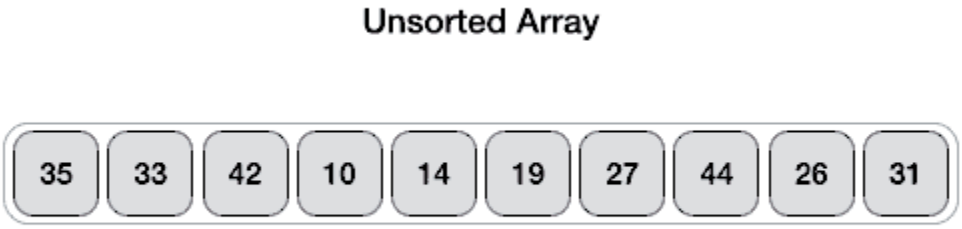
Data Structure and Algorithms - Quick Sort

- Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays.
- A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.
- Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays.
- This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are $O(n^2)$, respectively.

Partition in Quick Sort

Following animated representation explains how to find the pivot value in an array.



The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

Step 1 − Choose the highest index value has pivot
Step 2 − Take two variables to point left and right of the list excluding pivot
Step 3 − left points to the low index
Step 4 − right points to the high
Step 5 − while value at left is less than pivot move right

Step 6 − while value at right is greater than pivot move left
Step 7 − if both step 5 and step 6 does not match swap left and right
Step 8 − if left ≥ right, the point where they met is new pivot

Quick Sort Pivot Pseudocode

The pseudocode for the above algorithm can be derived as −

```
function partitionFunc(left, right, pivot)
   leftPointer = left
   rightPointer = right - 1

   while True do
      while A[++leftPointer] < pivot do
         //do-nothing
      end while

      while rightPointer > 0 && A[--rightPointer] > pivot do
         //do-nothing
      end while

      if leftPointer >= rightPointer
         break
      else
         swap leftPointer,rightPointer
      end if

   end while

   swap leftPointer,right
   return leftPointer

end function
```

Quick Sort Algorithm

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows −

Step 1 − Make the right-most index value pivot
Step 2 − partition the array using pivot value
Step 3 − quicksort left partition recursively
Step 4 − quicksort right partition recursively

Quick Sort Pseudocode

To get more into it, let see the pseudocode for quick sort algorithm −

```
procedure quickSort(left, right)

   if right-left <= 0
      return
   else
      pivot = A[right]
      partition = partitionFunc(left, right, pivot)
      quickSort(left,partition-1)
      quickSort(partition+1,right)
   end if

end procedure
```

Quick sort is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. A large array is partitioned into two arrays one of which holds values smaller than the specified value, say pivot, based on which the partition is made and another array holds values greater than the pivot value.

Implementation in C

```
#include <stdio.h>
#include <stdbool.h>

#define MAX 7

int intArray[MAX] = {4,6,3,2,1,9,7};

void printline(int count) {
```

```
   int i;

   for(i = 0;i < count-1;i++) {
      printf("=");
   }

   printf("=\n");
}

void display() {
   int i;
   printf("[");

   // navigate through all items
   for(i = 0;i < MAX;i++) {
      printf("%d ",intArray[i]);
   }

   printf("]\n");
}

void swap(int num1, int num2) {
   int temp = intArray[num1];
   intArray[num1] = intArray[num2];
   intArray[num2] = temp;
}

int partition(int left, int right, int pivot) {
   int leftPointer = left -1;
   int rightPointer = right;

   while(true) {
      while(intArray[++leftPointer] < pivot) {
         //do nothing
      }

      while(rightPointer > 0 && intArray[--rightPointer] > pivot) {
         //do nothing
      }
```

```
      if(leftPointer >= rightPointer) {
         break;
      } else {
         printf(" item swapped :%d,%d\n",
intArray[leftPointer],intArray[rightPointer]);
         swap(leftPointer,rightPointer);
      }
   }

   printf(" pivot swapped :%d,%d\n", intArray[leftPointer],intArray[right]);
   swap(leftPointer,right);
   printf("Updated Array: ");
   display();
   return leftPointer;
}

void quickSort(int left, int right) {
   if(right-left <= 0) {
      return;
   } else {
      int pivot = intArray[right];
      int partitionPoint = partition(left, right, pivot);
      quickSort(left,partitionPoint-1);
      quickSort(partitionPoint+1,right);
   }
}

int main() {
   printf("Input Array: ");
   display();
   printline(50);
   quickSort(0,MAX-1);
   printf("Output Array: ");
   display();
   printline(50);
}
```

If we compile and run the above program, it will produce the following result −

Output

```
Input Array: [4 6 3 2 1 9 7 ]
================================================
 pivot swapped :9,7
Updated Array: [4 6 3 2 1 7 9 ]
 pivot swapped :4,1
Updated Array: [1 6 3 2 4 7 9 ]
 item swapped :6,2
 pivot swapped :6,4
Updated Array: [1 2 3 4 6 7 9 ]
 pivot swapped :3,3
Updated Array: [1 2 3 4 6 7 9 ]
Output Array: [1 2 3 4 6 7 9 ]
================================================
```

How Quick Sort Works?

1. A pivot element is chosen from the array.
2. You can choose any element from the array as the pivot element.
   Here, we have taken the rightmost (ie. the last element) of the array as the

   pivot element.
   Select a pivot element
3. The elements smaller than the pivot element are put on the left and the elements greater than the pivot element are put on the right.

4. Put all the smaller elements on the left and greater on the right of pivot element
   The above arrangement is achieved by the following steps.

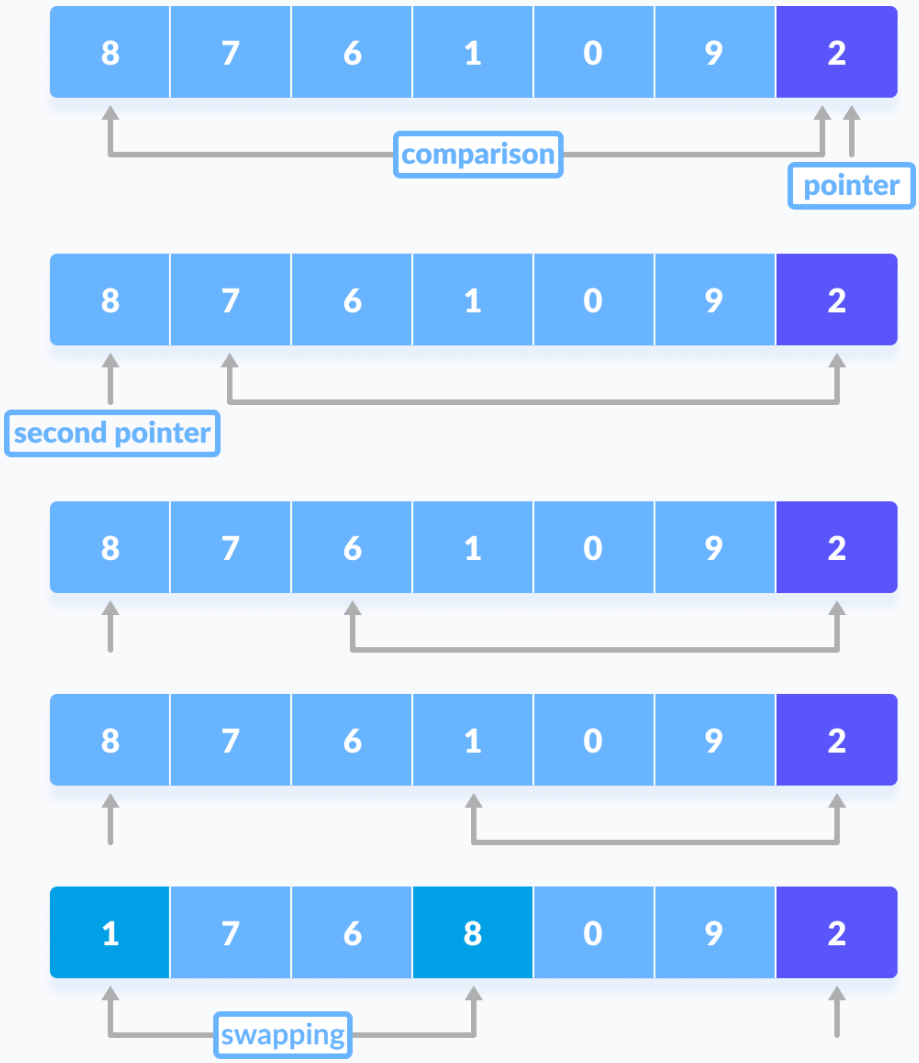
a. A pointer is fixed at the pivot element.

b. The pivot element is compared with the elements beginning from the first index.

c. If the element greater than the pivot element is reached, a second pointer is set for that element.

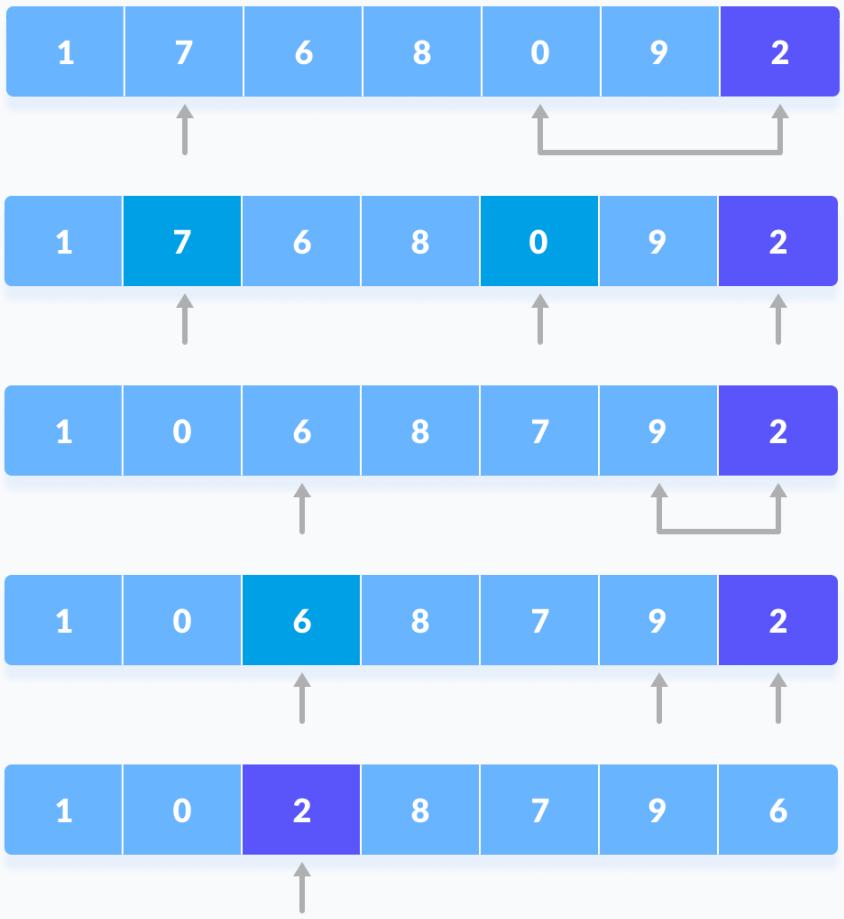
d. Now, the pivot element is compared with the other elements (a third pointer).

e. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.



f. Comparison of pivot element with other elements

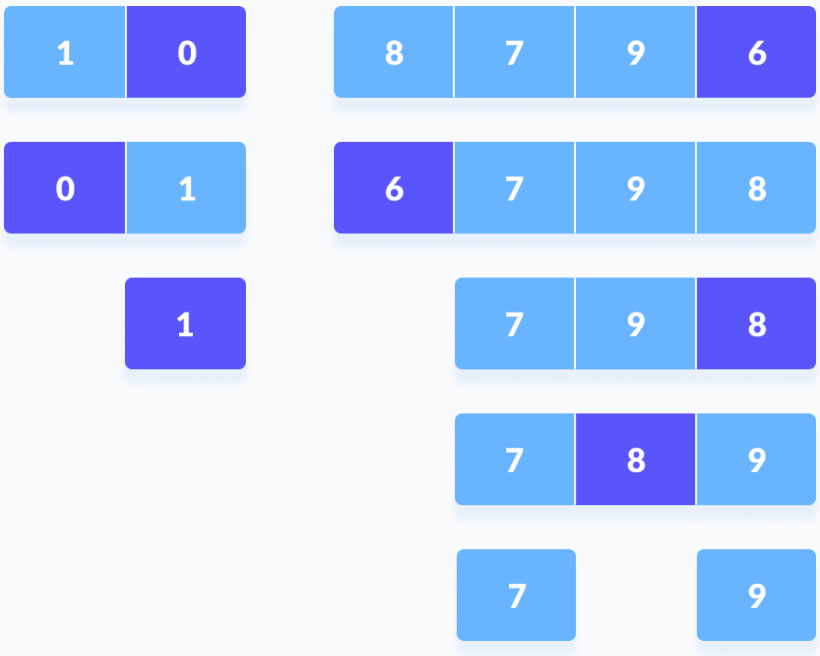
g. The process goes on until the second last element is reached. Finally, the pivot element is swapped with the second pointer.



h. Swap pivot element with the second pointer
i. Now the left and right subparts of this pivot element are taken for further processing in the steps below.

5. Pivot elements are again chosen for the left and the right sub-parts separately. Within these sub-parts, the pivot elements are placed at their

right position. Then, step 2 is repeated.



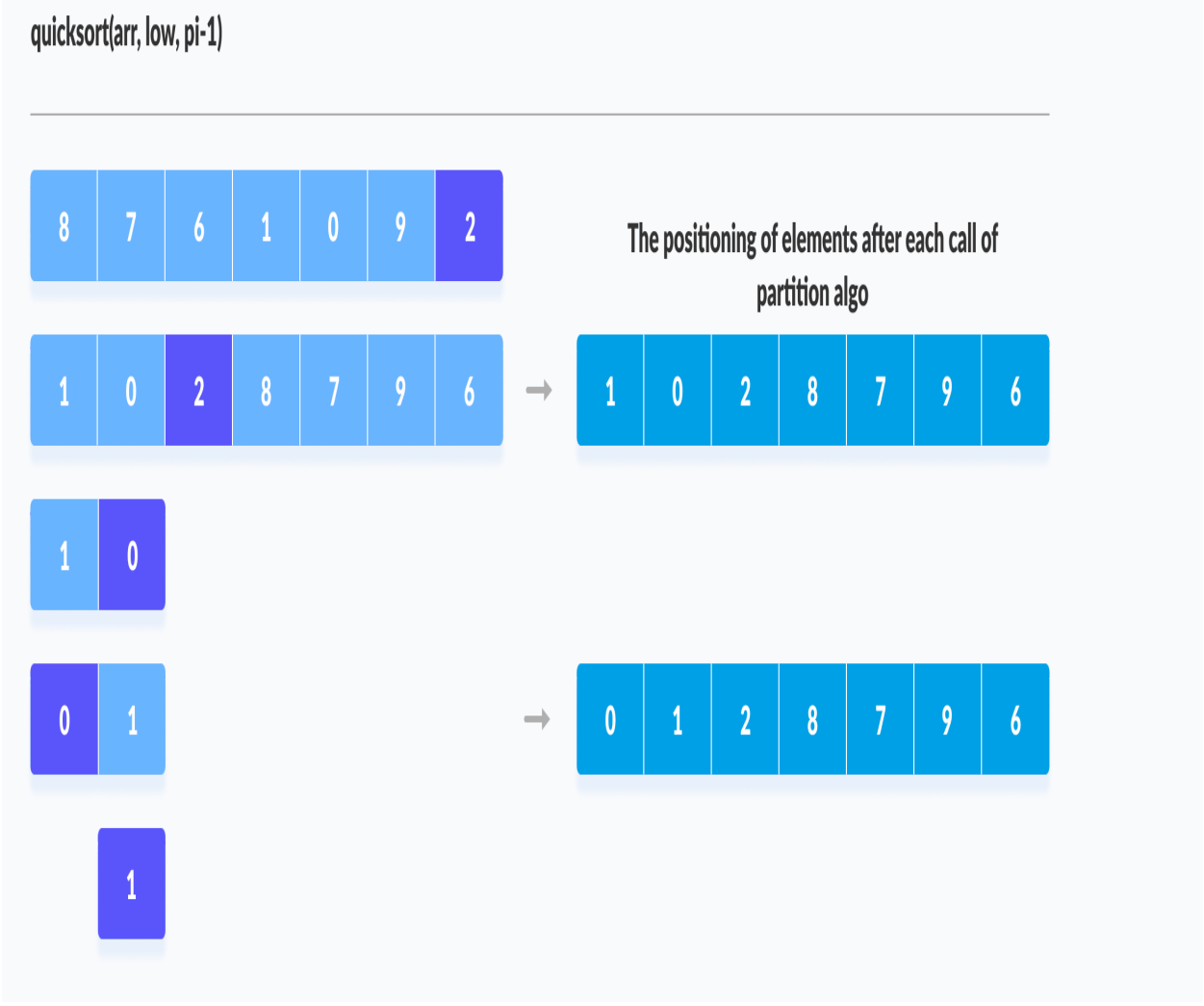
6. Select pivot element of in each half and put at correct place using recursion
7. The sub-parts are again divided into smaller sub-parts until each subpart is formed of a single element.
8. At this point, the array is already sorted.

Quicksort uses recursion for sorting the sub-parts.
On the basis of Divide and conquer approach, quicksort algorithm can be explained as:

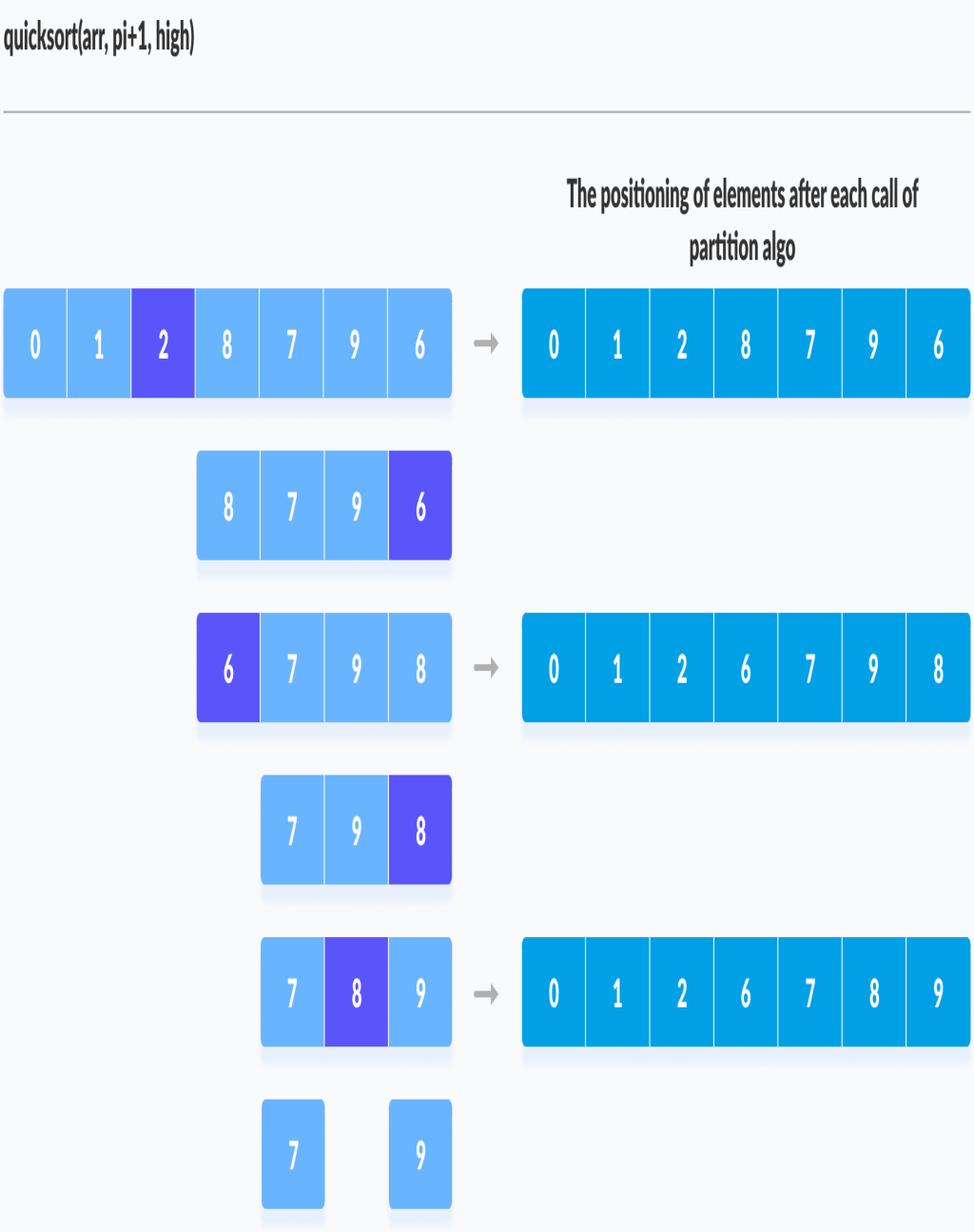
- **Divide**
  The array is divided into subparts taking pivot as the partitioning point. The elements smaller than the pivot are placed to the left of the pivot and the elements greater than the pivot are placed to the right.
- **Conquer**
  The left and the right subparts are again partitioned using the by selecting

pivot elements for them. This can be achieved by recursively passing the subparts into the algorithm.

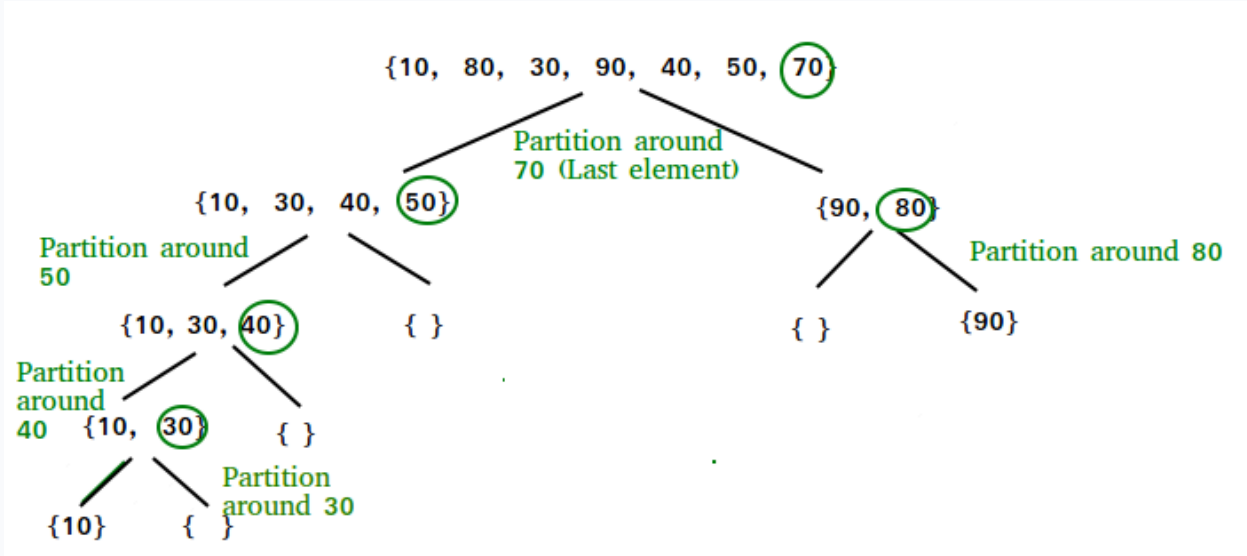
- **Combine**
  This step does not play a significant role in quicksort. The array is already sorted at the end of the conquer step.
  You can understand the working of quicksort with the help of the illustrations below.



Sorting the elements on the left of pivot using recursion



Sorting the elements on the right of pivot using recursion



Quick Sort Algorithm

```
quickSort(array, leftmostIndex, rightmostIndex)
  if (leftmostIndex < rightmostIndex)
    pivotIndex <- partition(array,leftmostIndex, rightmostIndex)
    quickSort(array, leftmostIndex, pivotIndex)
    quickSort(array, pivotIndex + 1, rightmostIndex)
partition(array, leftmostIndex, rightmostIndex)
  set rightmostIndex as pivotIndex
  storeIndex <- leftmostIndex - 1
  for i <- leftmostIndex + 1 to rightmostIndex
  if element[i] < pivotElement
    swap element[i] and element[storeIndex]
    storeIndex++
```

```
  swap pivotElement and element[storeIndex+1]

return storeIndex + 1
```

```
# Quick sort in Python


# Function to partition the array on the basis of pivot element
def partition(array, low, high):

    # Select the pivot element
    pivot = array[high]
    i = low - 1

    # Put the elements smaller than pivot on the left and greater
    #than pivot on the right of pivot
    for j in range(low, high):
        if array[j] <= pivot:
            i = i + 1
            (array[i], array[j]) = (array[j], array[i])

    (array[i + 1], array[high]) = (array[high], array[i + 1])

    return i + 1


def quickSort(array, low, high):
    if low < high:

        # Select pivot position and put all the elements smaller
        # than pivot on left and greater than pivot on right
        pi = partition(array, low, high)
```

```
        # Sort the elements on the left of pivot
        quickSort(array, low, pi - 1)

        # Sort the elements on the right of pivot
        quickSort(array, pi + 1, high)


data = [8, 7, 2, 1, 0, 9, 6]
size = len(data)
quickSort(data, 0, size - 1)
print('Sorted Array in Ascending Order:')
print(data)
```

Quicksort Complexity

Time Complexities

- **Worst Case Complexity [Big-O]**: $O(n^2)$
  It occurs when the pivot element picked is either the greatest or the smallest element.
  This condition leads to the case in which the pivot element lies in an extreme end of the sorted array. One sub-array is always empty and another sub-array contains n - 1 elements. Thus, quicksort is called only on this sub-array.
  However, the quick sort algorithm has better performance for scattered pivots.
- **Best Case Complexity [Big-omega]**: O(n*log n)
  It occurs when the pivot element is always the middle element or near to the middle element.
- **Average Case Complexity [Big-theta]**: O(n*log n)
  It occurs when the above conditions do not occur.

Space Complexity

The space complexity for quicksort is O(log n).

Quicksort Applications

Quicksort is implemented when

- the programming language is good for recursion
- time complexity matters
- space complexity matters

QuickSort

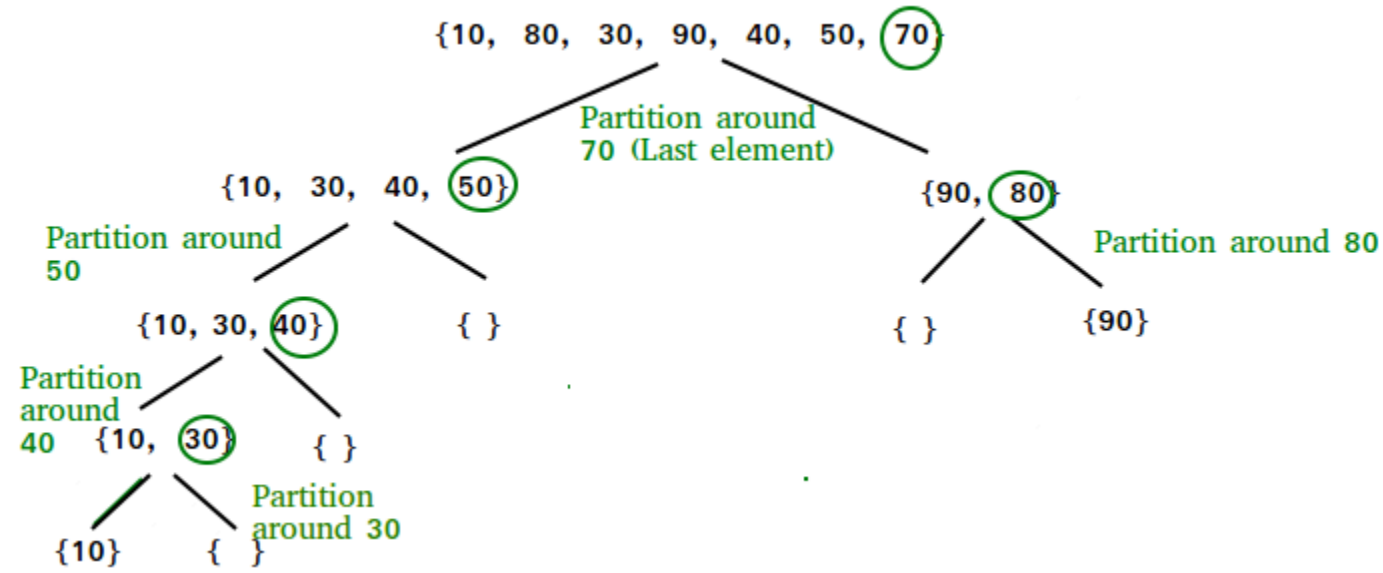
- Quick Sort is a Divide and Conquer algorithm.
- It picks an element as pivot and partitions the given array around the picked pivot.
- There are many different versions of Quick Sort that pick pivot in different ways.

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.
   - The key process in quickSort is partition ().
   - Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

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
    }
}
```



Partition Algorithm

- There can be many ways to do partition, following pseudo code adopts the method.
- The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i.
- While traversing, if we find a smaller element, we swap current element with arr[i].
- Otherwise we ignore current element.
-

```
/* 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)
}
```

Illustration of partition() :

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.

Implementation:

Following are the implementations of QuickSort:

```
/* C++ implementation of QuickSort */

#include <bits/stdc++.h>

using namespace std;


// A utility function to swap two elements
```

```
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* 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 */
int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element
```

```
    for (int 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], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
/* The main function that implements QuickSort
arr[] --> Array to be sorted,
low --> Starting index,
high --> Ending index */
void quickSort(int arr[], int low, int high)
```

```
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);
        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
```

```
        cout << arr[i] << " ";
    cout << endl;
}
// Driver Code
int main()
{
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    cout << "Sorted array: \n";
    printArray(arr, n);
    return 0;
}


//
```

Output:

Sorted array:

1 5 7 8 9 10

Analysis of QuickSort

Time taken by QuickSort in general can be written as following.

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

- 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.

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.
- **Following is recurrence for worst case**.

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

which is equivalent to

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

The solution of above recurrence is (n^2).

Best Case:

- The best case occurs when the partition process always picks the middle element as pivot.
- **Following is recurrence for best case.**

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

The solution of above recurrence is (nLogn). It can be solved using case 2 of Master Theorem.

Average Case:

- To do average case analysis, we need to 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. Following is recurrence for this case.

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

Solution of above recurrence is also O(nLogn)

Although the worst case time complexity of QuickSort is $O(n^2)$ 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. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

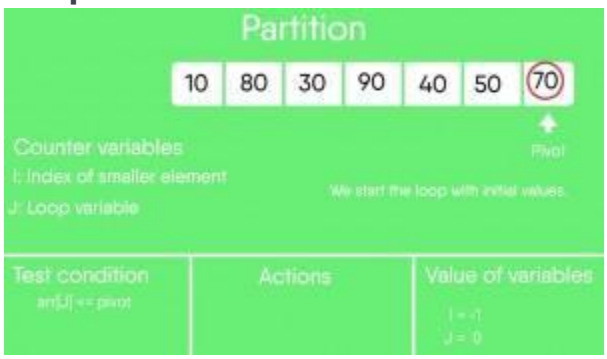
Is QuickSort stable?

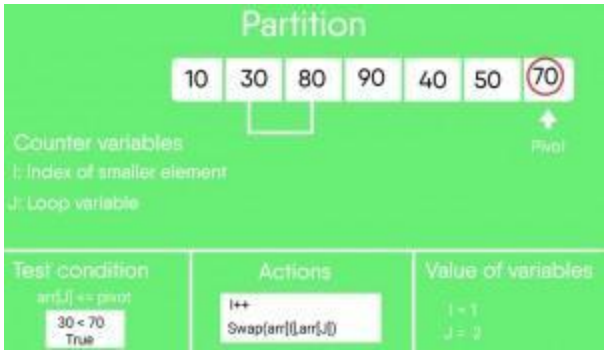
The default implementation is not stable. However any sorting algorithm can be made stable by considering indexes as comparison parameter.

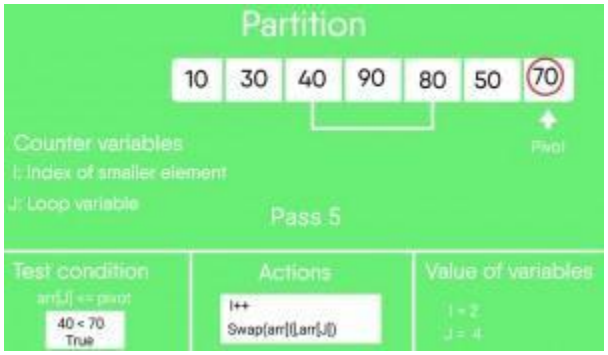
Is QuickSort In-place?

As per the broad definition of in-place algorithm it qualifies as an in-place sorting algorithm as it uses extra space only for storing recursive function calls but not for manipulating the input.

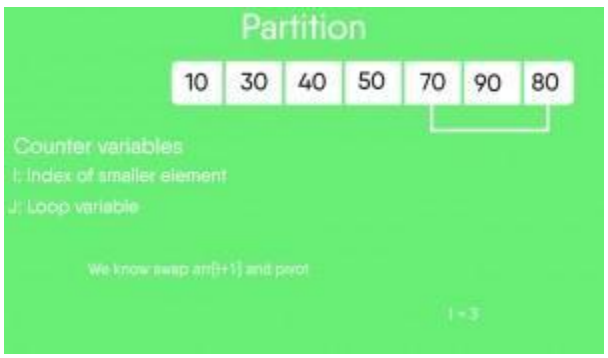
Snapshots:

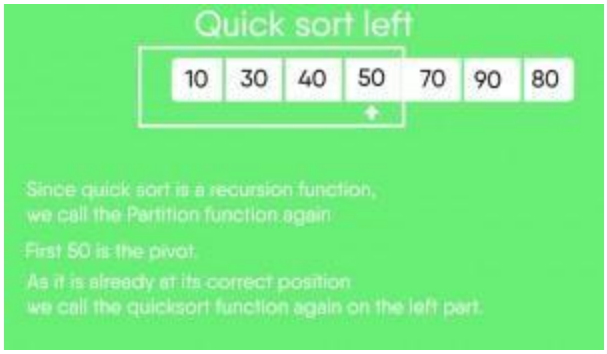



Partition
10 30 80 90 40 50 70
Pivot
Counter variables
I: Index of smaller element
J: Loop variable
Test condition
arr[J] <= pivot
30 < 70
True
Actions
I++
Swap(arr[I],arr[J])
Value of variables
I = 1
J = 2


Partition
10 30 40 90 80 50 70
Pivot
Counter variables
I: Index of smaller element
J: Loop variable
Pass 5
Test condition
arr[J] <= pivot
40 < 70
True
Actions
I++
Swap(arr[I],arr[J])
Value of variables
I = 2
J = 4


Partition
10 30 40 50 80 90 70
Pivot
Counter variables
I: Index of smaller element
J: Loop variable
Before Pass 7, J becomes 6
so we come out of the loop
Test condition
arr[J] <= pivot
Actions
Value of variables
I = 3
J = 6


Partition
10 30 40 50 70 90 80
Counter variables
I: Index of smaller element
J: Loop variable
We know swap arr[I+1] and pivot
I = 3


Quick sort left
10
30
40
50
70
90
80
Since quick sort is a recursion function,
we call the Partition function again
First 50 is the pivot.
As it is already at its correct position
we call the quicksort function again on the left part.