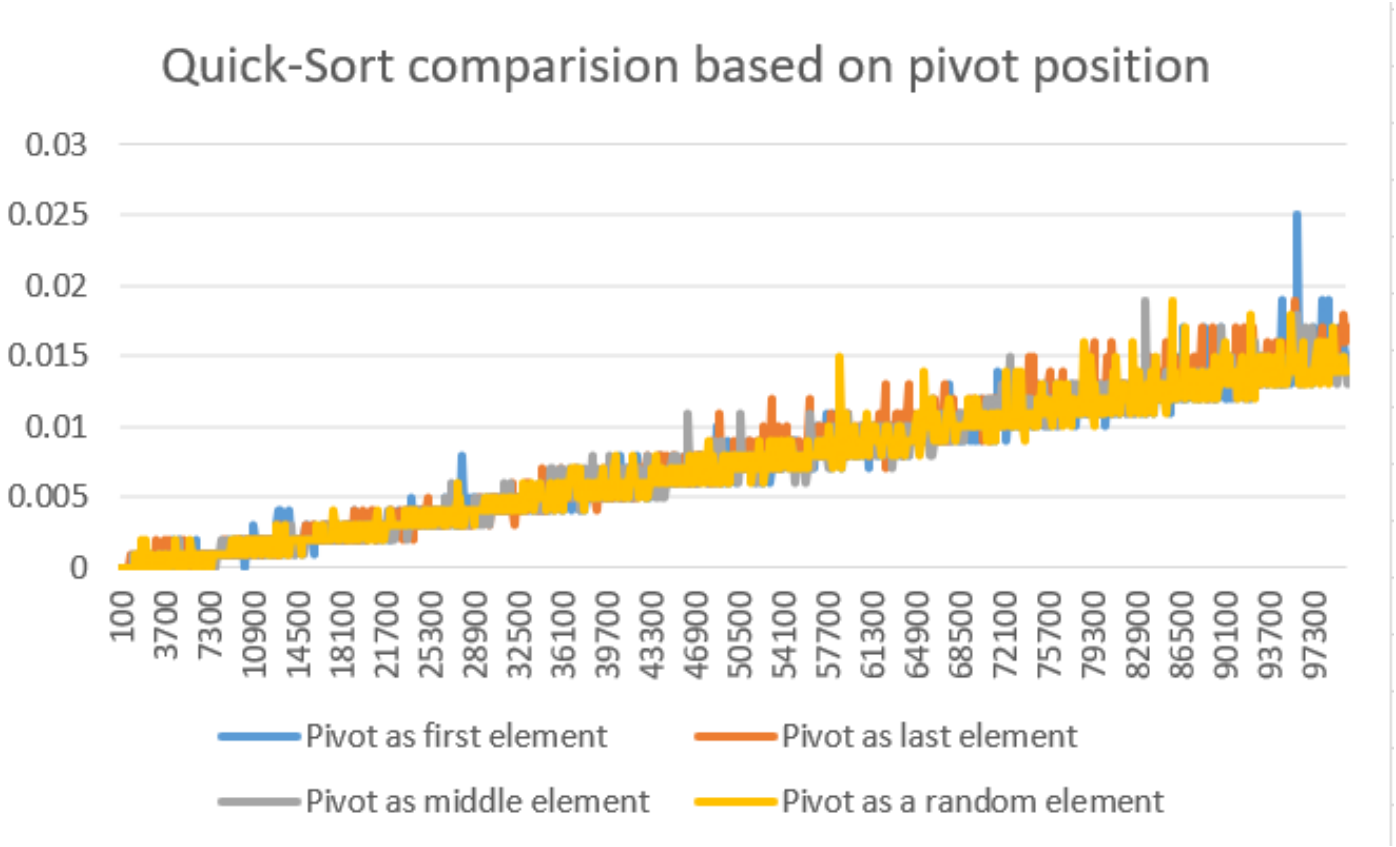
After running the above-mentioned algorithms, a file of data regarding the time taken by these sorting algorithms to sort a given number of elements was obtained. This was then copied to MS Excel for obtaining a graph.

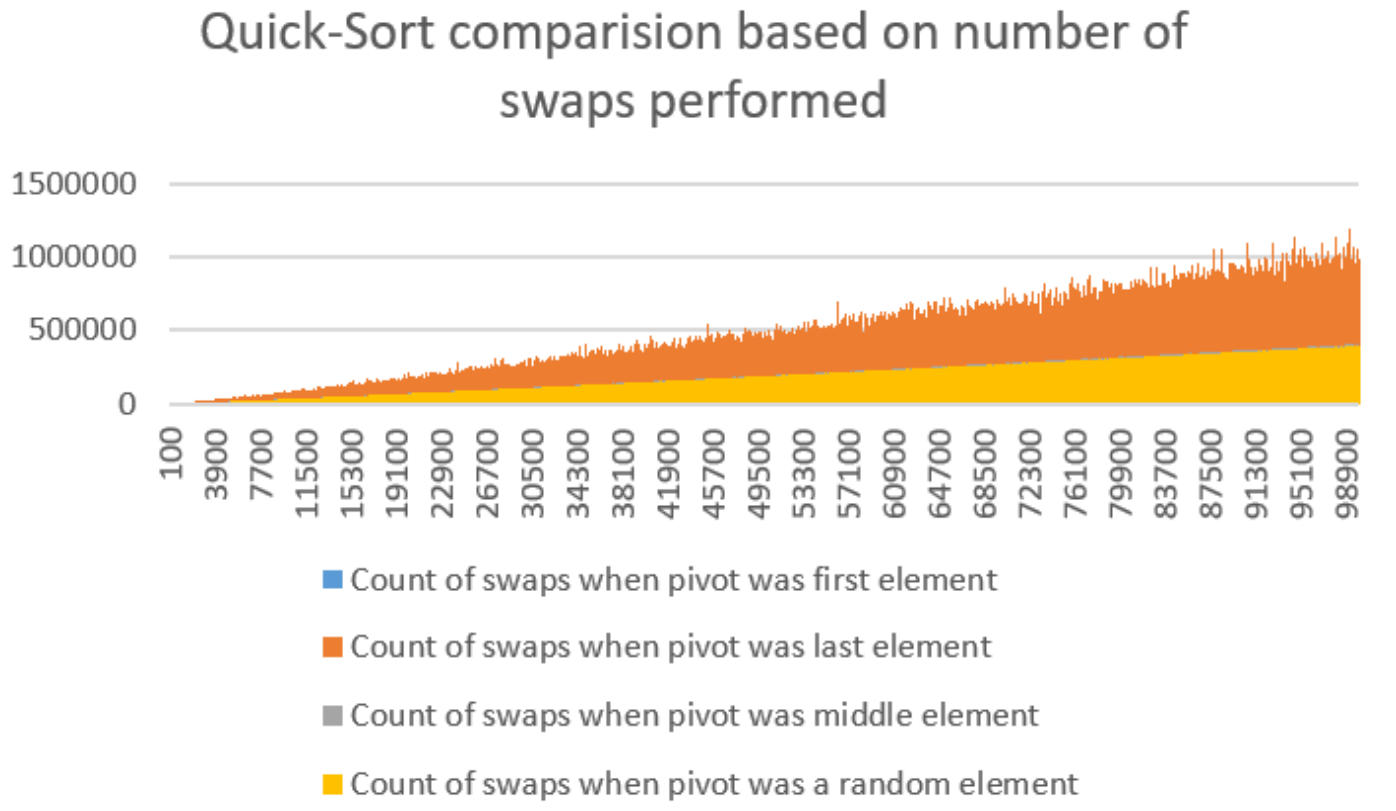
The following plot was obtained-



Apart from comparing Merge sort and Quick sort, four variations of quick sort depending on the position of the pivot were also compared based on the time required for sorting and the total number of swaps performed for the same.

The following plots were obtained-





Inference:

From the first graph obtained, we notice that quick sort is a faster sorting algorithm as compared to merge sort. The time taken by quick sort algorithm to sort 100000 elements was recorded to be about 0.01 to 0.015 seconds. On the other hand, merge sort took about 0.03 seconds for the same. This explains the

difference in efficiency of these algorithms. We notice that initially, when the elements are less in number, both the sorting algorithms work equally well; however, upon the increase of elements further, these two deviate in terms of time taken for the process. Nevertheless, both of these algorithms are very efficient and take much less time than insertion or selection sort algorithms.

From the second plot, we notice that the time required for sorting the elements remains almost independent of the position of pivot. The line graph of the time taken for sorting when pivot was considered as a random element seems to represent the average time taken for general quick-sort procedure for respective number of elements. From the final plot, we observe that number of swaps required in the quick-sort procedure is much more in case where pivot is the last element as compared to other considered cases. One thing to understand is that this observation would change depending on the data that is to be sorted, but a general remark can be made on the observed results.

Conclusion:

By performing this experiment, I was able to understand the implementation of merge and quick sort algorithms. I was also able to do a comparative study on these and was able to realize that these algorithms are much faster than the ones that were previously studied.