Laboratory 10

1. Questions

   Write a C program to implement sorting of numbers using bubble sort, selection sort and quick sort techniques. Calculate the time required for each approach.

2. Algorithm

```
Step1: start

Step2: for bubble sort
```

   2.1 Starting with the first element (index = 0), compare the current element with the next element of the array.

   2.2 If the current element is greater than the next element of the array, swap them.

   2.3 If the current element is less than the next element, move to the next element. Repeat Step 1.

```
Step3: for selection sort
```

   3.1 Starting from the first element, we search the smallest element in the array, and replace it with the element in the first position.

   3.2 We then move on to the second position, and look for smallest element present in the subarray, starting from index 1, till the last index.

   3.3 We replace the element at the second position in the original array, or we can say at the first position in the subarray, with the second smallest element.

   3.4 This is repeated, until the array is completely sorted.

Step4: for quick sort

4.1 After selecting an element as pivot, which is the
last index of the array in our case, we divide the array
for the first time.

4.2 In quick sort, we call this partitioning. It is not
simple breaking down of array into 2 subarrays, but in
case of partitioning, the array elements are so
positioned that all the elements smaller than
the pivot will be on the left side of the pivot and all
the elements greater than the pivot will be on the right
side of it.

4.3 And the pivot element will be at its
final sorted position.

4.4 The elements to the left and right, may not be
sorted.

4.5 Then we pick subarrays, elements on the left
of pivot and elements on the right of pivot, and we
perform partitioning on them by choosing a pivot in the
subarrays.

Step5: stop

3. Program

```c
// sorts
#include <stdio.h>
#include <time.h>

#define max 10000

// for bubble sort
void bubbleSort(int array[], int size)
{
  for (int step = 0; step < size - 1; ++step)
  {
    for (int i = 0; i < size - step - 1; ++i)
    {
      if (array[i] > array[i + 1])
      {
        int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;
      }
    }
  }
}

void swap(int *a, int *b)
{
  int temp = *a;
  *a = *b;
  *b = temp;
}
// for selection sort
void selectionSort(int array[], int size)
{
  for (int step = 0; step < size - 1; step++)
  {
    int min_idx = step;
    for (int i = step + 1; i < size; i++)
    {
      if (array[i] < array[min_idx])
        min_idx = i;
    }
    swap(&array[min_idx], &array[step]);
  }
}
// for quick sort
int partition(int array[], int low, int high)
{
  int pivot = array[high];
  int i = (low - 1);
  for (int j = low; j < high; j++)
  {
    if (array[j] <= pivot)
    {
      i++;
      swap(&array[i], &array[j]);
    }
  }
  swap(&array[i + 1], &array[high]);
  return (i + 1);
}
void quickSort(int array[], int low, int high)
{
  if (low < high)
  {
    int pi = partition(array, low, high);
    quickSort(array, low, pi - 1);
    quickSort(array, pi + 1, high);
  }
}
// for printing
void printArray(int array[], int size)
{
  for (int i = 0; i < size; ++i)
  {
    printf("%d  ", array[i]);
  }
  printf("\n");
}
int data[max];
void createdata(){
  // we are getting time for worst-case
  for(size_t i = max; i < -1; i--)
  {
    data[i] = i;
  }
}

int main()
{
  int size = sizeof(data) / sizeof(data[0]);
  clock_t start_t, end_t;

  createdata();
  start_t = clock();
  bubbleSort(data, max);
  end_t = clock();
  printf("\nExecution time for Bubble-sort : %f seconds", (double)(end_t - start_t) / CLOCKS_PER_SEC);

  createdata();
  start_t = clock();
  selectionSort(data, max);
  end_t = clock();
  printf("\nExecution time for Selection-sort : %f seconds", (double)(end_t - start_t) / CLOCKS_PER_SEC);

  createdata();
  start_t = clock();
  quickSort(data, 0, max - 1);
  end_t = clock();
  printf("\nExecution time for Quick-sort : %f seconds", (double)(end_t - start_t) / CLOCKS_PER_SEC);
}
```

Figure 1 program for different sorting techniques with their execution time.

4. Presentation of Results

Fig 2 shows the output is execution time for worst-case when sorting from 10,000 to 1 in ascending order.

```
PS D:\RUAS-sem-03\DSA lab\programs> cd "d:\RUAS-sem-03\DSA lab\programs\" ;

Execution time for Bubble-sort : 0.078000 seconds
Execution time for Selection-sort : 0.094000 seconds
Execution time for Quick-sort : 0.218000 seconds
PS D:\RUAS-sem-03\DSA lab\programs> []
```

Figure 2 Execution time for different sorting techniques for 10,000 values in worst cases.

Fig 3 shows output for 10 random numbers given by user to check if the sorting algorithms are working properly or not.

```
PS D:\RUAS-sem-03\DSA lab\programs> cd "d:\RUAS-sem-03\DSA lab\programs\" ;
Enter array: 9 4 2 3 1 5 6 8 7 0
Bubble Sort: 0  1  2  3  4  5  6  7  8  9

Execution time for Bubble-sort : 0.000000 seconds
Selection Sort: 0  1  2  3  4  5  6  7  8  9

Execution time for Selection-sort : 0.000000 seconds
Quick Sort: 0  1  2  3  4  5  6  7  8  9

Execution time for Quick-sort : 0.000000 seconds
PS D:\RUAS-sem-03\DSA lab\programs> |
```

Figure 3 sorting technique for 10 user given inputs and their execution time

Note: As shown in Fig 3, for such a small input the execution time is 0 sec.

Actually, the **clock ()** function is not capable of showing precise time up to nano seconds.

It can also be noticed that sorting algorithms are working.

5. Conclusions

Learning happened:

Bubble sort:

- Worst Case Time Complexity [Big-O]: $O(n^2)$
- Best Case Time Complexity [Big-omega]: $O(n)$
- Average Time Complexity [Big-theta]: $O(n^2)$
- Space Complexity: $O(1)$

Selection sort:

- Worst Case Time Complexity [Big-O]: $O(n^2)$

- Best Case Time Complexity [Big-omega]: $O(n^2)$

- Average Time Complexity [Big-theta]: $O(n^2)$

- Space Complexity: $O(1)$

Quick sort:

- Worst Case Time Complexity [Big-O]: $O(n^2)$

- Best Case Time Complexity [Big-omega]: $O(n * log\ n)$

- Average Time Complexity [Big-theta]: $O(n * log\ n)$

- Space Complexity: $O(n * log\ n)$