Quick Sort

By : Neelanjan Mukherji & Amit Chaudhary

Introduction

- Quick Sort is a widely used sorting algorithm in computer science that follows the divide-and-conquer strategy.
- It efficiently sorts a given list or array of elements by repeatedly partitioning it into two subarrays based on a chosen pivot element.

Algorithm

- Choose a pivot element from the list.

35	**50**	**15**	**25**	**80**	**20**	**90**	**45**

```
if p ≥ q then:
    Stop

if p < 1 then:
    Stop
else
    Arr(p + 1) for p
    Arr(q - 1) for q
        continue
```

Algorithm(Contd.)

35	20	15	25	80	50	90	45

```
swap(p, q)
    temp <- p
    p <- q
    q <- temp
```

Algorithm(Contd.)

25	20	15	35	80	50	90	45

```
if p & q overlapped then:
    swap(q, pivot)
    continue
```

Algorithm(Contd.)

```
partition(arr, p, q):
pivot := arr[q]  // Choose the last element as the pivot
i := (p - 1)  // Index of smaller element

for j := p to q - 1 do:
    if arr[j] <= pivot then:
        i := i + 1
        swap(arr[i], arr[j])  // Swap arr[i] and arr[j]

swap(arr[i + 1], arr[q])  // Swap arr[i+1] and arr[q]
return (i + 1)
```

Algorithm(Contd.)

Sub Array 1	Pivot	Sub Array 2
25 20 15	35	80 50 90 45

```
quickSort(arr, p, q):
    if p < q then:
        pivotIndex <= partition(arr, p, q)
        quickSort(arr, p, pivotIndex - 1)
        quickSort(arr, pivotIndex + 1, q)
```

Algorithm(Contd.)

- Recursively apply the above steps to the subarrays until the entire list is sorted.

Sub Array 1	Pivot	Sub Array 2
15 20 25	35	80 50 45 90

Sub Array 1	Pivot	Sub Array 2
15 20 25	35	45 50 80 90

Algorithm(Contd.)

- Final Sorted Array

15	**20**	**25**	**35**	**45**	**50**	**80**	**90**

Time Complexity

The time complexity of Quick Sort can be analyzed in different scenarios:

- Average Case
- Best Case
- Worst Case

Average Case

- In the average case, Quick Sort demonstrates excellent performance. The partitioning step divides the array into two roughly equal subarrays, resulting in balanced recursive calls.
- The time complexity of Quick Sort in the average case is O(n log n), where 'n' represents the number of elements in the array.
- This average time complexity arises due to the repeated partitioning of the array into halves.

Best Case

- The best case occurs when the pivot selection consistently divides the array into two equal-sized subarrays.
- In the best case, the time complexity of Quick Sort is also $O(n \ \log n)$.

Worst Case

- The worst case of Quick Sort occurs when the pivot selection is consistently unfavorable, resulting in highly imbalanced partitions.
- For example, if the pivot is always the smallest or largest element in the array, each partitioning step divides the array into one subarray with $(n-1)$ elements and another with no elements.

Worst Case(Contd.)

- In the worst case, Quick Sort's time complexity is $O(n^2)$, which happens when the array is already sorted or contains many equal elements.
- However, the worst-case scenario is less likely to occur in practice, especially with randomized pivot selection techniques.

Code

```c
#include <stdio.h>

// Function to swap two elements
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}
```

Code(Contd.)

```
// Function to partition the array and return the pivot index
int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // Choose the last element as the pivot
    int i = (low - 1);  // Index of smaller element

    for (int j = low; j <= high - 1; j++) {
        // If the current element is smaller than or equal to the pivot
        if (arr[j] <= pivot) {
            i++;
            swap(&arr[i], &arr[j]);  // Swap arr[i] and arr[j]
        }
    }

    swap(&arr[i + 1], &arr[high]);  // Swap arr[i+1] and arr[high]
    return (i + 1);
}
```

Code(Contd.)

```
// Recursive function to implement Quick Sort
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Find the pivot index
        int pivotIndex = partition(arr, low, high);

        // Recursive calls to sort the subarrays
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}
```

Code(Contd.)

```c
// Function to print the array
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}
```

Code(Contd.)

```c
// Test the algorithm
int main() {
    int arr[] = {9, 4, 7, 2, 1, 5, 3};
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, size);

    quickSort(arr, 0, size - 1);

    printf("Sorted array: ");
    printArray(arr, size);

    return 0;
}
```

Output

```
Original array: 9 4 7 2 1 5 3
Sorted array: 1 2 3 4 5 7 9
```