Quick Sort Algorithm

Quick Sort is one of the different Sorting Technique which is based on the concept of **Divide and Conquer**, just like merge sort. But in quick sort all the heavy lifting(major work) is done while **dividing** the array into subarrays, while in case of merge sort, all the real work happens during **merging** the subarrays. In case of quick sort, the combine step does absolutely nothing.

It is also called **partition-exchange sort**. This algorithm divides the list into three main parts:

1. Elements less than the **Pivot** element
2. Pivot element(Central element)
3. Elements greater than the pivot element

Pivot element can be any element from the array, it can be the first element, the last element or any random element. In this tutorial, we will take the rightmost element or the last element as **pivot**.

For example: In the array {52, 37, 63, 14, 17, 8, 6, 25}, we take 25 as **pivot**. So after the first pass, the list will be changed like this.

{6 8 17 14 **25** 63 37 52}

Hence after the first pass, pivot will be set at its position, with all the elements **smaller** to it on its left and all the elements **larger** than to its right. Now 6 8 17 14 and 63 37 52 are considered as two separate sunarrays, and same recursive logic will be applied on them, and we will keep doing this until the complete array is sorted.

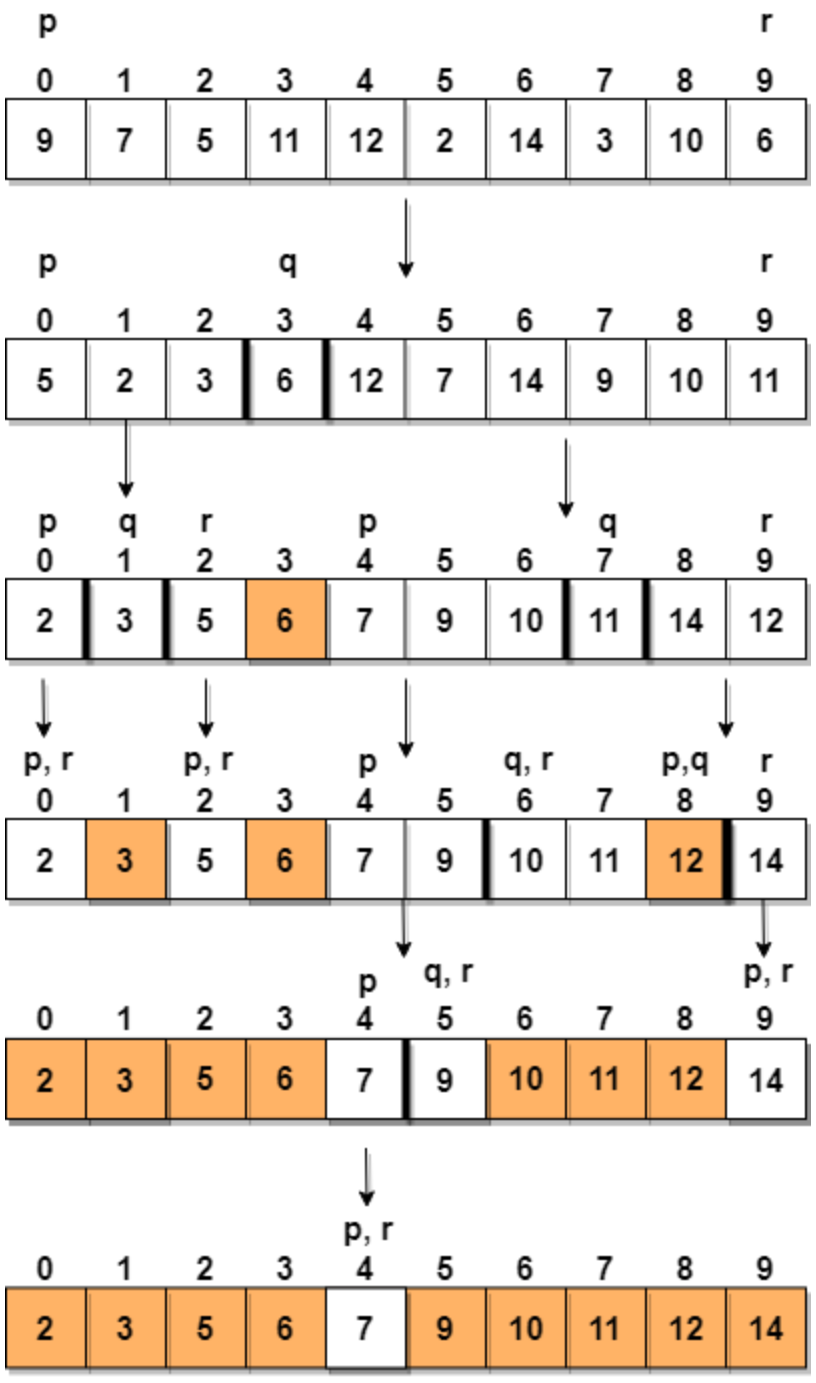
How Quick Sorting Works?

Following are the steps involved in quick sort algorithm:

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.
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.
3. And the **pivot** element will be at its final **sorted** position.
4. The elements to the left and right, may not be sorted.
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.

Let's consider an array with values {9, 7, 5, 11, 12, 2, 14, 3, 10, 6}

Below, we have a pictorial representation of how quick sort will sort the given array.



In step 1, we select the last element as the **pivot**, which is `6` in this case, and call for `partitioning`, hence re-arranging the array in such a way that `6` will be placed in its final position and to its left will be all the elements less than it and to its right, we will have all the elements greater than it.

Then we pick the subarray on the left and the subarray on the right and select a **pivot** for them, in the above diagram, we chose `3` as pivot for the left subarray and `11` as pivot for the right subarray.

And we again call for `partitioning`.

Implementing Quick Sort Algorithm

Below we have a simple C program implementing the Quick sort algorithm:

```
// simple C program for Quick Sort

#include <stdio.h>

int partition(int a[], int beg, int end);

void quickSort(int a[], int beg, int end);

void main()

{

    int i;

    int arr[10]={90,23,101,45,65,28,67,89,34,29};
```

```
    quickSort(arr, 0, 9);

    printf("\n The sorted array is: \n");

    for(i=0;i<10;i++)

    printf(" %d\t", arr[i]);

}

int partition(int a[], int beg, int end)

{

    int left, right, temp, loc, flag;

    loc = left = beg;

    right = end;

    flag = 0;

    while(flag != 1)

    {

        while((a[loc] <= a[right]) && (loc!=right))

        right--;

        if(loc==right)

        flag =1;

        else if(a[loc]>a[right])

        {

            temp = a[loc];

            a[loc] = a[right];

            a[right] = temp;

            loc = right;

        }

        if(flag!=1)

        {

            while((a[loc] >= a[left]) && (loc!=left))

            left++;

            if(loc==left)

            flag =1;
```

```
            else if(a[loc] < a[left])
            {
                temp = a[loc];
                a[loc] = a[left];
                a[left] = temp;
                loc = left;
            }
        }
    }
    return loc;
}
void quickSort(int a[], int beg, int end)
{
    int loc;
    if(beg<end)
    {
        loc = partition(a, beg, end);
        quickSort(a, beg, loc-1);
        quickSort(a, loc+1, end);
    }
}
```

Copy

```
The sorted array is:
23      28      29      34      45      65      67      89      90      101
```

Complexity Analysis of Quick Sort

For an array, in which **partitioning** leads to unbalanced subarrays, to an extent where on the left side there are no elements, with all the elements greater than the **pivot**, hence on the right side.

And if keep on getting unbalanced subarrays, then the running time is the worst case, which is $O(n^2)$

Where as if **partitioning** leads to almost equal subarrays, then the running time is the best, with time complexity as **O(n*log n)**.

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)**

As we know now, that if subarrays **partitioning** produced after partitioning are unbalanced, quick sort will take more time to finish. If someone knows that you pick the last index as **pivot** all the time, they can intentionally provide you with array which will result in worst-case running time for quick sort.

To avoid this, you can pick random **pivot** element too. It won't make any difference in the algorithm, as all you need to do is, pick a random element from the array, swap it with element at the last index, make it the **pivot** and carry on with quick sort.

- Space required by quick sort is very less, only $O(n*\log n)$ additional space is required.
- Quick sort is not a stable sorting technique, so it might change the occurence of two similar elements in the list while sorting.

Now that we have learned Quick sort algorithm, you can check out these sorting algorithms and their applications as well:

- Insertion Sort
- Selection Sort
- Bubble Sort
- Merge sort
- Heap Sort
- Counting Sort