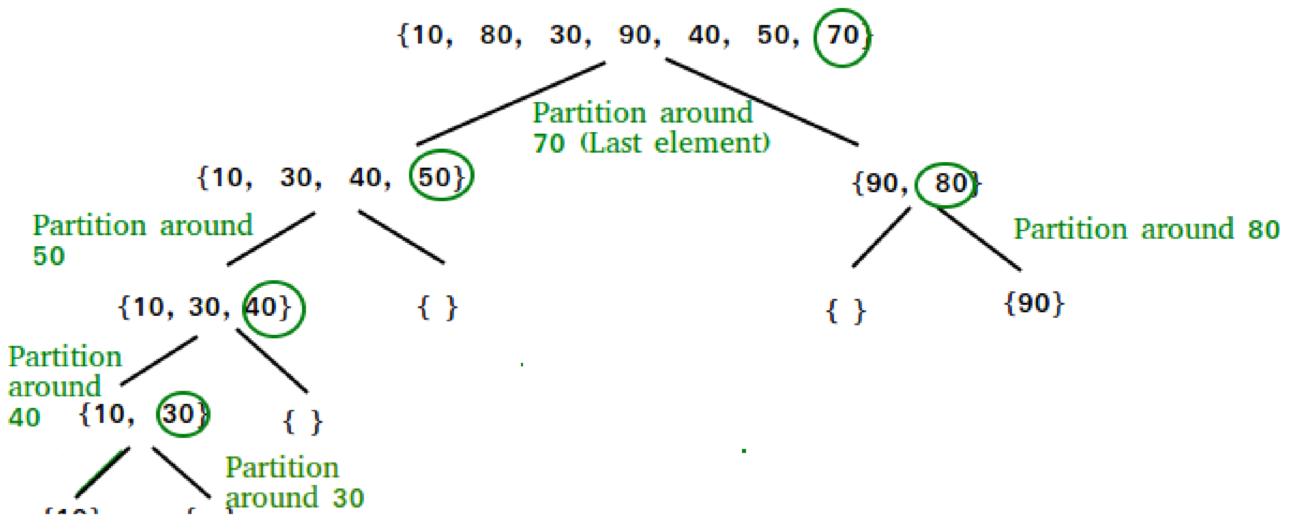
QuickSort

Difficulty Level : Medium • Last Updated : 17 Jan, 2023

Like Merge Sort, **QuickSort** is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- Pick median as the pivot.

The key process in **quickSort** is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a 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.



keep track of the index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise, we ignore the current element.

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
  }
}
```

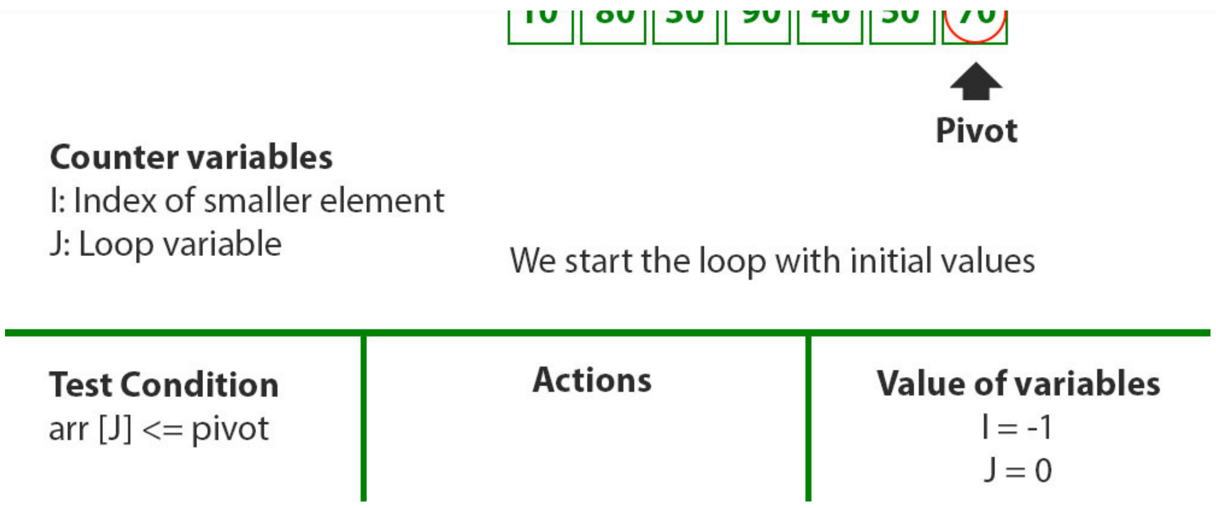
Pseudo code for partition()

```
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low – 1)  // Index of smaller element and indicates the
    // right position of pivot found so far

    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)
  }
```

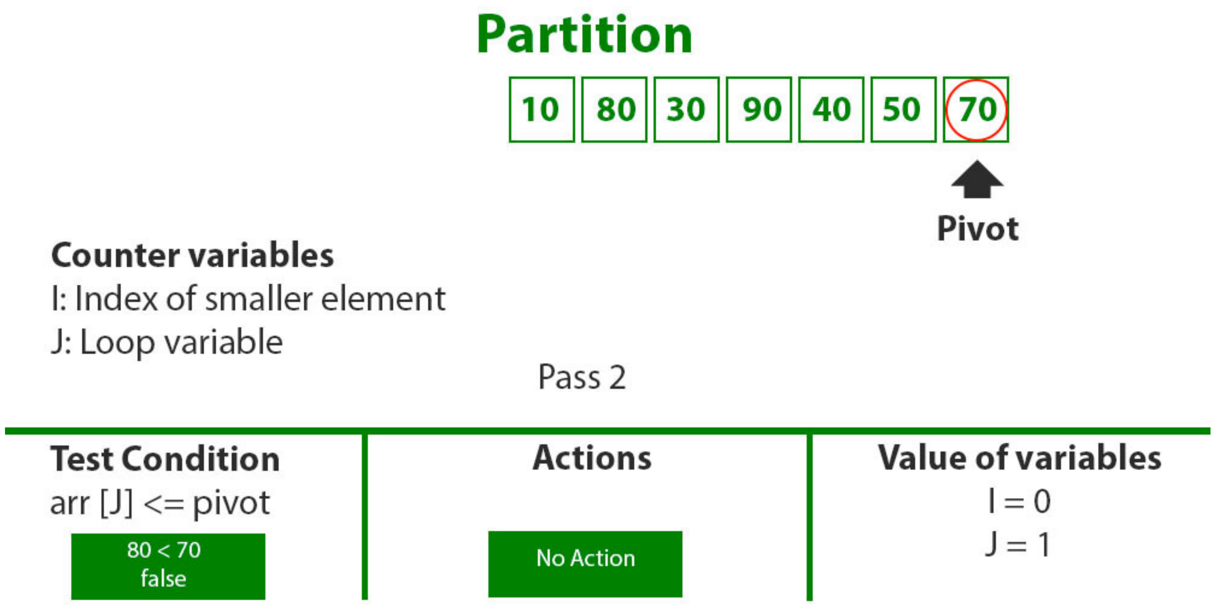
Illustration of partition() :

Consider: arr[] = {10, 80, 30, 90, 40, 50, 70}

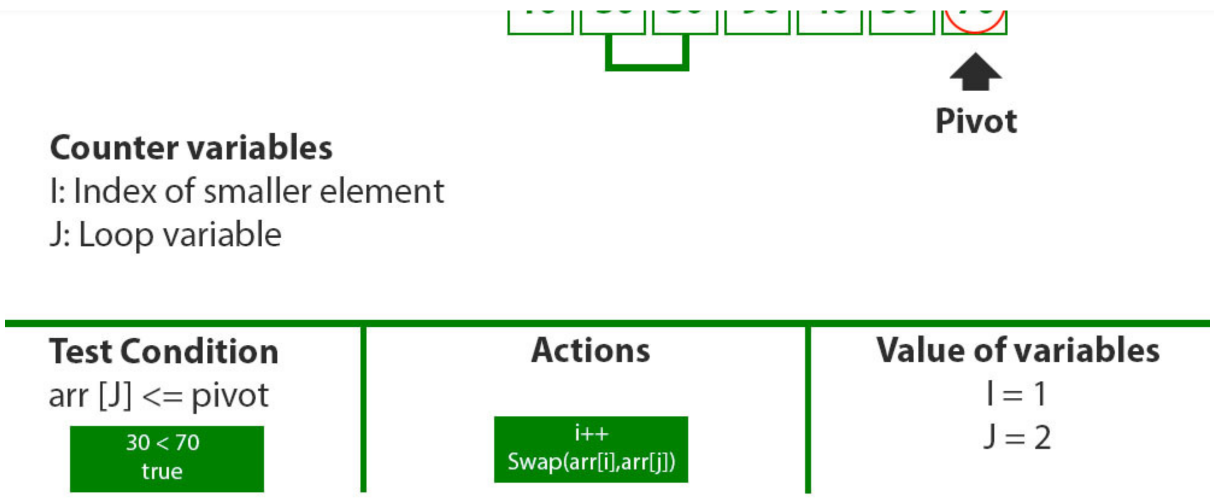
- *Indexes: 0   1   2   3   4   5   6*
- *low = 0, high = 6, pivot = arr[h] = 70*
- *Initialize index of smaller element, **i = -1***



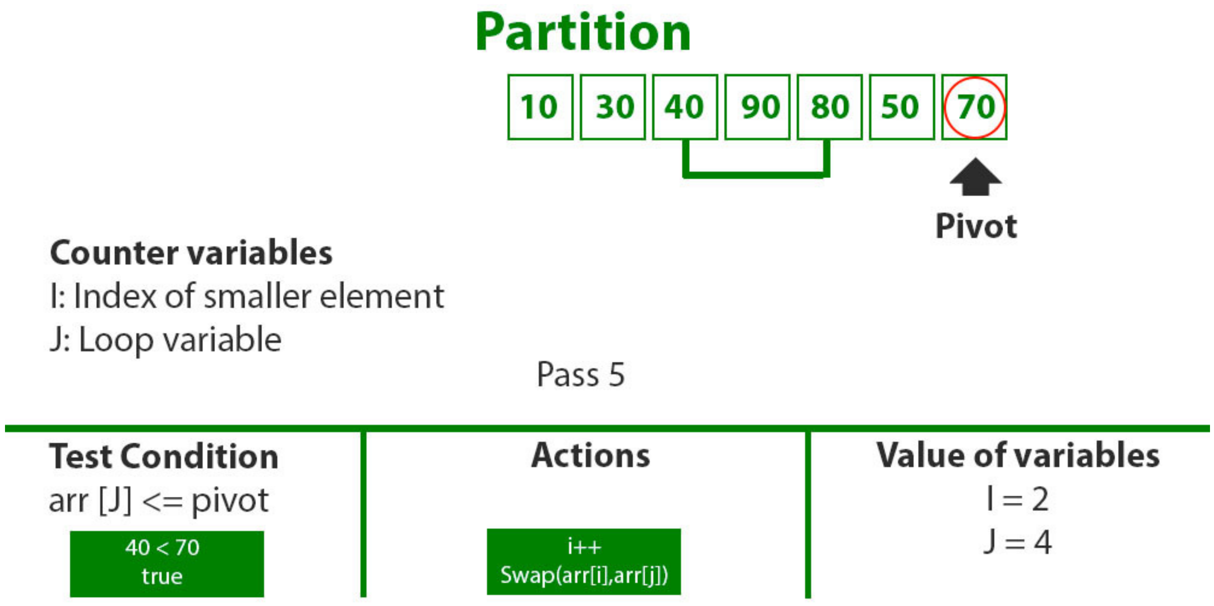
- *Traverse elements from j = low to high-1*
  - ***j = 0**: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])*
  - ***i = 0***
- *arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same*
- ***j = 1**: Since arr[j] > pivot, do nothing*



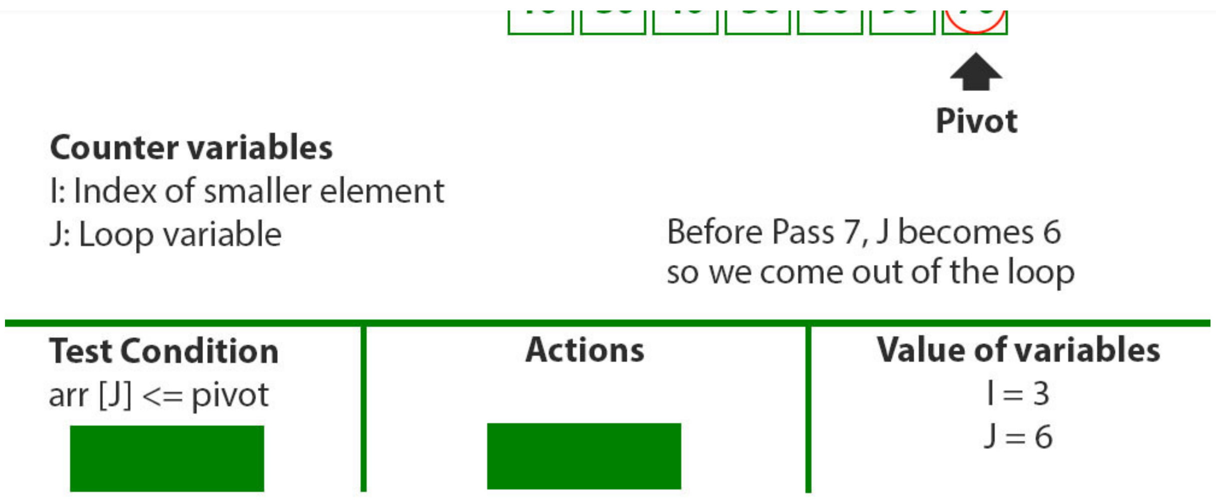
- ***j = 2** : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])*
- ***i = 1***
- *arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30*



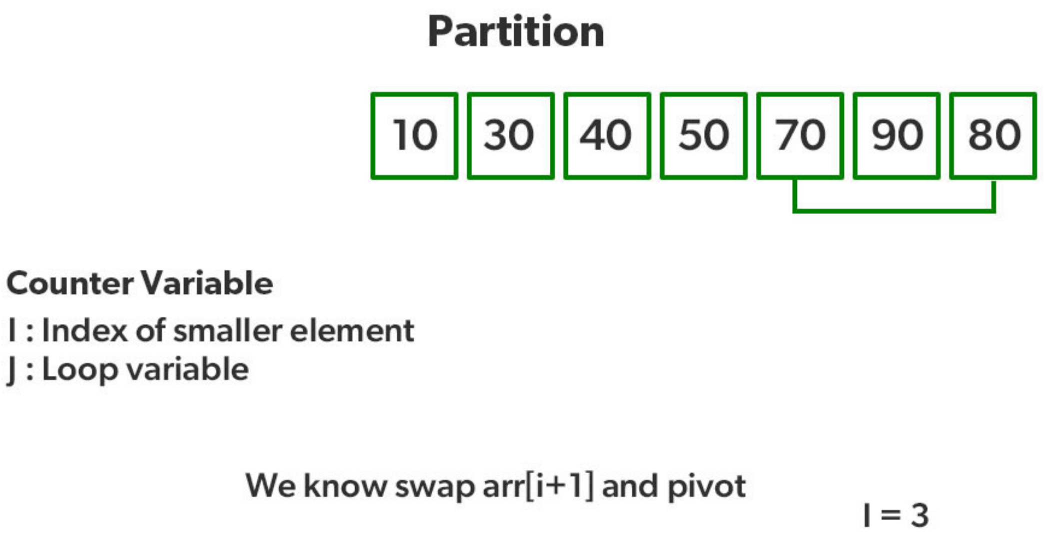
- ***j = 3*** *: Since arr[j] > pivot, do nothing // No change in i and arr[]*
- ***j = 4*** *: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])*
- ***i = 2***
- *arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped*



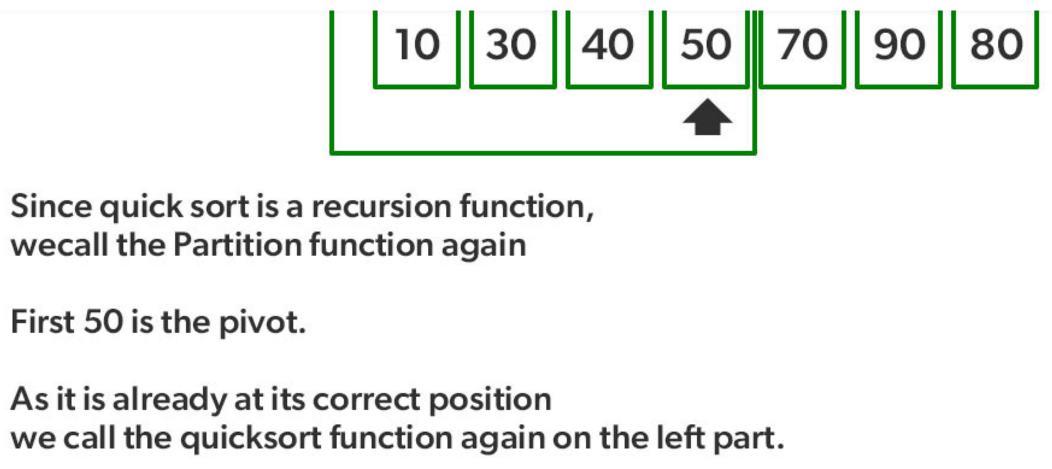
- ***j = 5*** *: Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]*
- ***i = 3***
- *arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped*



- *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.*
- *Since quick sort is a recursive function, we call the partition function again at left and right partitions*

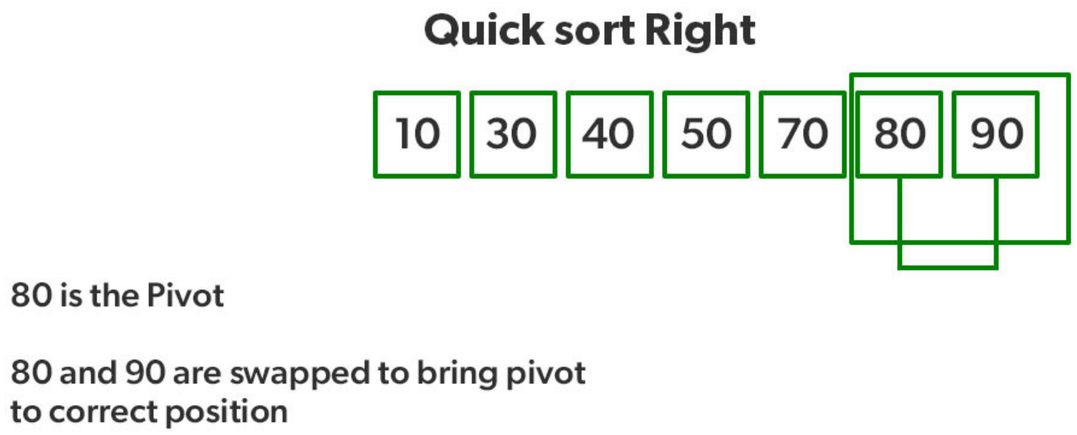


Since quick sort is a recursion function, wecall the Partition function again

First 50 is the pivot.

As it is already at its correct position we call the quicksort function again on the left part.

- *Again call function at right part and swap 80 and 90*



80 is the Pivot

80 and 90 are swapped to bring pivot to correct position

Recommended Problem

Quick Sort

Divide and Conquer Sorting +1 more VMWare Amazon +11 more

Solve Problem

Submission count: 1.2L

Implementation:

```
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 and indicates
                 // the right position of pivot found so far

    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
```

Start Your Coding Journey Now!

```
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;
}

// This code is contributed by rathbhupendra
```

Java

```
// Java implementation of QuickSort
import java.io.*;

class GFG {

    // A utility function to swap two elements
    static void swap(int[] arr, int i, int j)
    {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    /* 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 */
    static int partition(int[] arr, int low, int high)
    {
```

Start Your Coding Journey Now!

```
        for (int j = low; j <= high - 1; j++) {

            // If current element is smaller
            // than the pivot
            if (arr[j] < pivot) {

                // Increment index of
                // smaller element
                i++;
                swap(arr, i, j);
            }
        }
        swap(arr, i + 1, high);
        return (i + 1);
    }

    /* The main function that implements QuickSort
              arr[] --> Array to be sorted,
              low --> Starting index,
              high --> Ending index
     */
    static 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
    static void printArray(int[] arr, int size)
    {
        for (int i = 0; i < size; i++)
            System.out.print(arr[i] + " ");

        System.out.println();
    }

    // Driver Code
```

Start Your Coding Journey Now!

```
        printArray(arr, n);
    }
}

// This code is contributed by Ayush Choudhary
```

Python3

```python
# Python3 implementation of QuickSort


# Function to find the partition position
def partition(array, low, high):

    # Choose the rightmost element as pivot
    pivot = array[high]

    # Pointer for greater element
    i = low - 1

    # Traverse through all elements
    # compare each element with pivot
    for j in range(low, high):
        if array[j] <= pivot:
            # If element smaller than pivot is found
            # swap it with the greater element pointed by i
            i = i + 1

            # Swapping element at i with element at j
            (array[i], array[j]) = (array[j], array[i])

    # Swap the pivot element with
    # e greater element specified by i
    (array[i + 1], array[high]) = (array[high], array[i + 1])

    # Return the position from where partition is done
    return i + 1

# Function to perform quicksort


def quick_sort(array, low, high):
    if low < high:
```

Start Your Coding Journey Now!

```
        # Recursive call on the right of pivot
        quick_sort(array, pi + 1, high)


# Driver code
array = [10, 7, 8, 9, 1, 5]
quick_sort(array, 0, len(array) - 1)

print(f'Sorted array: {array}')

# This code is contributed by Adnan Aliakbar
```

C#

```
// C# implementation of QuickSort

using System;

class GFG {

    // A utility function to swap two elements
    static void swap(int[] arr, int i, int j)
    {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    /* 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 */
    static int partition(int[] arr, int low, int high)
    {

        // pivot
        int pivot = arr[high];

        // Index of smaller element and
        // indicates the right position
        // of pivot found so far
        int i = (low - 1);
```

Start Your Coding Journey Now!

```
                i++;
                swap(arr, i, j);
            }
        }
        swap(arr, i + 1, high);
        return (i + 1);
    }

    /* The main function that implements QuickSort
                arr[] --> Array to be sorted,
                low --> Starting index,
                high --> Ending index
       */
    static 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
    static void printArray(int[] arr, int size)
    {
        for (int i = 0; i < size; i++)
            Console.Write(arr[i] + " ");

        Console.WriteLine();
    }

    // Driver Code
    public static void Main()
    {
        int[] arr = { 10, 7, 8, 9, 1, 5 };
        int n = arr.Length;

        quickSort(arr, 0, n - 1);
        Console.Write("Sorted array: ");
        printArray(arr, n);
```

Start Your Coding Journey Now!

```
// Javascript implementation of QuickSort


// A utility function to swap two elements
function swap(arr, i, j) {
    let temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

/* 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 */
function partition(arr, low, high) {

    // pivot
    let pivot = arr[high];

    // Index of smaller element and
    // indicates the right position
    // of pivot found so far
    let i = (low - 1);

    for (let j = low; j <= high - 1; j++) {

        // If current element is smaller
        // than the pivot
        if (arr[j] < pivot) {

            // Increment index of
            // smaller element
            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i + 1, high);
    return (i + 1);
}

/* The main function that implements QuickSort
          arr[] --> Array to be sorted,
          low --> Starting index,
          high --> Ending index
```

Start Your Coding Journey Now!

```
        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to print an array
function printArray(arr, size) {
    for (let i = 0; i < size; i++)
        document.write(arr[i] + " ");

    document.write("<br>");
}

// Driver Code

let arr = [10, 7, 8, 9, 1, 5];
let n = arr.length;

quickSort(arr, 0, n - 1);
document.write("Sorted array: <br>");
printArray(arr, n);

// This code is contributed by Saurabh Jaiswal
```

Output

```
Sorted array:
1 5 7 8 9 10
```

Hoare's vs Lomuto Partition

Please note that the above implementation is Lomuto Partition. A more optimized implementation of QuickSort is Hoare's partition which is more efficient than Lomuto's partition scheme because it does three times less swaps on average.

How to pick any element as pivot?

Start Your Coding Journey Now!

Analysis of QuickSort

Time taken by QuickSort, in general, can be written as follows.

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

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements that are smaller than the 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 the greatest or smallest element as the pivot. If we consider the above partition strategy where the last element is always picked as a pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for the worst case.

$$T(n) = T(0) + T(n-1) + \theta(n) \text{ which is equivalent to } T(n) = T(n-1) + \theta(n)$$

The solution to the above recurrence is (n^2).

Best Case:
The best case occurs when the partition process always picks the middle element as the pivot. The following is recurrence for the best case.

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

Start Your Coding Journey Now!

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.

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

The 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.

What is 3-Way QuickSort?

arr[l..r] is divided in 3 parts:

- arr[l..i] elements less than pivot.
- arr[i+1..j-1] elements equal to pivot.
- arr[j..r] elements greater than pivot.

See this for implementation.

How to implement QuickSort for Linked Lists?

QuickSort on Singly Linked List
QuickSort on Doubly Linked List

Can we implement QuickSort Iteratively?

Yes, please refer Iterative Quick Sort.

Why Quick Sort is preferred over MergeSort 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. Allocating and de-allocating the extra space used for merge sort increases the running time of the algorithm. Comparing average complexity we find that both type of sorts have O(NlogN) average complexity but the constants differ. For arrays, merge sort loses due to the use of extra O(N) storage space.
Most practical implementations of Quick Sort use randomized version. The randomized version has expected time complexity of O(nLogn). The worst case is possible in randomized version also, but worst case doesn't occur for a particular pattern (like sorted array) and randomized Quick Sort works well in practice.
Quick Sort is also a cache friendly sorting algorithm as it has good locality of reference when used for arrays.
Quick Sort is also tail recursive, therefore tail call optimizations is done.

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.
In arrays, we can do random access as elements are continuous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i*4). Unlike arrays, we can not do random access in linked list. Quick Sort requires a lot of this kind of access. In linked list to access i'th index, we have to travel each and every node from the head to i'th node as we don't have continuous block of memory. Therefore, the overhead increases for quick sort. Merge sort accesses data sequentially and the need of random access is low.

How to optimize QuickSort so that it takes O(Log n) extra space in worst case?

Please see QuickSort Tail Call Optimization (Reducing worst case space to Log n

- Quiz on QuickSort
- Recent Articles on QuickSort
- Coding practice for sorting.

Advantages of Quick Sort:

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- It is a stable sort, meaning that if two elements have the same key, their relative order will be preserved in the sorted output.
- It has a low overhead, as it only requires a small amount of memory to function.

Disadvantages of Quick Sort:

- It has a worst-case time complexity of O(n^2), which occurs when the pivot is

Summary:

- Quick sort is a fast and efficient sorting algorithm with an average time complexity of O(n log n).
- It is a divide-and-conquer algorithm that breaks down the original problem into smaller subproblems that are easier to solve.
- It can be easily implemented in both iterative and recursive forms and it is efficient on large data sets, and can be used to sort data in-place.
- However, it also has some drawbacks such as worst case time complexity of O(n^2) which occurs when the pivot is chosen poorly.
- It is not a good choice for small data sets, it is not cache-efficient, and is sensitive to the choice of pivot.

References:
http://en.wikipedia.org/wiki/Quicksort

Other Sorting Algorithms on GeeksforGeeks/GeeksQuiz:
Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, QuickSort, Radix Sort, Counting Sort, Bucket Sort, ShellSort, Comb Sort, Pigeonhole Sort
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Previous Next

Start Your Coding Journey Now!

2. Why quicksort is better than mergesort ?
3. Dual pivot Quicksort
4. C++ Program for QuickSort
5. Java Program for QuickSort
6. Python Program for QuickSort
7. Python Program For QuickSort On Singly Linked List
8. Generic Implementation of QuickSort Algorithm in C
9. C++ Program For QuickSort On Singly Linked List
10. Merge two sorted arrays in O(1) extra space using QuickSort partition

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Medium

Easy Normal Medium Hard Expert

Start Your Coding Journey Now!

Practice Tags : Adobe, Goldman Sachs, HSBC, Qualcomm, Samsung, SAP Labs, Target Corporation, Divide and Conquer, Sorting

Improve Article

Report Issue

Start Your Coding Journey Now!

Django Tutorial	Improve an Article
HTML	Pick Topics to Write
JavaScript	Write Interview Experience
Bootstrap	Internships
ReactJS	Video Internship
NodeJS	