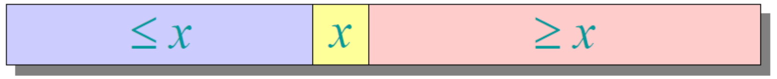
Quick Sort

SWE2016-44

Quick Sort

- **Divide and Conquer algorithm**
  1. **Divide:** Partition the array into two subarrays around a **pivot** x such that $elements\ in\ lower\ subarray \le x \le elements\ in\ upper\ subarray.$
  2. **Conquer:** Recursively sort the two subarrays.
  3. **Combine:** Trivial.

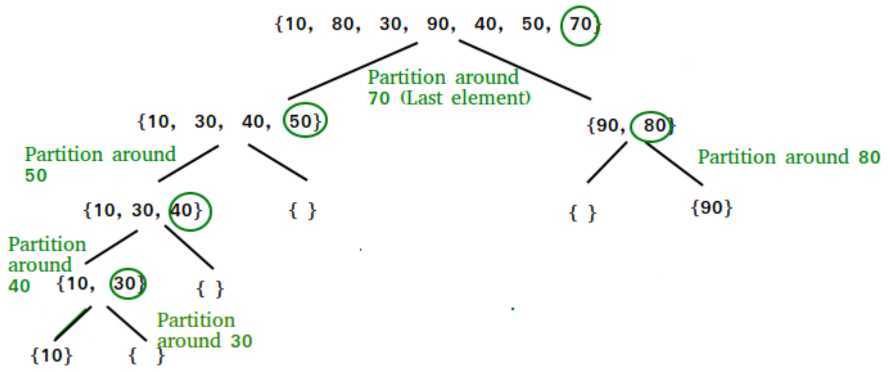


- It picks an element as **pivot** and partitions the given array around the picked pivot. There are many different versions of Quick Sort that pick pivot in different ways:
  - Always pick first element as pivot
  - **Always pick last element as pivot**
  - **Pick a random element as pivot (Randomized Quick Sort)**
  - **Pick median as pivot**

- The key process is partition()

Quick Sort 1

- Always pick last element as pivot



Pseudo Code for recursive quickSort function

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

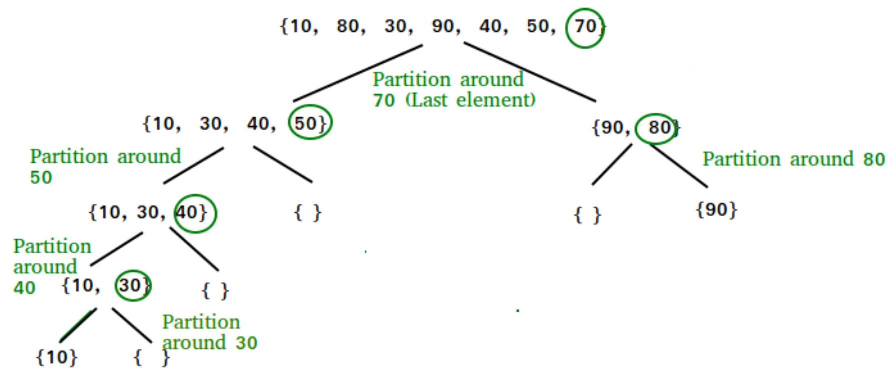
```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Quick Sort 1

- Always pick last element as pivot



arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70
Initialize index of smaller element, **i = -1**
Traverse elements from j = low to high-1

Pseudo Code for recursive quickSort function

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

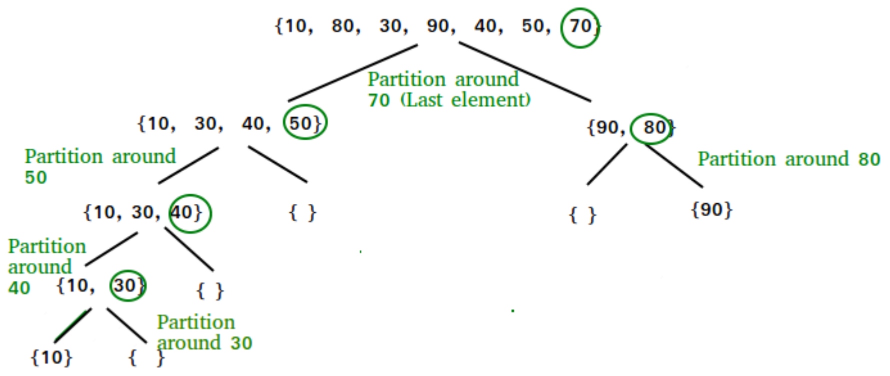
```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Quick Sort 1

- Always pick last element as pivot



arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70
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[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same

Pseudo Code for recursive quickSort function

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

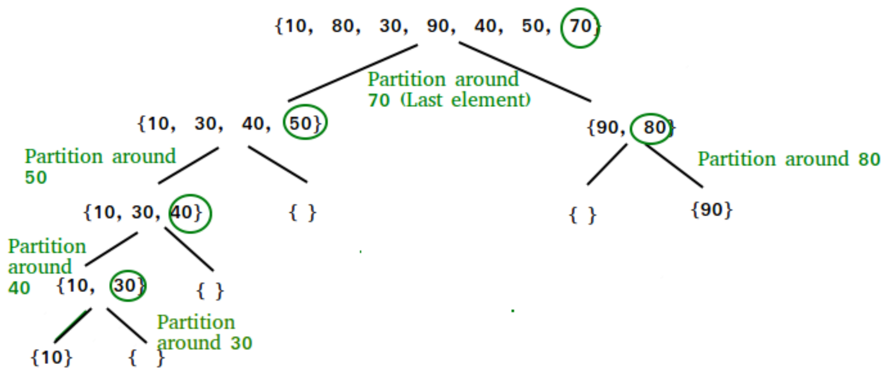
```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Quick Sort 1

- Always pick last element as pivot



arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70
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[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same

j = 1: Since arr[j] > pivot, do nothing // No change in i and arr[]

Pseudo Code for recursive quickSort function

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

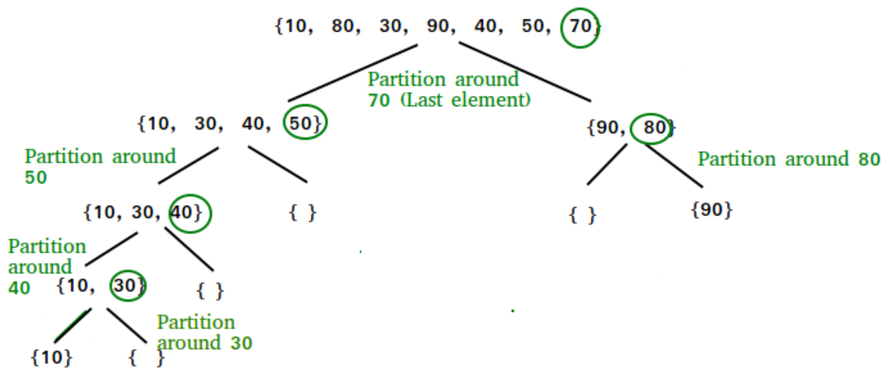
```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Quick Sort 1

- Always pick last element as pivot



arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes: 0   1   2   3   4   5   6

low = 0, high = 6, pivot = arr[h] = 70
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[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same

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[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

Pseudo Code for recursive quickSort function

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

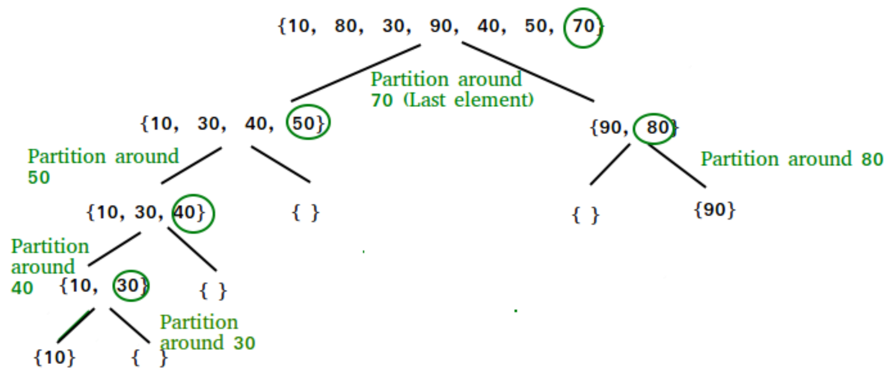
```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Quick Sort 1

- Always pick last element as pivot



arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70
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[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same

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[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3: Since arr[j] > pivot, do nothing // No change in i and arr[]

Pseudo Code for recursive quickSort function

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

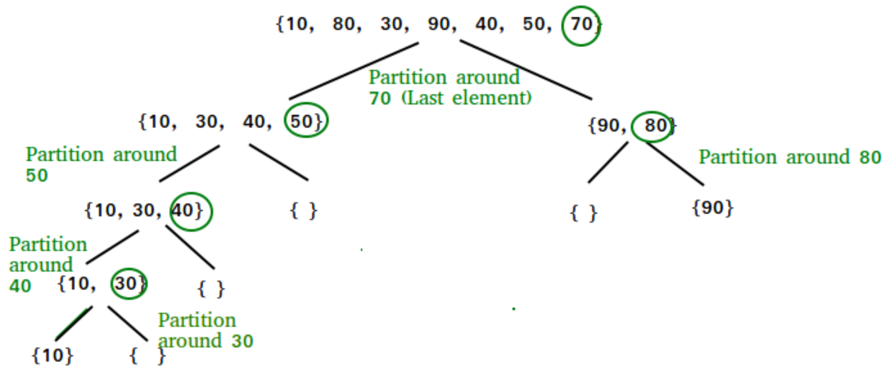
```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Quick Sort 1

- Always pick last element as pivot



arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70
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[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same

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[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

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[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

Pseudo Code for recursive quickSort function

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

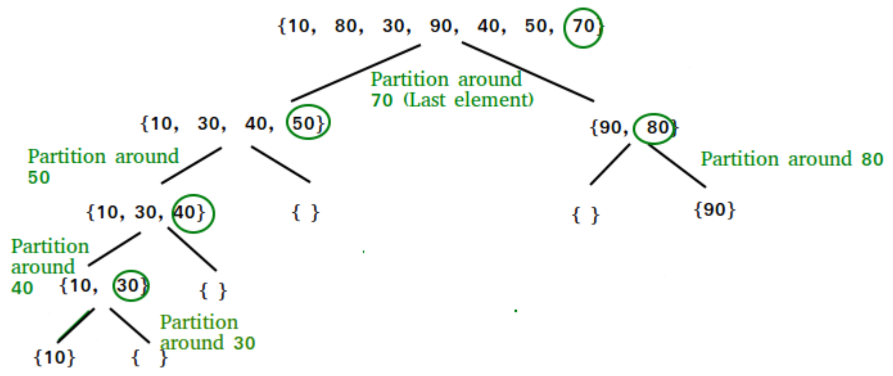
```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Quick Sort 1

- Always pick last element as pivot



arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70
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[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same

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[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

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[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

j = 5: Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

Pseudo Code for recursive quickSort function

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

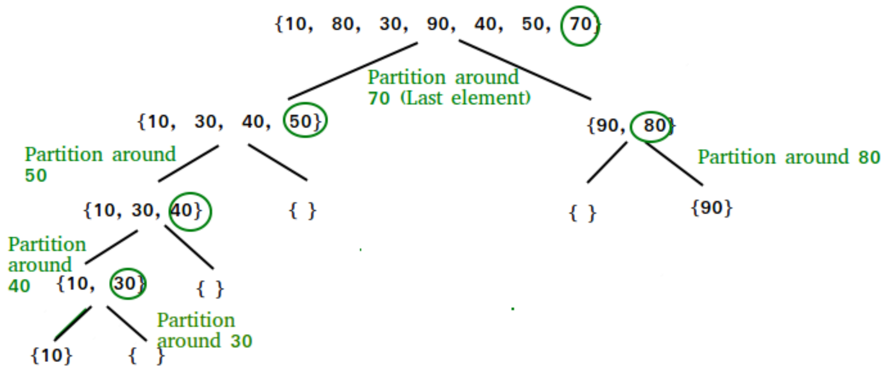
```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Quick Sort 1

- Always pick last element as pivot



arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes: 0   1   2   3   4   5   6

low = 0, high = 6, pivot = arr[h] = 70
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[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same

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[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

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[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

j = 5: Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 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[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Pseudo Code for recursive quickSort function

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

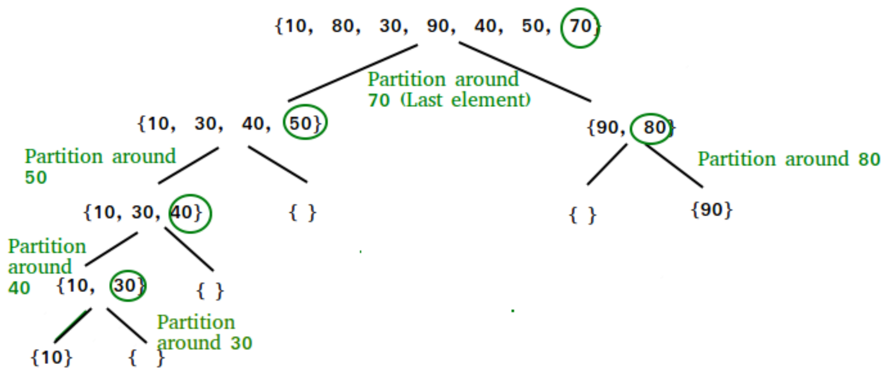
```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Quick Sort 1

- Always pick last element as pivot



arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes: 0 1 2 3 4 5 6

low = 0, high = 6, pivot = arr[h] = 70
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[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same

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[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

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[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

j = 5: Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 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[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

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

Pseudo Code for recursive quickSort function

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low - 1)  // Index of smaller element

    for (j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++;    // increment index of smaller element
            swap arr[i] and arr[j]
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Analysis of Quick Sort 1

- Time taken by QuickSort in general can be written as following:

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

  - The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.

- The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases:
  1. Worst Case
  2. Best Case
  3. Average Case

Analysis of Quick Sort 1

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

- **Worst Case**
  - The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order

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

- **Best Case**
  - The best case occurs when the partition process always picks the middle element as pivot.

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

- **Average Case**
  - Consider all possible permutation of array and calculate time taken by every permutation which doesn’t look easy.
  - We can get an idea of average case by considering the case when partition puts $O(n/9)$ elements in one set and $O(9n/10)$ elements in other set.

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + \Theta(n)$$

Analysis of Quick Sort 1

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

- **Worst Case** $\boldsymbol{\Theta(n^2)}$
  - The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order

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

- **Best Case** $\boldsymbol{\Theta(n \log n)}$ ← case 2 of Master Theorem
  - The best case occurs when the partition process always picks the middle element as pivot.

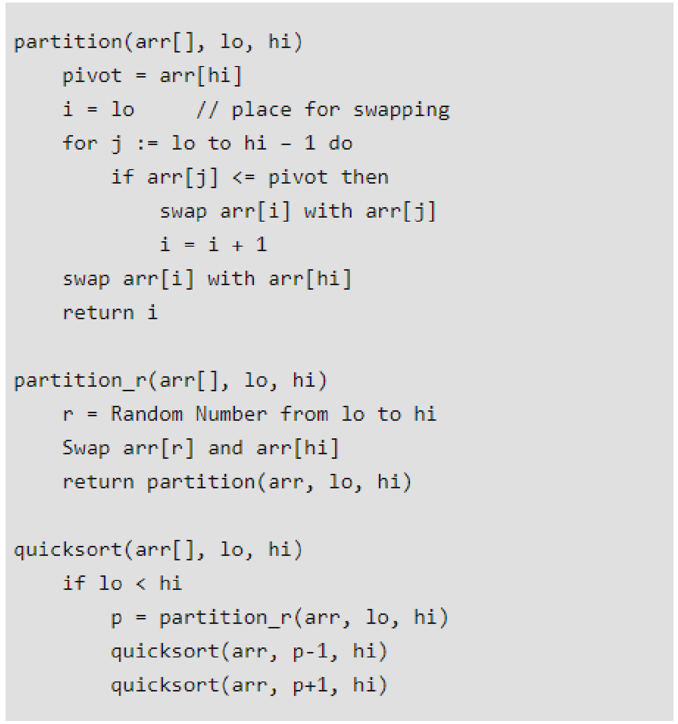
$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

- **Average Case** $\boldsymbol{\Theta(n\ log\, n)}$
  - Consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.
  - We can get an idea of average case by considering the case when partition puts $O(n/9)$ elements in one set and $O(9n/10)$ elements in other set.

$$T(n) = T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + \Theta(n)$$

Quick Sort 2

- **Random Pivoting**
  - Using a randomly generated pivot we can further improve the time complexity of QuickSort.
  - An example:



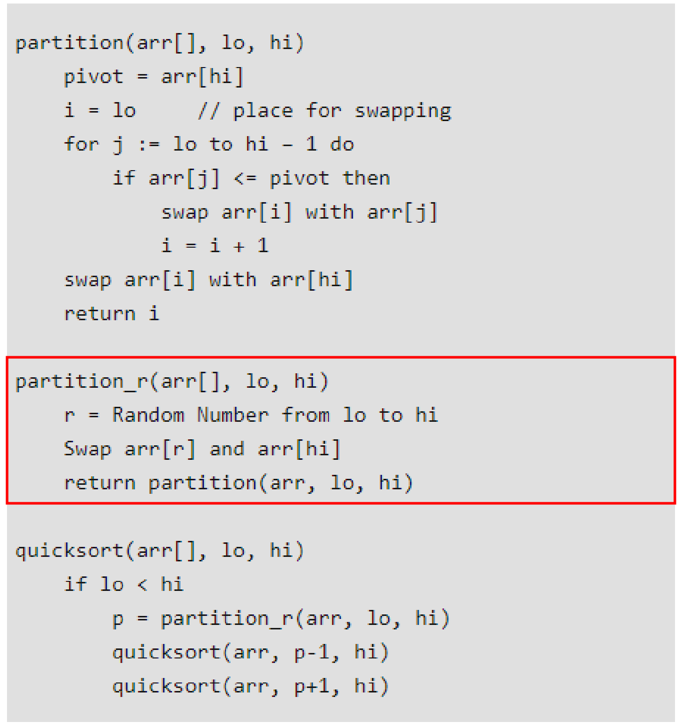
```
partition(arr[], lo, hi)
    pivot = arr[hi]
    i = lo     // place for swapping
    for j := lo to hi – 1 do
        if arr[j] <= pivot then
            swap arr[i] with arr[j]
            i = i + 1
    swap arr[i] with arr[hi]
    return i

partition_r(arr[], lo, hi)
    r = Random Number from lo to hi
    Swap arr[r] and arr[hi]
    return partition(arr, lo, hi)

quicksort(arr[], lo, hi)
    if lo < hi
        p = partition_r(arr, lo, hi)
        quicksort(arr, p-1, hi)
        quicksort(arr, p+1, hi)
```

Quick Sort 2

- **Random Pivoting**
  - Using a randomly generated pivot we can further improve the time complexity of QuickSort.
  - An example:



```
partition(arr[], lo, hi)
    pivot = arr[hi]
    i = lo     // place for swapping
    for j := lo to hi – 1 do
        if arr[j] <= pivot then
            swap arr[i] with arr[j]
            i = i + 1
    swap arr[i] with arr[hi]
    return i

partition_r(arr[], lo, hi)
    r = Random Number from lo to hi
    Swap arr[r] and arr[hi]
    return partition(arr, lo, hi)

quicksort(arr[], lo, hi)
    if lo < hi
        p = partition_r(arr, lo, hi)
        quicksort(arr, p-1, hi)
        quicksort(arr, p+1, hi)
```

Analysis of Quick Sort 2

- **Random Pivoting**
    - In Randomized Quick Sort, we use random number to pick the next pivot (or we randomly shuffle the array)
    - Time complexity of some randomized algorithms is dependent on value of random variable. Such Randomized algorithms are called **Las Vegas Algorithms**. These algorithms are typically analyzed for **expected worst case**.
    - To compute expected time taken in worst case, all possible values of the used random variable needs to be considered in worst case and time taken by every possible value needs to be evaluated. Average of all evaluated times is the expected worst case time complexity.

Analysis of Quick Sort 2

- **Random Pivoting**
  - Let $T(n)$ = the random variable for the running time of randomized quicksort on an input of size n, assuming random numbers are independent.
  - For $k = 0, 1, \dots, n-1$, define the indicator random variable

$$X_k = \begin{cases} 1 & \text{if PARTITION generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise} \end{cases}$$

  - $E[X_k] = \Pr\{X_k = 1\} = \frac{1}{n}$, since all splits are equally likely, assuming elements are distinct.

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split,} \\ T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split,} \\ \vdots & \\ T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split,} \end{cases}$$

$$= \sum_{k=0}^{n-1} X_k\big(T(k) + T(n-k-1) + \Theta(n)\big).$$

Analysis of Quick Sort 2

- **Random Pivoting**

$$T(n) = \sum_{k=0}^{n-1} X_k \left(T(k) + T(n-k-1) + \Theta(n)\right)$$

Analysis of Quick Sort 2

- **Random Pivoting**

$$T(n) = \sum_{k=0}^{n-1} X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)$$

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\right]$$

Take expectations of both sides.

Analysis of Quick Sort 2

- **Random Pivoting**

$$T(n) = \sum_{k=0}^{n-1} X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)$$

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\right]$$

$$= \sum_{k=0}^{n-1} E\big[X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\big]$$

Linearity of expectation.

Analysis of Quick Sort 2

- **Random Pivoting**

$$T(n) = \sum_{k=0}^{n-1} X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)$$

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\right]$$

$$= \sum_{k=0}^{n-1} E\big[X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\big]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

Independence of X_k from other random choices.

Analysis of Quick Sort 2

- **Random Pivoting**

$$T(n) = \sum_{k=0}^{n-1} X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)$$

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\right]$$

$$= \sum_{k=0}^{n-1} E\big[X_k\big(T(k) + T(n-k-1) + \Theta(n)\big)\big]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$$

Linearity of expectation; $E[X_k] = \frac{1}{\mathrm{n}}$.

Analysis of Quick Sort 2

- **Random Pivoting**

$$T(n) = \sum_{k=0}^{n-1} X_k\left(T(k) + T(n-k-1) + \Theta(n)\right)$$

$$E[T(n)] = E\left[\sum_{k=0}^{n-1} X_k\left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E\left[X_k\left(T(k) + T(n-k-1) + \Theta(n)\right)\right]$$

$$= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]$$

$$= \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$$

Summations have identical terms.

$$= \frac{2}{n}\sum_{k=0}^{n-1} E[T(k)] + \Theta(n)$$

Analysis of Quick Sort 2

- **Random Pivoting**

$$E[T(n)] = \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

(The $k = 0, 1$ terms can be absorbed in the $\Theta(n)$.)

Prove: $E[T(n)] \leq an\log n$ for constant $a > 0$.

- Choose a large enough so that $an\log n$ dominates $E[T(n)]$ for sufficiently small $n \geq 2$.
- Use

$$\sum_{k=2}^{n-1} k\log k \leq \frac{1}{2}n^2\log n - \frac{1}{8}n^2$$

Analysis of Quick Sort 2

- **Random Pivoting**

$$E[T(n)] = \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

Analysis of Quick Sort 2

- **Random Pivoting**

$$E[T(n)] = \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

$$\leq \frac{2}{n}\sum_{k=2}^{n-1} ak\log k + \Theta(n)$$

Substitute inductive hypothesis.

Analysis of Quick Sort 2

- **Random Pivoting**

$$E[T(n)] = \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

$$\leq \frac{2}{n}\sum_{k=2}^{n-1} ak\log k + \Theta(n)$$

$$\leq \frac{2a}{n}\left(\frac{1}{2}n^2\log n - \frac{1}{8}n^2\right) + \Theta(n)$$

Use $\sum_{k=2}^{n-1} k\log k \leq \frac{1}{2}n^2\log n - \frac{1}{8}n^2$

Analysis of Quick Sort 2

- **Random Pivoting**

$$E[T(n)] = \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

$$\leq \frac{2}{n}\sum_{k=2}^{n-1} ak\log k + \Theta(n)$$

$$\leq \frac{2a}{n}\left(\frac{1}{2}n^2\log n - \frac{1}{8}n^2\right) + \Theta(n)$$

$$= an\log n - \left(\frac{an}{4} - \Theta(n)\right)$$

Analysis of Quick Sort 2

- **Random Pivoting**

$$E[T(n)] = \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

$$\le \frac{2}{n}\sum_{k=2}^{n-1} ak\log k + \Theta(n)$$

$$\le \frac{2a}{n}\left(\frac{1}{2}n^2\log n - \frac{1}{8}n^2\right) + \Theta(n)$$

$$= an\log n - \left(\frac{an}{4} - \Theta(n)\right)$$

$$\le an\log n$$

if a is chosen large enough so that $\frac{an}{4}$ dominates the $\Theta(n)$.

Analysis of Quick Sort 2

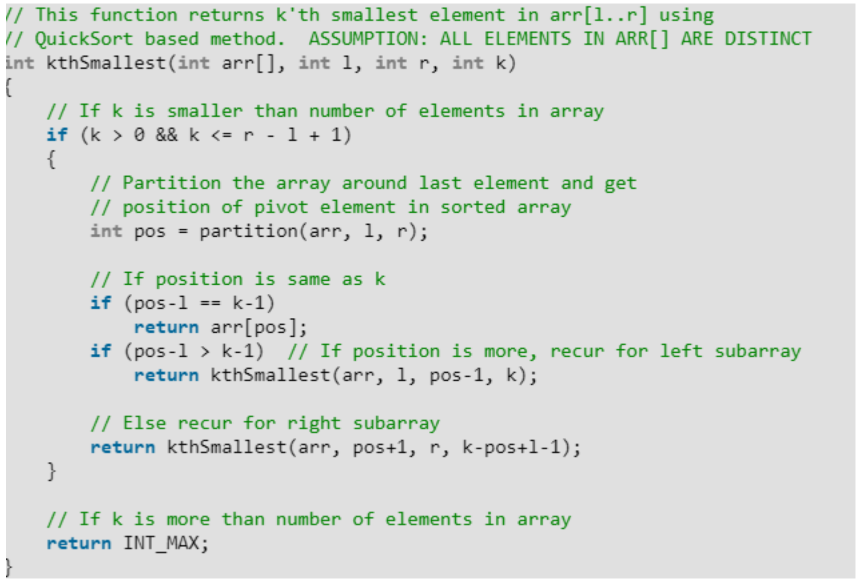
- **Random Pivoting**

$$
\begin{aligned}
E[T(n)] &= \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n) \\
&\le \frac{2}{n}\sum_{k=2}^{n-1} ak\log k + \Theta(n) \\
&\le \frac{2a}{n}\left(\frac{1}{2}n^2\log n - \frac{1}{8}n^2\right) + \Theta(n) \\
&= an\log n - \left(\frac{an}{4} - \Theta(n)\right) \\
&\le an\log n
\end{aligned}
$$

→ Expected worst case time complexity: O(n Log n)

Quick Sort 3

- **Pick median as pivot**
  - Find the median first, then partition the array around the median element.
  - *K*’th Smallest/Largest Element in Unsorted Array method for finding the median.
    - **QuickSelect: worst $\Theta(n^2)$, average $\Theta(n)$**
    - **Modified QuickSelect: worst $\Theta(n)$**
      - Select a pivot that divides array in a balanced way

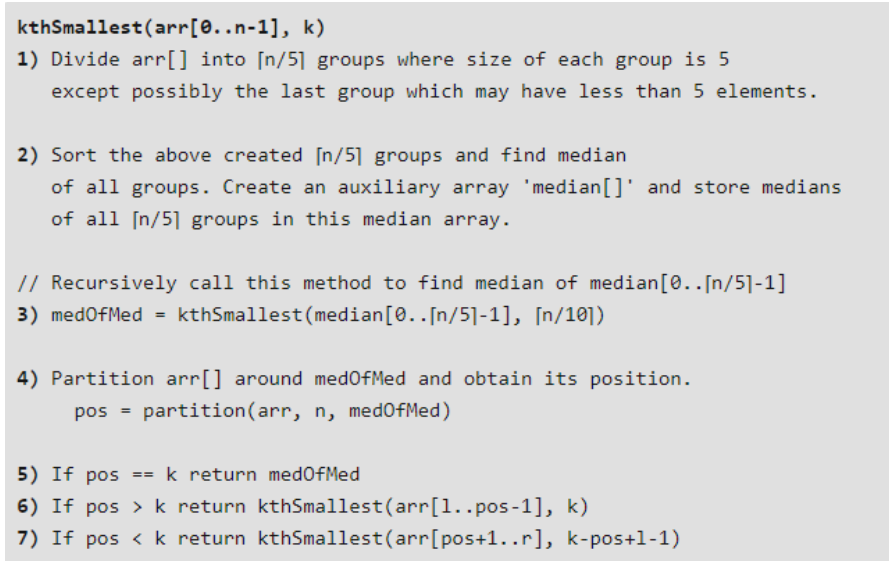


```
// This function returns k'th smallest element in arr[l..r] using
// QuickSort based method.  ASSUMPTION: ALL ELEMENTS IN ARR[] ARE DISTINCT
int kthSmallest(int arr[], int l, int r, int k)
{
    // If k is smaller than number of elements in array
    if (k > 0 && k <= r - l + 1)
    {
        // Partition the array around last element and get
        // position of pivot element in sorted array
        int pos = partition(arr, l, r);

        // If position is same as k
        if (pos-l == k-1)
            return arr[pos];
        if (pos-l > k-1)  // If position is more, recur for left subarray
            return kthSmallest(arr, l, pos-1, k);

        // Else recur for right subarray
        return kthSmallest(arr, pos+1, r, k-pos+l-1);
    }

    // If k is more than number of elements in array
    return INT_MAX;
}
```



```
kthSmallest(arr[0..n-1], k)
1) Divide arr[] into ⌈n/5⌉ groups where size of each group is 5
   except possibly the last group which may have less than 5 elements.

2) Sort the above created ⌈n/5⌉ groups and find median
   of all groups. Create an auxiliary array 'median[]' and store medians
   of all ⌈n/5⌉ groups in this median array.

// Recursively call this method to find median of median[0..⌈n/5⌉-1]
3) medOfMed = kthSmallest(median[0..⌈n/5⌉-1], ⌈n/10⌉)

4) Partition arr[] around medOfMed and obtain its position.
     pos = partition(arr, n, medOfMed)

5) If pos == k return medOfMed
6) If pos > k return kthSmallest(arr[l..pos-1], k)
7) If pos < k return kthSmallest(arr[pos+1..r], k-pos+l-1)
```

→ Worst case time complexity: O(n Log n)

Notes

- Although the worst case time complexity of QuickSort is $\boldsymbol{O(n^2)}$ which is more than many other sorting algorithms like Merge Sort and Heap Sort, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data.
- Why Quick Sort is preferred over Merge Sort for sorting Arrays?
  - → Quick Sort in its general form is an in-place sort (i.e. it doesn't require any extra storage) whereas merge sort requires O(N) extra storage, N denoting the array size which may be quite expensive.
- Why MergeSort is preferred over QuickSort for Linked Lists?
  - → In case of linked lists the case is different mainly due to difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike array, in linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore merge operation of merge sort can be implemented without extra space for linked lists.