Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to University of Mumbai)
Munshi Nagar, Andheri (W), Mumbai – 400 058.

Experiment No. 2

Name: Om Bhutki

UID: 2022301003

Subject: DAA

Experiment No: 2

Aim – Experiment based on divide and conquers approach.

Objective: To understand the running time of algorithms by implementing two sorting algorithms based on divide and conquers approach namely Merge and Quick sort.

Theory:

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.

Quick sort– Like Merge Sort, QuickSort 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.

It is mainly divided in three basic steps:

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Algorithm:

step 1: start

Step2: call rand_num() function

Step 2: create rand_num file and store the random numbers in it.

Step3: open rand_num file in reading mode

Step 4: Store all random numbers in an array

Step5: Traverse all elements using for loop take n as 100

Step6: Perform insertion and selection sort on each block of 100 numbers

Step7: Calculate time required to perform insertion and selection sort at each iteration

Step8: Increment n by 100

Step 9: If n reaches 1000 then end else go to step 6

rand_num() 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

Merge sort:

step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.

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

Quick Sort:

Algorithm for choosing pivot:

Step 1 − Choose the highest index value as pivot

Step 2 − Take two variables to point left and right of the list excluding pivot

Step 3 − left points to the low index

Step 4 − right points to the high

Step 5 − while value at left is less than pivot move right

Step 6 − while value at right is greater than pivot move left

Step 7 − if both step 5 and step 6 does not match swap left and right

Step 8 − if left ≥ right, the point where they met is new pivot

Algorithm for quick sort:

Step 1 − Make the right-most index value pivot

Step 2 − partition the array using pivot value

Step 3 − quicksort left partition recursively

Step 4 − quicksort right partition recursively

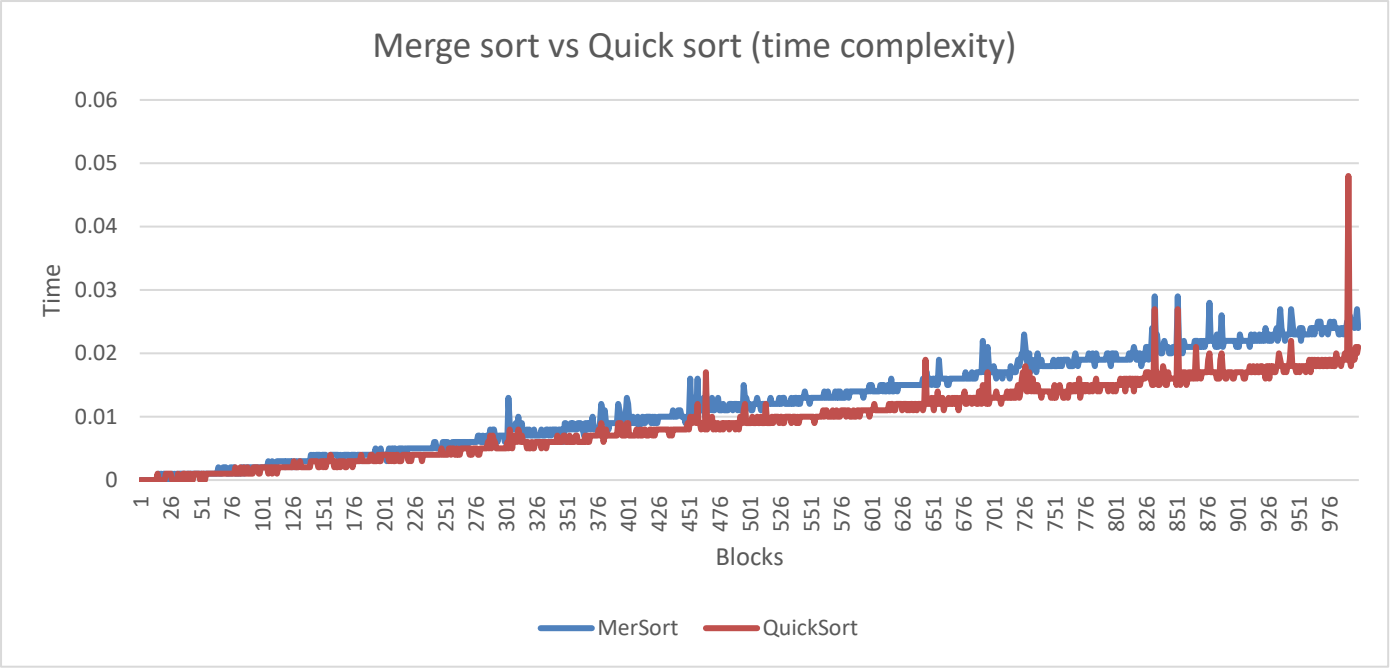
Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
long SWAP = 0;

void merge(int arr[], int p, int q, int r)
{
   int i, j, k;
   int n1 = q - p + 1;
   int n2 = r - q;

   int L[n1], R[n2];

   for (i = 0; i < n1; i++)
      L[i] = arr[p + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[q + 1 + j];

   i = 0;
   j = 0;

   k = p;
   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);
   }
}

int quicksort(int a[], int start, int end)
{
   int pivot = a[end];
//int pivot = a[start];
//int random = start + rand() % (end - start);
//int pivot = a[random];
//int mid = start + (end - start)/2;
//int pivot = a[mid];
   int i = (start - 1);

   for (int j = start; j <= end - 1; j++)
   {
      if (a[j] < pivot)
      {
         i++;
         int t = a[i];
         a[i] = a[j];
         a[j] = t;
         SWAP++;
      }
   }
   int t = a[i + 1];
   a[i + 1] = a[end];
   a[end] = t;
   SWAP++;
   return (i + 1);
}

double quick(int a[], int start, int end)
{
   if (start < end)
   {
      int p = quicksort(a, start, end);
      quick(a, start, p - 1);
      quick(a, p + 1, end);
   }
}

int main()
{
   double qust,mest;
   srand(time(0));
```

```
    FILE *fp,*file;
    fp = fopen("random.txt", "w");
    for (int i = 0; i < 100000; i++)
    {
        fprintf(fp, "%d\n", rand() % 900001 + 100000);
    }
    int upper_limit = 100;
    fclose(fp);
    file = fopen("output.txt","w");
    fprintf(file,"Block\tMerSort\tQuickSort\tSwaps\n");
    for (int i = 0; i < 1000; i++)
    {
        fp = fopen("random.txt", "r");
        int arr1[upper_limit], arr2[upper_limit], temp_num;
        for (int j = 0; j < upper_limit; j++)
        {
            fscanf(fp, "%d", &temp_num);
            arr1[j] = temp_num;
            arr2[j] = temp_num;
        }
        fclose(fp);

        clock_t t;
        t = clock();
        mergeSort(arr2,0,upper_limit-1);
        t = clock() - t;
        mest = ((double)t) / CLOCKS_PER_SEC;
        clock_t t1;
        t1 = clock();
        qust=quick(arr1,0,upper_limit-1);
        t1 = clock() - t1;
        qust = ((double)t1) / CLOCKS_PER_SEC;
        fprintf(file,"%d\t%lf\t%lf\t%ld\n",i+1, mest, qust,SWAP);
        fflush(stdout);
        upper_limit += 100;
    }
    return 0;
}
```

Result:



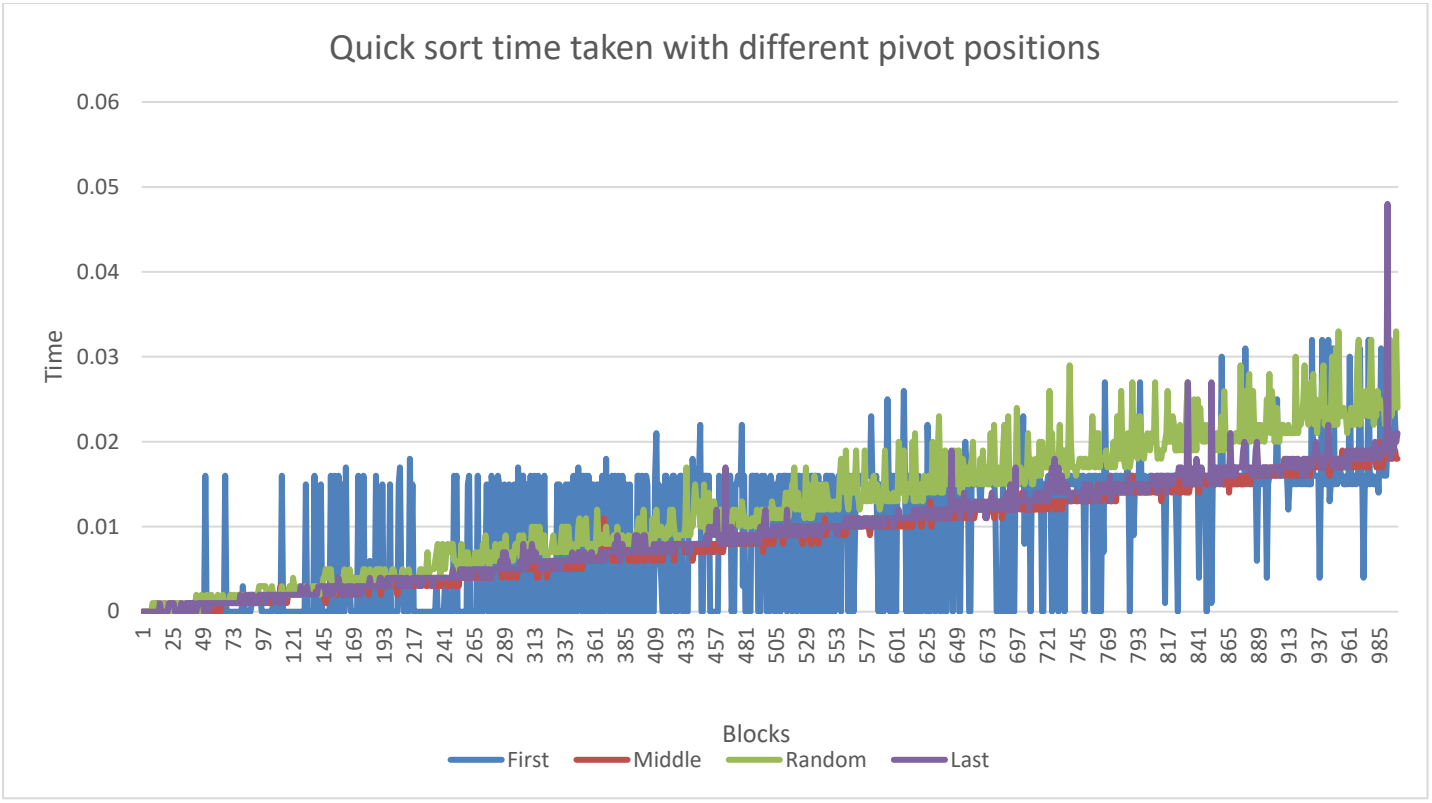
Inference:

Merge sort: As we can see from the graph, merge sort takes more time at each iteration than quick sort as its always about quick sort in the graph. There are some spikes in at certain movements but overall we can observe a linear increase in the graph.

Quick sort: On the other hand, time taken to perform quick sort is just slightly less than merge sort but on the other hand there are a lot of spikes that can be observed where one rises to as high as 0.05 seconds. It is expected that with some proper work some of these spikes can be overcome. Also, compare with merge sort, which achieves similar results but has much lower round-trip-time.

Also, these values differ from system to system depending upon hardware configuration, operating system, memory, cpu speed etc.

Overall both take about the same time with negligible difference but on my system, quick sort performs slightly better.



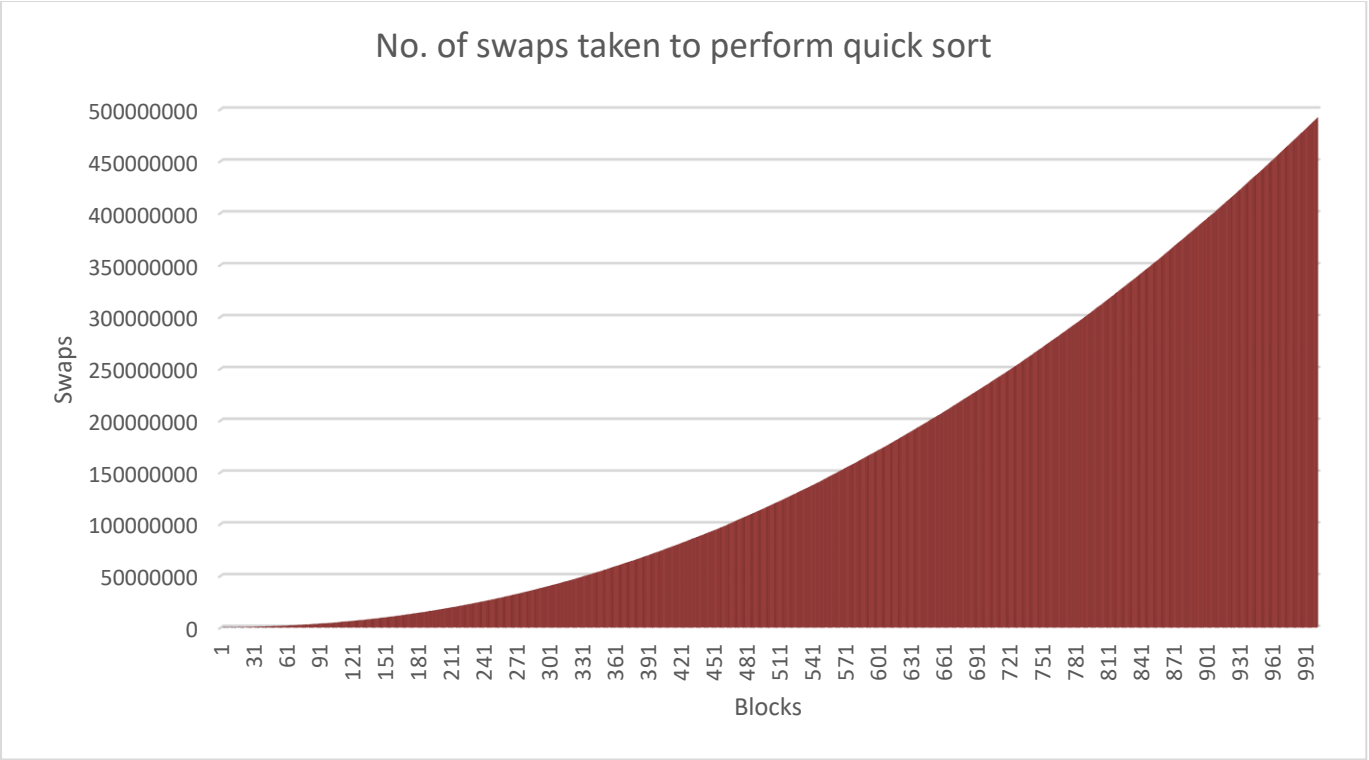
Inference:

When we take pivot as the first element there are a lot of variations in the graph. It takes around zero seconds to perform the sorting at the start but suddenly takes about 0.02 seconds and after suddenly goes back to zero. It continues this behavior till the end and we don't see any linear increase like the other graphs.

When we take pivot as the middle element there is a linear increase in the graph and not many spikes. It behaves normally and we don't see many variations in the graph as compared to when we take pivot as first.

Similarly when we take pivot as the last element we see a linear increase in the graph like when we take it as the middle element but we can observe more spikes in time here than middle. Besides that, it takes around the same time as middle.

When we take pivot as a random element it takes on average the most time than all 3 methods and the spikes are a lot more prevalent. We can see a lot of up and down motion but not as much as when we take pivot as first element.



Inference: As we observe as we add more numbers at each iteration to the block the number of swaps required increase significantly reaching hundreds of millions. Linear increase is observed.

Conclusion:

Thus, after performing this experiment I conclude that quick sort performs better than merge sort but it also unstable whereas merge sort takes a little more time but it has a linear increase and doesn't have a lot of ups and downs. Moreover, when performing quick sort on large data the position of the pivot matter a lot as just changing the pivot yields drastic results. Also, no. of swaps required to sort data is directly proportional to the size of the data.