

Quick Sort

Quick sort is based on the divide and conquer approach where:

It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of Quick Sort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

Array is divided into subarrays by selecting a pivot element

Array should be divided in such a way that all the less than pivot are kept on left side and elements greater are on right side of the selected pivot element.

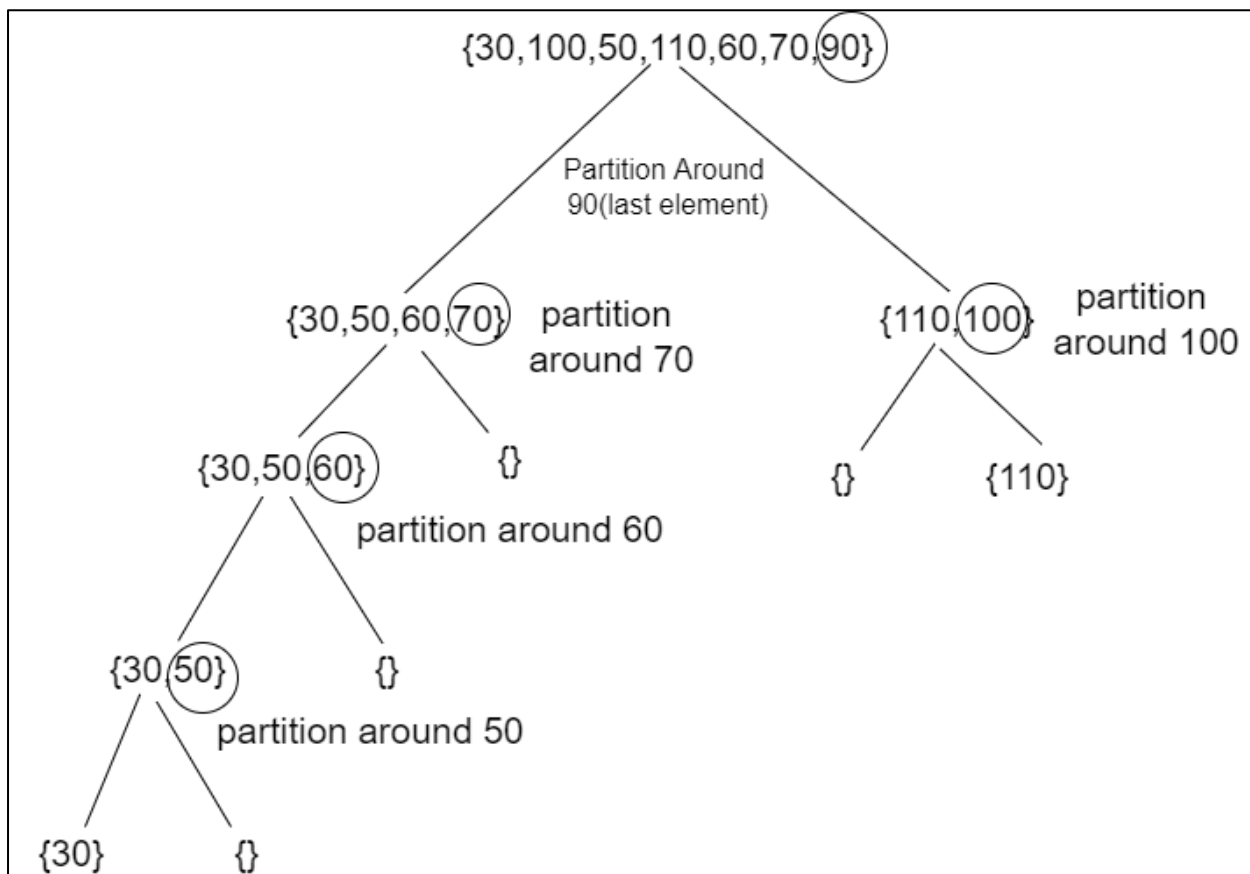
Both the left and right subarrays are also divided using same approach until subarray contains a single element

Now, elements are already sorted. Just combined to form two sorted arrays

Illustration of how partition functions works:

arr[] = {30, 100, 50, 110, 60, 70, 90}

Index 0 1 2 3 4 5 6



We start from left keeping track of index of smaller or equal i. While doing it if we find a smaller element , we swap current element with arr[i] otherwise we ignore it .

Low = 0, high = 6,

Pivot = arr[h] = 90 (Taking last element as pivot)

Initialize index of smaller element ,

i = -1

Traverse elements from j = low to high-1

j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 0

arr[] = { **30**, 100, 50, 110, 60, 70, 90 } //no changes

j = 1 : Since arr[j] > pivot, do nothing

// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 1

arr[] = { **30**, **50**, **100**, 110, 60, 70, 90 } // We swap 100 and 50

j = 3 : Since arr[j] > pivot, do nothing

// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])

i = 2

arr[] = { **30**, 50, **60**, 110, **100**, 70, 90 } // 100 and 60 Swapped

j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]

i = 3

arr[] = { **30**, 50, 60, **70**, **100**, **110**, 90 } // 110 and 70 Swapped

We come out of loop because j is now equal to high-1.

Finally we place pivot at correct position by swapping

arr[i+1] and arr[high] (or pivot)


arr[] = {30, 50, 60, 70, 90, 110, 100} // 100 and 90 Swapped

Now 90 is at its correct place. All elements smaller than 70 are before it and all elements greater than 90 are after it.

Algorithm for Partition Function:


```
partition(arr[], low, high)
    pivot = arr[high]
    i = low
    for j := low to high - 1 do
        if arr[j] <= pivot then
            swap arr[i] with arr[j]
            i = i + 1
    swap arr[i] with arr[high]
    return i
```

C++ Code for Partition Function:



```
1  int iPartition(int array[], int startingIndex, int endingIndex)
2  {
3      int partitioningIndexpivot = array[endingIndex];
4      // Here, i is Index of smaller element
5      int i = startingIndex - 1;
6
7      for (int j = startingIndex; j <= endingIndex - 1; j++)
8      {
9          if (array[j] < partitioningIndexpivot)
10         {
11             i++;
12
13             swap(array[i], array[j]);
14         }
15     }
16     swap(array[i + 1], array[endingIndex]);
17
18     return i + 1;
19 }
```

C++ Code for Recursive Quicksort Function:



```
1 void qSort(int array[], int startingIndex, int endingIndex)
2 {
3     if (startingIndex < endingIndex)
4     {
5         int partitioningIndex = iPartition(array, startingIndex, endingIndex);
6
7         // Before partitioningIndex
8         qSort(array, startingIndex, partitioningIndex - 1);
9
10        // After partitioningIndex
11        qSort(array, partitioningIndex + 1, endingIndex);
12    }
13 }
```

Analysis of Quicksort:

1. Best Case Complexity

The Best case occurs when the pivot element is the middle element or near the middle element so time could be saved since all to the left of it or to the right of it are already partitioned.

$$T(n) = 2T(n/2) + \Theta(n)$$

The time complexity is **$O(n \cdot \log n)$** .

2. Average Case Complexity

It occurs when the elements of the array are not properly increasing or not properly decreasing.

$$T(n) = T(n/9) + T(9n/10) + \Theta(n)$$

The average time complexity of quicksort is **$O(n \cdot \log n)$** .

3. Worst Case Complexity

Worst case occurs when the pivot element is either greatest or smallest element. Sometimes, if we pick pivot element as last element in sorted array the worst case would occur.

$$T(n) = T(0) + T(n-1) + \Theta(n)$$

which is equivalent to

$$T(n) = T(n-1) + \Theta(n)$$

The worst-case time complexity of quicksort is **$O(n^2)$** .

C++ Implementation for the above array:

```
#include <bits/stdc++.h>
using namespace std;

int iPartition(int array[], int startingIndex, int endingIndex)
{
    int partitioningIndexpivot = array[endingIndex];
    // Here,i is Index of smaller element
    int i = startingIndex - 1;

    for (int j = startingIndex; j <= endingIndex - 1; j++)
    {
        if (array[j] < partitioningIndexpivot)
        {
            i++;

            swap(array[i], array[j]);
        }
    }
    swap(array[i + 1], array[endingIndex]);
    return i + 1;
}

void qSort(int array[], int startingIndex, int endingIndex)
{
    if (startingIndex < endingIndex)
    {
        int partitioningIndex = iPartition(array, startingIndex, endingIndex);

        // Before partitioningIndex
        qSort(array, startingIndex, partitioningIndex - 1);

        // After partitioningIndex
        qSort(array, partitioningIndex + 1, endingIndex);
    }
}

int main()
{
    int array[] = {30, 100, 50, 110, 60, 70, 90};

    int n = sizeof(array) / sizeof(array[0]);

    qSort(array, 0, n - 1);
    for (int x : array)
        cout << x << " ";
}
```

OUTPUT: 30 50 60 70 90 100 110