QuickSort

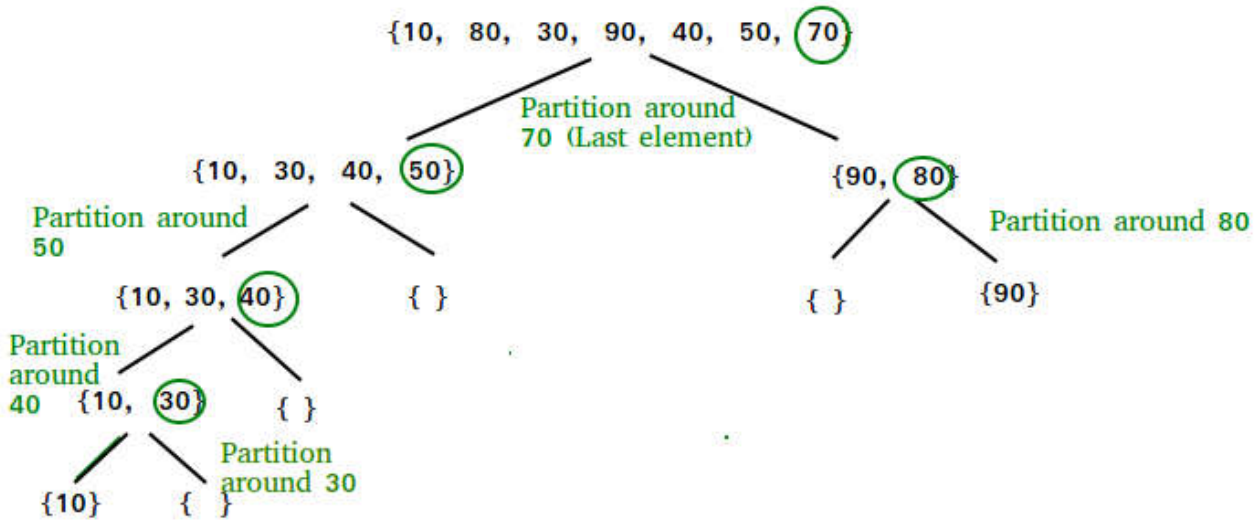
QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

QuickSort

How does QuickSort work?

The key process in **quickSort** is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater

This partition is done recursively which finally sorts the array. See the below image for a better understanding.



Choice of Pivot:

There are many different choices for picking pivots.

- 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 the middle as the pivot.

Partition Algorithm:

There can be many ways to do partition. The following pseudo-code adopts the method given in the CLRS book. The logic is simple, we start from the leftmost element and 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 Quick Sort:

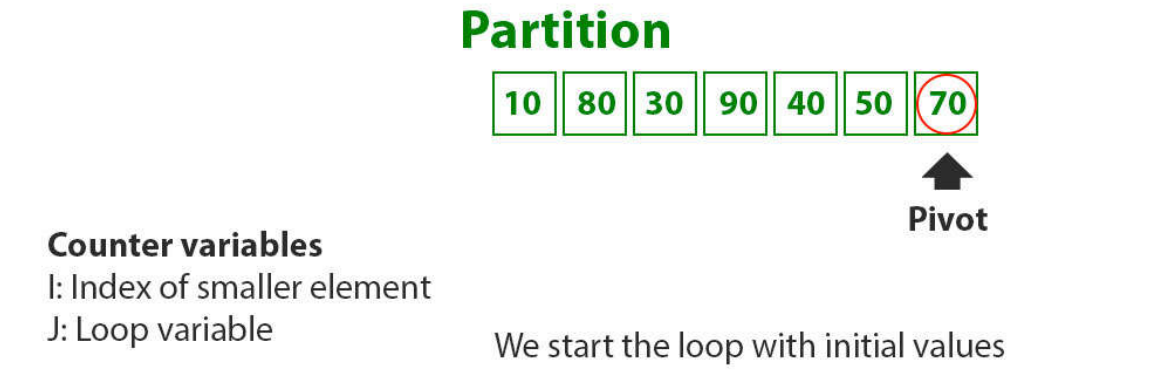
```
/* low  --> Starting index,  high  --> Ending index */
```

```
        /* 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():

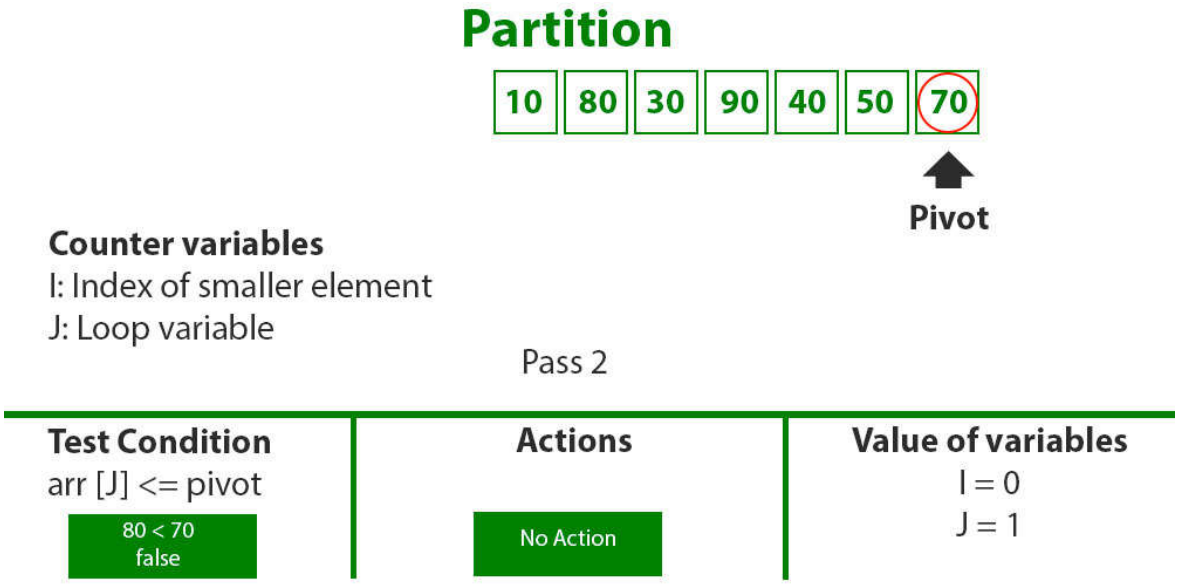
```
/* This function takes last element as pivot, places the pivot element at its
correct position in sorted array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right of pivot */

partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)
    pivot = arr[high];

    i = (low – 1)  // Index of smaller element 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)
}
```

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

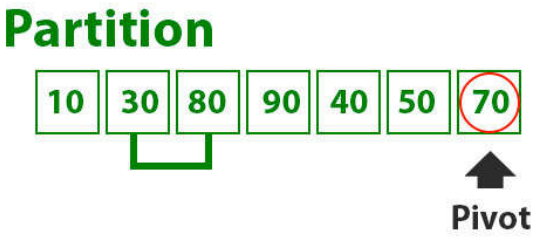


Compare pivot with first element

- *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*



Compare pivot with arr[1]

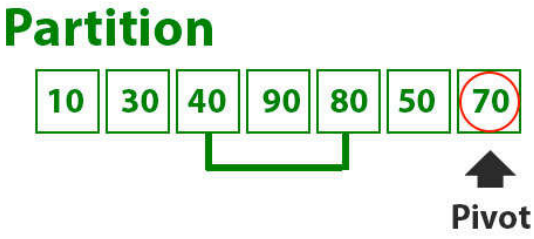


Counter variables
I: Index of smaller element
J: Loop variable

Test Condition	Actions	Value of variables
arr [J] <= pivot 30 < 70 true	i++ Swap(arr[i],arr[j])	I = 1 J = 2

Compare pivot with arr[2]

- ***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*



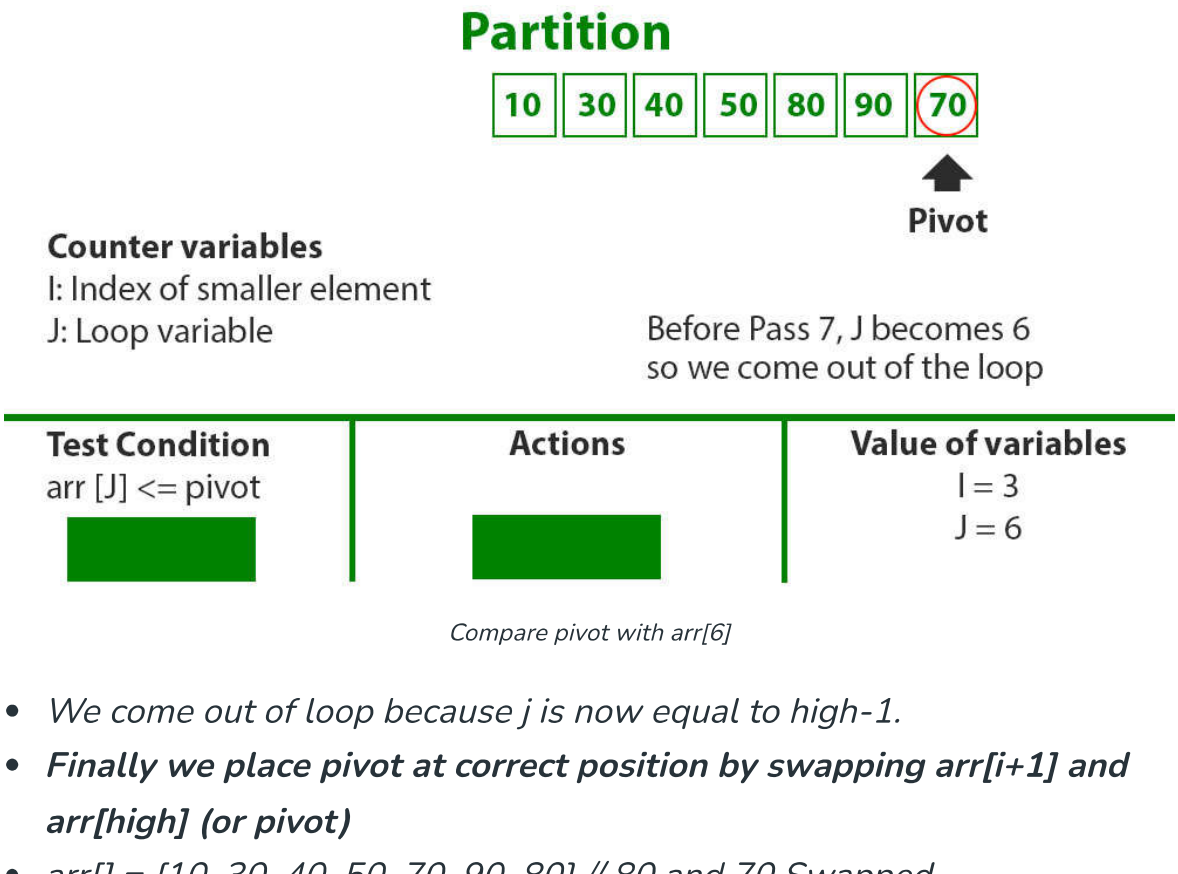
Counter variables
I: Index of smaller element
J: Loop variable

Pass 5

Test Condition	Actions	Value of variables
arr [J] <= pivot 40 < 70 true	i++ Swap(arr[i],arr[j])	I = 2 J = 4

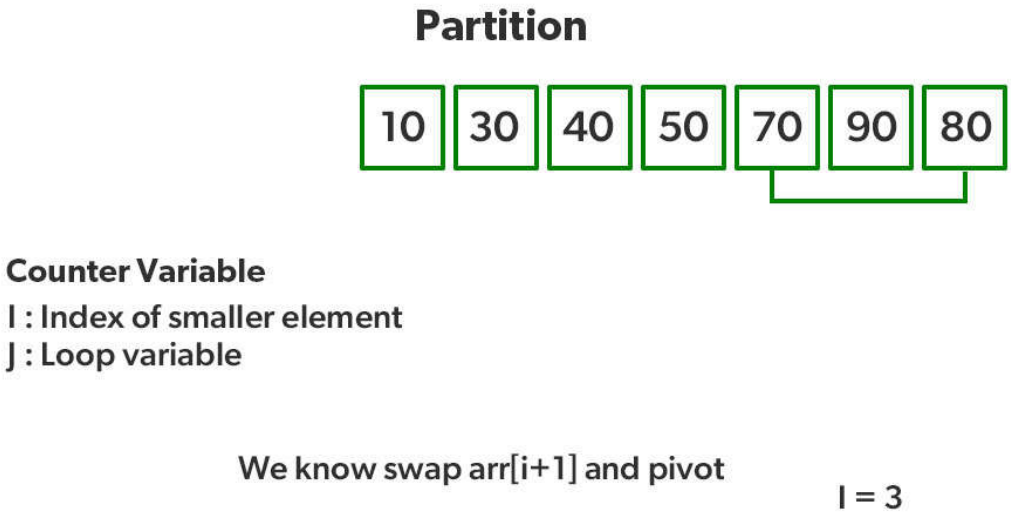
Compare pivot with arr[4]

- ***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*



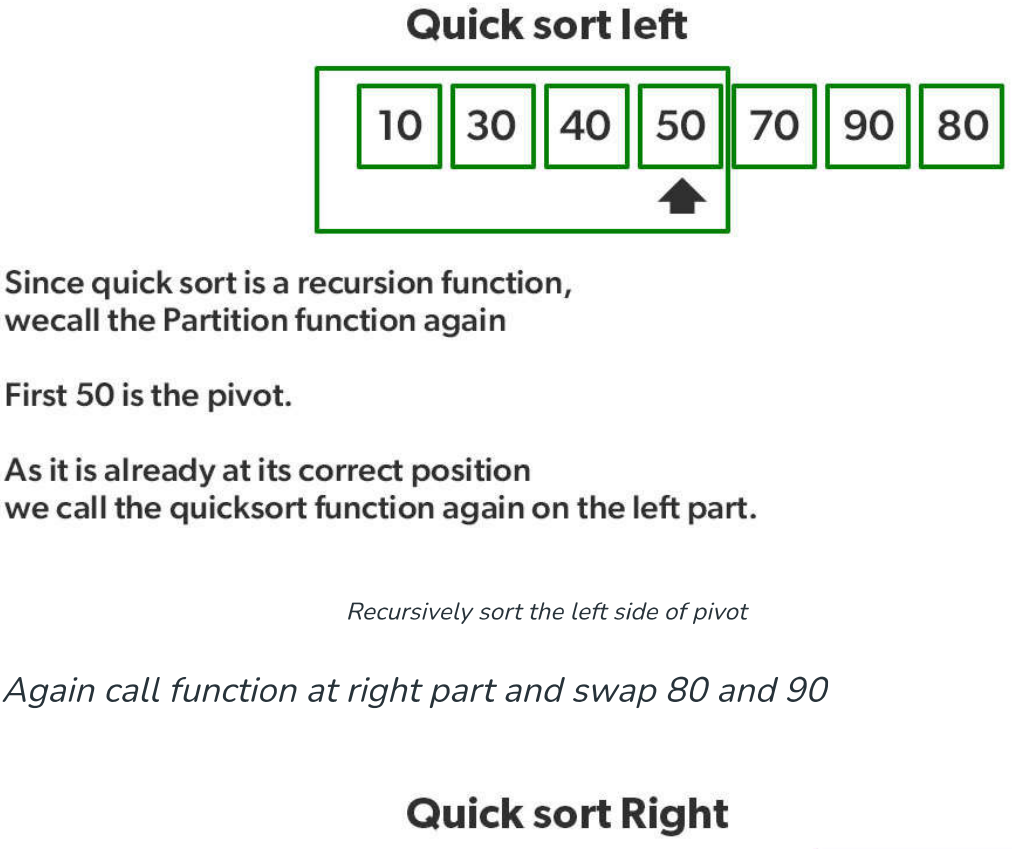
Compare pivot with arr[6]

- *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*



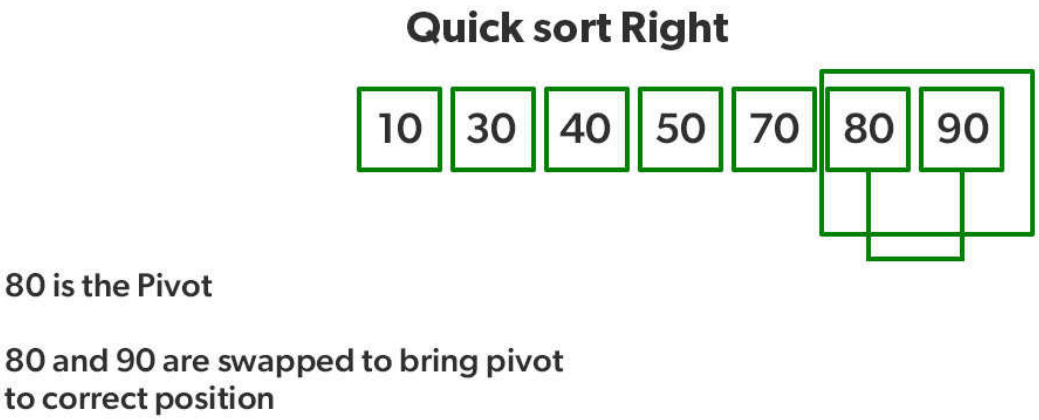
Swap arr[i+1] with pivot

- *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*



Recursively sort the left side of pivot

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



Recursively sort the right side of pivot

Program to implement QuickSort:

Follow the below steps to implement the algorithm:

- Create a recursive function (say **quicksort()**) to implement the quicksort.
- Partition the range to be sorted (initially the range is from **0 to N-1**) and return the correct position of the pivot (say **pi**).

- Return the correct position of the pivot.

- Recursively call the quicksort for the left and the right part of the **pi**.

Below is the implementation of the Quicksort:

C

```c
// C code to implement quicksort

#include <stdio.h>

// Function to swap two elements
void swap(int* a, int* b)
{
    int t = *a;
    *a = *b;
    *b = t;
}

// Partition the array using the last element as the pivot
int partition(int arr[], int low, int high)
{
    // Choosing the pivot
    int pivot = arr[high];

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

    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], &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
```

```
        // 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);
    }
}

// Driver code
int main()
{
    int arr[] = { 10, 7, 8, 9, 1, 5 };
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call
    quickSort(arr, 0, N - 1);
    printf("Sorted array: \n");
    for (int i = 0; i < N; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

C++

```
// C++ code to implement quicksort

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

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

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

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

```
            // Increment index of smaller element
            i++;
            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);
    }
}

// Driver Code
int main()
{
    int arr[] = { 10, 7, 8, 9, 1, 5 };
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function call
    quickSort(arr, 0, N - 1);
    cout << "Sorted array: " << endl;
    for (int i = 0; i < N; i++)
        cout << arr[i] << " ";
    return 0;
}
```

Java

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

```
{
    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 to left
// of pivot and all greater elements to right of pivot
static int partition(int[] arr, int low, int high)
{
    // Choosing the pivot
    int pivot = arr[high];

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

    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);
```

```
    public static void main(String[] args)
    {
        int[] arr = { 10, 7, 8, 9, 1, 5 };
        int N = arr.length;

        // Function call
        quickSort(arr, 0, N - 1);
        System.out.println("Sorted array:")
        for (int i = 0; i < N; i++)
            System.out.print(arr[i] + " ");
    }
}

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

Python3

```
# 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
    # the 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
```

```
    if low < high:

        # Find pivot element such that
        # element smaller than pivot are on the left
        # element greater than pivot are on the right
        pi = partition(array, low, high)

        # Recursive call on the left of pivot
        quicksort(array, low, pi - 1)

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


# Driver code
if __name__ == '__main__':
    array = [10, 7, 8, 9, 1, 5]
    N = len(array)

    # Function call
    quicksort(array, 0, N - 1)
    print('Sorted array:')
    for x in array:
        print(x, end=" ")

# 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 to left
    // of pivot and all greater elements to right of pivot
```

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

    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
        // and after partition index
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

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

    // Function call
    quickSort(arr, 0, N - 1);
    Console.WriteLine("Sorted array:");
    for (int i = 0; i < N; i++)
```

Cobol

```
IDENTIFICATION DIVISION.
PROGRAM-ID. QUICK-SORT.
DATA DIVISION.
WORKING-STORAGE SECTION.
01   ARRAY-SIZE          PIC 99 VALUE 20.
01   ARRAY              OCCURS 1 TO 20 TIMES
                           INDEXED BY ARRAY-INDEX.
    05   ARRAY-ELEM    PIC 99.

PROCEDURE DIVISION.
BEGIN.
    DISPLAY "Before Sorting:".
    PERFORM DISPLAY-ARRAY.
    PERFORM QUICKSORT(1, ARRAY-SIZE).
    DISPLAY "After Sorting:".
    PERFORM DISPLAY-ARRAY.
    STOP RUN.

QUICKSORT PROCEDURE.
    USING LEFT-RIGHT-INDEX.
    IF LEFT-RIGHT-INDEX(1) < LEFT-RIGHT-INDEX(2)
        PERFORM SORT-PARTITION THRU SORT-PARTITION-EXIT
            USING LEFT-RIGHT-INDEX
        PERFORM QUICKSORT THRU QUICKSORT-EXIT
            USING LEFT-RIGHT-INDEX(1), PARTITION-INDEX - 1
        PERFORM QUICKSORT THRU QUICKSORT-EXIT
            USING PARTITION-INDEX + 1, LEFT-RIGHT-INDEX(2)
    END-IF.
    EXIT.

SORT-PARTITION PROCEDURE.
    USING LEFT-RIGHT-INDEX.
    COMPUTE PARTITION-INDEX = ARRAY-ELEM(LEFT-RIGHT-INDEX(2))
                              BEFORE ARRAY-ELEM(LEFT-RIGHT-INDEX(1)).
    SET LOWER-INDEX TO LEFT-RIGHT-INDEX(1) - 1.
    SET HIGHER-INDEX TO LEFT-RIGHT-INDEX(2).
    PERFORM UNTIL LOWER-INDEX >= HIGHER-INDEX
        PERFORM UNTIL ARRAY-ELEM(LOWER-INDEX + 1) >= PARTITION-INDEX
            ADD 1 TO LOWER-INDEX
        END-PERFORM.
        PERFORM UNTIL ARRAY-ELEM(HIGHER-INDEX - 1) <= PARTITION-INDEX
            SUBTRACT 1 FROM HIGHER-INDEX
        END-PERFORM.
        IF LOWER-INDEX < HIGHER-INDEX
            COMPUTE SWAP-TEMP = ARRAY-ELEM(LOWER-INDEX + 1)
```

```
    END-PERFORM.
    COMPUTE SWAP-TEMP = ARRAY-ELEM(LEFT-RIGHT-INDEX(1))
                        BEFORE ARRAY-ELEM(HIGHER-INDEX - 1).
    MOVE SWAP-TEMP TO ARRAY-ELEM(HIGHER-INDEX - 1).
    MOVE ARRAY-ELEM(HIGHER-INDEX) TO ARRAY-ELEM(LEFT-RIGHT-INDEX(1)).
    COMPUTE PARTITION-INDEX = HIGHER-INDEX.
    EXIT.

DISPLAY-ARRAY PROCEDURE.
    PERFORM VARYING ARRAY-INDEX FROM 1 BY 1 UNTIL ARRAY-INDEX > ARRAY-SIZE
        DISPLAY ARRAY-ELEM(ARRAY-INDEX)
    END-PERFORM.
    EXIT.
```

Output

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

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.

Worst Case:

The worst case occurs when the partition process always picks the first or last 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 the recurrence for the worst case.

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

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

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.

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

The solution for the above recurrence is O(N * logN). It can be solved using case 2 of the Master Theorem.

Average Case:

To do an average case analysis, we need to consider all possible permutations of the array and calculate the time taken by every permutation which doesn't look easy.

We can get an idea of an average case by considering the case when partition puts O(N/9) elements in one set and O(9N/10) elements in the other set. Following is the recurrence for this case.

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

The solution of the above recurrence is also O(N * logN):

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

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- 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 chosen poorly.
- It is not a good choice for small data sets.
- It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

Some Frequently asked Questions (FAQs) on QuickSort:

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?

With one minor change to the above code, we can pick any element as pivot. For example, to make the first element as pivot, we can simply swap the first and last elements and then use the same code. Same thing can be done to pick any random element as a pivot

Is QuickSort stable?

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

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?

In simple QuickSort algorithm, we select an element as pivot, partition the array around pivot and recur for subarrays on left and right of pivot.
Consider an array which has many redundant elements. For example, {1, 4, 2, 4, 2, 4, 1, 2, 4, 1, 2, 2, 2, 2, 4, 1, 4, 4, 4}. If 4 is picked as pivot in Simple QuickSort, we fix only one 4 and recursively process remaining occurrences. In 3 Way QuickSort, an array 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

- Most practical implementations of Quick Sort use randomized versions. The randomized version has an expected time complexity of O(N logN). The worst case is possible in the randomized version also, but the 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 a good locality of reference when used for arrays.
- Quick Sort is also tail recursive, therefore tail call optimizations are done.

Why MergeSort is preferred over QuickSort for Linked Lists ?

- In the case of linked lists, the case is different mainly due to the difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory.
- Unlike arrays, in linked lists, 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.
- Unlike arrays, we can not do random access in linked lists. Quick Sort requires a lot of this kind of access. In a linked list to access the i^{th} index, we have to travel each and every node from the head to i^{th} node as we don't have a continuous block of memory. Therefore, the overhead increases for quicksort. Merge sort accesses data sequentially and the need for random access is low.

How to optimize QuickSort so that it takes O(log N) extra space in the worst case?

As the recursion call is performed at the end of the recursive function, we can use the concept of tail recursion to optimize the space taken by Quicksort. Please refer to QuickSort Tail Call Optimization (Reducing worst case space to log N).

Conclusion:

To sum up, it can be said that Quicksort is a fast and efficient sorting algorithm with an average time complexity of O(N logN). It is a divide-and-conquer algorithm that breaks down the original problem into smaller subproblems that

$O(N^2)$ which occurs when the pivot is chosen poorly. The performance of quicksort is sensitive to the choice of the pivot.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Last Updated : 20 Apr, 2023

701

Recommended Problem

Quick Sort

Divide and Conquer Sorting +1 more

VMWare Amazon +11 more

Solve Problem

Submission count: 1.2L

Similar Reads

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

1. Learn Data Structures with Javascript | DSA Tutorial
2. Introduction to Max-Heap – Data Structure and Algorithm Tutorials
3. Introduction to Set – Data Structure and Algorithm Tutorials
4. Introduction to Map – Data Structure and Algorithm Tutorials
5. What is Dijkstra's Algorithm? | Introduction to Dijkstra's Shortest Path Algorithm

Previous Next

Article Contributed By :

GeeksforGeeks

Vote for difficulty

Current difficulty : Medium

Easy | Normal | Medium | Hard | Expert

Improved By : Palak Jain 5, lakshaygupta2807, UditChaudhary, rathbhupendra, ays14, devi_johns, dhavaldhavalb588, code_ayush, gfgking, adnanaliakbar99, sreenivasulureddyk19, animeshdey, saurabhaniket2903, shreyasnaphad, kashishkumar2, harendrakumar123, yatinpandit, dakshdictef, saiyam99b1mq, manishk4514

Article Tags : Adobe, Goldman Sachs, HSBC, Qualcomm, Quick Sort, Samsung, SAP Labs, Target Corporation, Divide and Conquer, DSA, Sorting

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

A-143, 9th Floor, Sovereign Corporate Tower,
Sector-136, Noida, Uttar Pradesh - 201305

feedback@geeksforgeeks.org

Computer Science

GATE CS Notes

Operating Systems

Computer Network

Database Management System

Software Engineering

Digital Logic Design

Engineering Maths

Data Science & ML

Data Science With Python

Data Science For Beginner

Machine Learning Tutorial

Maths For Machine Learning

Pandas Tutorial

NumPy Tutorial

NLP Tutorial

Interview Corner

Company Preparation

Preparation for SDE

Company Interview Corner

Experienced Interview

Internship Interview

Competitive Programming

Aptitude

Python

Python Tutorial

Python Programming Examples

Django Tutorial

Python Projects

Python Tkinter

OpenCV Python Tutorial

GfG School

CBSE Notes for Class 8

CBSE Notes for Class 9

CBSE Notes for Class 10

CBSE Notes for Class 11

CBSE Notes for Class 12

English Grammar

UPSC/SSC/BANKING

SSC CGL Syllabus

SBI PO Syllabus

IBPS PO Syllabus

UPSC Ethics Notes

UPSC Economics Notes

UPSC History Notes