

Enrol in top rated Coding Courses and get assured Scholarship | Apply Now FREE

takeUforward

~ Strive for Excellence



January 4, 2022 ▪ Arrays / Data Structure

Quick Sort Algorithm

Problem Statement: Given an array of n integers, sort the array using the **Quicksort** method.

Examples:

Example 1:

Input: $N = 5$, $Arr[] = \{4, 1, 7, 9, 3\}$

Output: 1 3 4 7 9

Explanation: After sorting the array becomes 1, 3, 4, 7, 9

Example 2:

Input: $N = 8$, $Arr[] = \{4, 6, 2, 5, 7, 9, 1, 3\}$

Output: 1 2 3 4 5 6 7 9

Explanation: After sorting the array becomes 1, 3, 4, 7, 9

Subscribe

I want to receive latest posts and interview tips

Name*

Email*

Join takeUforward

Search

Solution

Disclaimer: Don't jump directly to the solution, try it out yourself first.

[Problem Link](#).

Intuition:

Quick Sort is a divide-and-conquer algorithm like [the Merge Sort](#). But unlike Merge sort, this algorithm does not use any extra array for sorting(though it uses an auxiliary stack space). So, from that perspective, Quick sort is slightly better than Merge sort.

This algorithm is basically a repetition of two simple steps that are the following:

- Pick a pivot and place it in its correct place in the sorted array.
- Shift smaller elements(i.e. Smaller than the pivot) on the left of the pivot and larger ones to the right.

Now, let's discuss the steps in detail considering the array {4,6,2,5,7,9,1,3}:

Step 1: The first thing is to choose the pivot.

A pivot is basically a chosen element of the given array. The element or the pivot can be chosen by our choice. So, in an array a pivot can be any of the following:

- The first element of the array

Recent Posts

Pattern-1:

Rectangular Star Pattern

Time and Space

Complexity –

Strivers A2Z DSA Course

Hashing | Maps |

Time Complexity |

Collisions | Division

Rule of Hashing |

Strivers A2Z DSA

Course

Print 1 to N using

Recursion

Print N to 1 using

Recursion

Accolite Digital

Amazon Arcesium

Bank of America Barclays

BFS Binary Search

Binary Search Tree

Commvault CPP DE Shaw

DFS DSA Self

Paced google

HackerEarth Hashing

- The last element of the array
- Median of array
- Any Random element of the array

After choosing the pivot(i.e. the element), we should place it in its correct position(*i.e. The place it should be after the array gets sorted*) in the array. For example, if the given array is {4,6,2,5,7,9,1,3}, the correct position of 4 will be the 4th position.

Note: *Here in this tutorial, we have chosen the first element as our pivot. You can choose any element as per your choice.*

Step 2: In step 2, we will shift the smaller elements(i.e. Smaller than the pivot) to the left of the pivot and the larger ones to the right of the pivot. In the example, if the chosen pivot is 4, after performing step 2 the array will look like: {3, 2, 1, 4, 6, 5, 7, 9}.

From the explanation, we can see that after completing the steps, pivot 4 is in its correct position with the left and right subarray unsorted. Now **we will apply these two steps on the left subarray and the right subarray recursively**. And we will continue this process until the size of the unsorted part becomes 1(as an array with a single element is always sorted).

So, from the above intuition, we can get a clear idea that we are going to use recursion

[infosys](#) [inorder](#) [Java](#) [Juspay](#)

[Kreeti Technologies](#) [Morgan](#)

[Stanley](#) [Newfold Digital](#)

[Oracle](#) [post order](#) [queue](#)

[recursion](#) [Samsung](#) [SDE](#)

[Core Sheet](#) [SDE](#)

[Sheet](#) [Searching set-](#)

[bits](#) [sorting](#)

[Strivers](#)

[A2ZDSA](#)

[Course](#) [sub-array](#)

[subarray](#) [Swiggy](#)

[takeuforward](#) [TCQ](#) [NINJA](#) [TCS](#)

[TCS](#) [CODEVITA](#) [TCS Ninja](#)

[TCS](#) [NQT](#)

[VMware](#) [XOR](#)

in this algorithm.

To summarize, the main intention of this process is to place the pivot, after each recursion call, at its final position, where the pivot should be in the final sorted array.

Approach:

Now, let's understand how we are going to implement the logic of the above steps. In the intuition, we have seen that the given array should be broken down into subarrays. But while implementing, we are not going to break down and create any new arrays. Instead, we will specify the range of the subarrays using two indices or pointers(i.e. **low** pointer and **high** pointer) each time while breaking down the array.

The algorithm steps are the following for the **quickSort()** function:

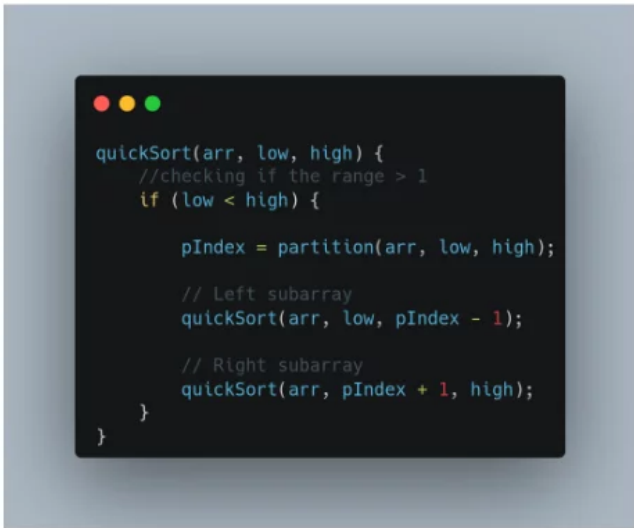
1. Initially, the **low** points to the first index and the **high** points to the last index(as the range is n i.e. the size of the array).
2. After that, we will get the index(*where the pivot should be placed after sorting*) while shifting the smaller elements on the left and the larger ones on the right using a `partition()` function discussed later.

Now, this index can be called the **partition index** as it creates a partition between the left and the right unsorted subarrays.

3. After placing the pivot in the partition index(within the partition() function specified), we need to call the function quickSort() for the left and the right subarray recursively. So, **the range of the left subarray will be [low to (partition index – 1)]** and **the range of the right subarray will be [(partition index + 1) to high]**.
4. This is how the recursion will continue until the range becomes 1.

Pseudocode:

So, the pseudocode will look like the following:



```
quickSort(arr, low, high) {  
    //checking if the range > 1  
    if (low < high) {  
        pIndex = partition(arr, low, high);  
  
        // Left subarray  
        quickSort(arr, low, pIndex - 1);  
  
        // Right subarray  
        quickSort(arr, pIndex + 1, high);  
    }  
}
```

Now, let's understand how to implement the partition() function to get the partition index.

1. Inside the function, we will first select the pivot(*i.e.* arr[low] *in our case*).
2. Now, we will again take two-pointers i and j. The i pointer points to low and the j

points to high.

3. Now, the pointer i will move forward and find the first element that is greater than the pivot. Similarly, the pointer j will move backward and find the first element that is smaller than the pivot.

Here, we need to add some checks like $i \leq \text{high}-1$ and $j \geq \text{low}+1$. Because it might happen that i is standing at high and trying to proceed or j is standing at low and trying to exceed.

4. Once we find such elements i.e. $\text{arr}[i] > \text{pivot}$ and $\text{arr}[j] < \text{pivot}$, and $i < j$, we will swap $\text{arr}[i]$ and $\text{arr}[j]$.
5. We will continue step 3 and step 4, until j becomes smaller than i .
6. Finally, we will swap the pivot element(i.e. $\text{arr}[\text{low}]$) with $\text{arr}[j]$ and will return the index j i.e. the partition index.

Pseudocode:

So, the pseudocode will look like the following:

```

int partition(arr, low, high) {
    pivot = arr[low];
    i = low;
    j = high;
    while (i < j) {
        //check 1:
        while (arr[i] <= pivot && i <= high - 1) {
            i++;
        }

        //check 2:
        while (arr[j] > pivot && j >= low + 1) {
            j--;
        }

        if (i < j) swap(arr[i], arr[j]);
    }

    swap(arr[low], arr[j]);
    return j;
}

```

Note: In the function, we have kept the elements equal to the pivot on the left side. If you choose to place them on the right, check 1 will be $arr[i] < pivot$ and check 2 will be $arr[j] >= pivot$.

Note: If you wish to see the dry run, refer to the video attached below.

Code:

C++ Code

```

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

int partition(vector<int> &arr, int low,
             int pivot = arr[low],
             int i = low,
             int j = high;

while (i < j) {
    while (arr[i] <= pivot && i <= h
        i++;
    }

    while (arr[j] > pivot && j >= lc

```

```

        j--;
    }
    if (i < j) swap(arr[i], arr[j]);
}
swap(arr[low], arr[j]);
return j;
}

void qs(vector<int> &arr, int low, int high) {
    if (low < high) {
        int pIndex = partition(arr, low, high);
        qs(arr, low, pIndex - 1);
        qs(arr, pIndex + 1, high);
    }
}

vector<int> quickSort(vector<int> arr) {
    qs(arr, 0, arr.size() - 1);
    return arr;
}

int main()
{
    vector<int> arr = {4, 6, 2, 5, 7, 9, 1, 3};
    int n = arr.size();
    cout << "Before Using quick Sort: " << endl;
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }

    arr = quickSort(arr);
    cout << "After Using quick sort: " << endl;
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    cout << "\n";
    return 0;
}

```

Output:

Before Using quick Sort:

4 6 2 5 7 9 1 3

After Using quick sort:

1 2 3 4 5 6 7 9

Time Complexity: $O(N \cdot \log N)$, where N = size of the array.

Reason: At each step, we divide the whole array, for that $\log N$ and n steps are taken for the partition() function, so overall time complexity will be $N \cdot \log N$.

The following recurrence relation can be written for Quick sort :

$$F(n) = F(k) + F(n-1-k)$$

Here k is the number of elements smaller or equal to the pivot and $n-1-k$ denotes elements greater than the pivot.

There can be 2 cases :

Worst Case – This case occurs when the pivot is the greatest or smallest element of the array. If the partition is done and the last element is the pivot, then the worst case would be either in the increasing order of the array or in the decreasing order of the array.

Recurrence:

$$F(n) = F(0) + F(n-1) \text{ or } F(n) = F(n-1) + F(0)$$

Worst Case Time complexity: $O(n^2)$

Best Case – This case occurs when the pivot is the middle element or near to middle element of the array.

Recurrence :

$$F(n) = 2F(n/2)$$

Time Complexity for the best and average

case: $O(N \cdot \log N)$

Space Complexity: $O(1) + O(N)$ auxiliary stack space.

Java Code

```
import java.util.*;

class Solution {
    static int partition(List<Integer> arr, int low, int high) {
        int pivot = arr.get(low);
        int i = low;
        int j = high;

        while (i < j) {
            while (arr.get(i) <= pivot && i < j) i++;
            while (arr.get(j) > pivot && j > i) j--;
            if (i < j) {
                int temp = arr.get(i);
                arr.set(i, arr.get(j));
                arr.set(j, temp);
            }
        }
        int temp = arr.get(low);
        arr.set(low, arr.get(j));
        arr.set(j, temp);
        return j;
    }

    static void qs(List<Integer> arr, int low, int high) {
        if (low < high) {
            int pIndex = partition(arr, low, high);
            qs(arr, low, pIndex - 1);
            qs(arr, pIndex + 1, high);
        }
    }
}
```

```

        qs(arr, pIndex + 1, high);
    }
}

public static List<Integer> quickSort(List<Integer> arr) {
    // Write your code here.
    qs(arr, 0, arr.size() - 1);
    return arr;
}

}

public class tUf {
    public static void main(String args[]) {
        List<Integer> arr = new ArrayList<>();
        arr = Arrays.asList(new Integer[] {4, 6, 2, 5, 7, 9, 1, 3});
        int n = arr.size();
        System.out.println("Before Using quick Sort:");
        for (int i = 0; i < n; i++) {
            System.out.print(arr.get(i) + " ");
        }
        System.out.println();
        arr = Solution.quickSort(arr);
        System.out.println("After Using quick Sort:");
        for (int i = 0; i < n; i++) {
            System.out.print(arr.get(i) + " ");
        }
        System.out.println();
    }
}

```



Output:

Before Using quick Sort:

4 6 2 5 7 9 1 3

After Using quick sort:

1 2 3 4 5 6 7 9

Time Complexity: $O(N \cdot \log N)$, where N = size of the array.

Reason: At each step, we divide the whole array, for that $\log N$ and n steps are taken for

partition() function, so overall time complexity will be $N \cdot \log N$.

The following recurrence relation can be written for Quick sort :

$$F(n) = F(k) + F(n-1-k)$$

Here k is the number of elements smaller or equal to the pivot and $n-1-k$ denotes elements greater than the pivot.

There can be 2 cases :

Worst Case – This case occurs when the pivot is the greatest or smallest element of the array. If the partition is done and the last element is the pivot, then the worst case would be either in the increasing order of the array or in the decreasing order of the array.

Recurrence:

$$F(n) = F(0) + F(n-1) \text{ or } F(n) = F(n-1) + F(0)$$

Worst Case Time complexity: $O(n^2)$

Best Case – This case occurs when the pivot is the middle element or near to middle element of the array.

Recurrence :

$$F(n) = 2F(n/2)$$

Time Complexity for the best and average case: $O(N \cdot \log N)$

Space Complexity: $O(1) + O(N)$ auxiliary stack space.

Special thanks to [Shreyas Vishwakarma](#) and [KRITIDIPTA GHOSH](#) for contributing to this article on takeUforward. If you also wish to share your knowledge with the takeUforward fam, [please check out this article](#)

« Previous Post

Find all repeating elements in an array

Next Post »

Reverse a String

Load Comments

Copyright © 2022 takeuforward | All rights reserved