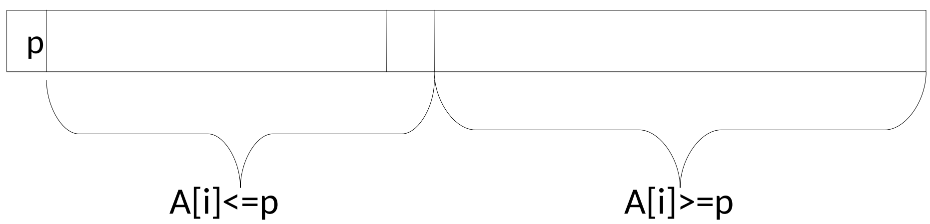
<u>Quick sort</u>

Quick sort is application of divide and conquer. It is inplace algorithm. Quick sort is not stable. Following are the step for quick sort:

- Select a *pivot* (partitioning element) – here, the first element.
- Rearrange the list so that all the elements in the first *s* positions are smaller than or equal to the pivot and all the i elements in the remaining *n-s* positions are larger than or equal to the pivot.



- Exchange the pivot with the last element in the first (i.e., _) sub-array — the pivot is now in its final position
- Sort the two sub-arrays recursively

Partitioning Algorithm

```
Partition(arr , p ,q)

{           x=arr[p], i=p;
for(j=p+1; j<=q; j++)
{
            if(arr[j]<=x)
            {   i=i+1;
            Interchange(arr[i], arr[j]);
            }
}
Interchange(arr[i],   arr[p]);
return (i);
}
```

Quick sort algorithm

```
Quick _sort(arr ,  p,  q)

{            if(p==q)
            {
            return (arr[p]);
            }
            else
             {
             m=Partition(arr, p,  q);
            Quick_sort(arr, p, m-1);
            Quick_sort(arr, , m+1, q);
            return(arr)
}
```

Example

Input: **65 70 75 80 85 60 55 50 45**

P: 65 i

Pass 1: 65 70 75 80 85 60 55 50 45

i. i j if(arr[j]<=P) then swap (A[++i], A[j])

65 70 75 80 85 60 55 50 45

ii. i j if(arr[j]<=P) then swap (A[++i], A[j])

65 70 75 80 85 60 55 50 45

iii. i j if(arr[j]<=P) then swap (A[++i], A[j])

65 70 75 80 85 60 55 50 45

iv. i j if(arr[j]<=P) then swap (A[++i], A[j])

65 70 75 80 85 60 55 50 45

v. i j if(arr[j]<=P) then swap (A[++i], A[j])

65 60 75 80 85 70 55 50 45

vi. i j if(arr[j]<=P) then swap (A[++i], A[j])

65 60 55 80 85 70 75 50 45

vii. i j if(arr[j]<=P) then swap (A[++i], A[j])

65 60 55 50 85 70 75 80 45

viii. i j

if(arr[j]<=P) then swap (A[++i], A[j])

65 60 55 50 45 70 75 80 85

i j > 9 then swap (A[i],P)

45 60 55 50 | 65 | 70 75 80 85

Less than 65 | Greater than 65

Apply same process in both the subsets

45 60 55 50 & 70 75 80 85

Let T(n) be the amount of time taken by quick sort to sort an array of element of size n.

Best case: Occurs when every time partition algorithm divide elements into equal size of groups

$$T(n) = \begin{cases} O(1) & \text{if } n=1 \end{cases}$$

T(n)=

O(n) + T(n/2) +T(n/2) if n>1

Partition /Divide cost 2 sub-array

T(n)=2T(n/2) + n after solving this recurrence equation:

Best case time complexity: $\mathbf{\Theta(n\log_2 n)}$

Worst case: In the worst case the partition algorithm create 0 elements one side and n-1 elements on other side.

T(n)=
- O(1) if n=1
- O(n) + T(0) +T(n-1) if n>1

Partition /Divide cost 2 sub-array

T(n)=n+ T(0) + T(n-1)

=n+ T(n-1) after solving this recurrence equation:

Worst case time complexity: $\mathbf{O(n^2)}$

Note: when array is not at all sorted then quick sort performance best.

Average case: In this case some time we get lucky (equal partition) and some time we get unlucky in next recursion partition (unequal partition).

T(n)=n+ T(0) + T(n-1)------- unlucky case

T(n)=2T(n/2) + n---------------lucky case

Average case time complexity= $\mathbf{\Theta(n\log_2 n)}$

Note: Quick sort give worst case when array is sorted (whether ascending or descending)

Stable sorting techniques: The relative order of repeated elements is not change after sorting then the sorting technique is called stable sorting technique.

e.g:

input: 10_a 11 1 10_b 12

output 1 : 1 10_a 10_b 11 12 --------------- stable sorting

output 2 : 1 10_b 10_a 11 12 ----------------- unstable sorting

Randomized quick sort: In this randomized quick sort difference is only pivot element is chosen randomly. But in normal quick sort pivot is first element.