Courses Practice Roadmap Pro

Algorithms and Data Structures for Beginners

13 / 35

12 - Quick Sort

16:43

Mark Lesson Complete

View Code

Prev

Next

11 Merge Sort

Suggested Problems

Status	Star	Problem	Difficulty	Video Solution	Code
		Sort An Array	Medium		
		Kth Largest Element In An Array	Medium		C++

Quick Sort

The idea behind quicksort is to pick an index, which is called the `pivot`, and partition the array such that every value to the left is less than or equal to the `pivot` and every value to the right is greater than the pivot.

Picking a pivot value

Generally, there are several tested and tried options when it comes to picking a pivot:

- Pick the first index

- Pick the last index
- Pick the median (pick the first, last and the middle value and find their median and perform the split at the median)
- Pick a random pivot

You may be asking which pivot to pick? This is dependent on the data itself, both the size and the initial order. For coding interviews we can keep things simple, so in this chapter we will use the last index as our pivot.

Implementation

We will pick a `pivot` if we haven't already hit the base case which is array of size `1` and pick a `left` pointer, which will point to the left-most element of the current subarray to begin with. Then, we will iterate through our array and if we find an element smaller than our `pivot` element, we will swap the current element with the element at our `left` pointer and increment the `left` pointer.

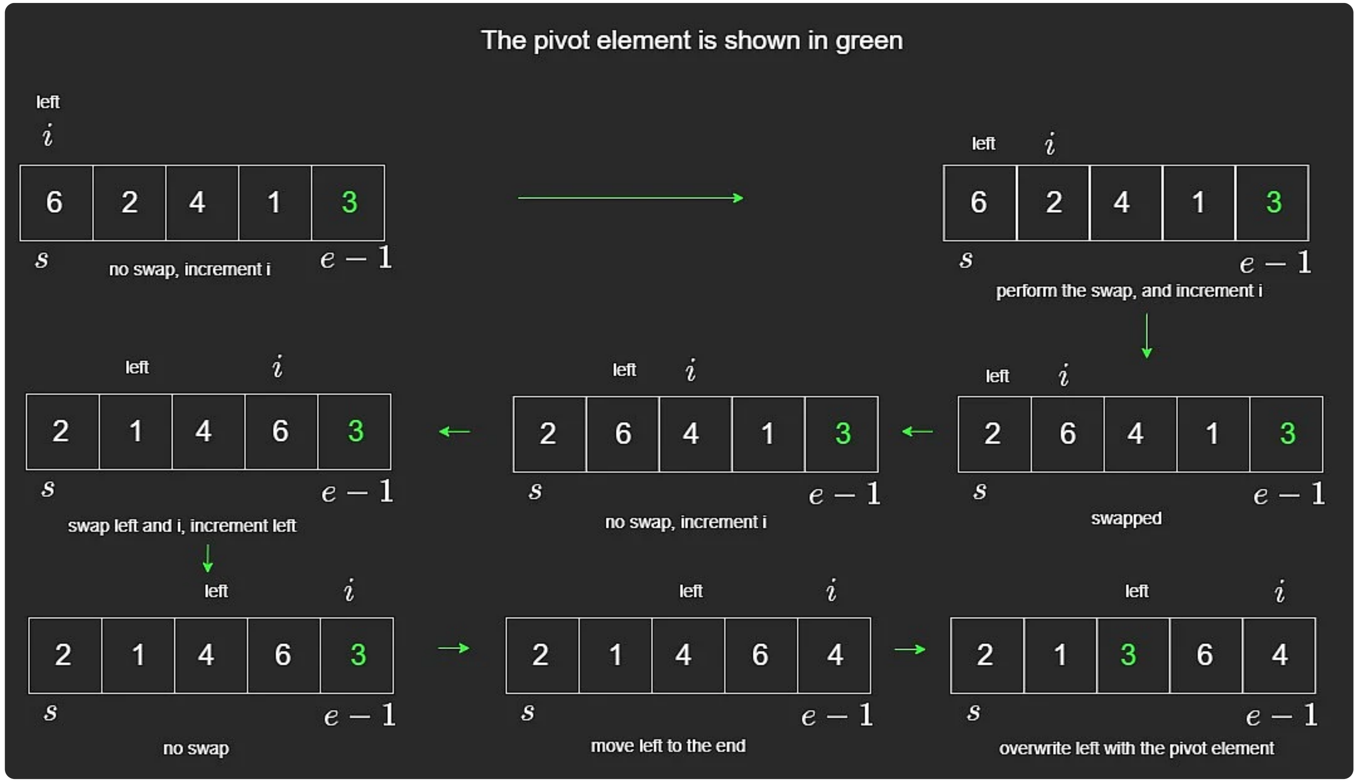
Once this condition terminates, we will bring the `left` element to the end of the array and bring the pivot element to the `left` position. This makes sense because once we have exhausted all the elements that are smaller than the `pivot` element, and put them to the left of the pivot, `left` will now be at the first element that is greater than the pivot. And, every element after it will also be greater than the pivot. This is why we move `left` element to the end and then bring the pivot element to the `left` .

This results in all the elements less than or equal to the `pivot` to be on the left and the rest on the right.

> *There is also a variation of Quicksort called randomized Quicksort, where the pivot is picked randomly, or the array is shuffled if you like, and this reduces the chance of hitting the worst case. This results in $O(n \ log \ n)$ in the worst case instead of the typical implementation where it is $O(n^2)$. This* ***Stackoverflow*** *post provides an interesting explanation behind it all.*

Let's take the array `[6,2,4,1,3]` to sort.

Performing a partition



As seen above, in the resulting array, we will have sorted the array such that all elements to the left are smaller than the `pivot` with the rest being on the right.

> *We are not going to visualize all of the steps but this process will be run recursively until we hit the base-case. It is important to notice that we are not allocating any new memory for these partitions. We are just moving around pointers to work on a smaller part of the original array each time until we end up with a sorted array.*

The pseudocode for this looks like the following.

```
fn quickSort(arr, s, e):
    if e - s + 1 <= 1:
        return

    pivot = arr[e] // End of the array
    left = s // Start of the subarray

    // Partition: All elements in the end of the array
    for i = s until e:
        if arr[i] < pivot:
            tmp = arr[left]
            arr[left] = arr[i]
            arr[i] = tmp
            left += 1

    // Move pivot in-between left & right sides
    arr[e] = arr[left]
    arr[left] = pivot

    // Quick sort left side
    quickSort(arr, s, left - 1)

    // Quick sort right side
    quickSort(arr, left + 1, e)
```

```
    return arr
```

Time Complexity

The analysis of quicksort is similar to merge sort. It also turns out to be $O(n \; log \; n)$ except only in the best case. The best case is that we pick a `pivot` such that we can always perform the partition in the middle. If the array is perfectly partitioned in the middle every single time and the pivot is the median, it is possible to keep getting $O(nlogn)$ as the ultimate result.

Picking a pivot where the `pivot` element is the smallest or the largest element will result in the worst case performance of $O(n^2)$. This is because in the worst case, picking the highest or the lowest element in the list would result in the worst pivot and picking the worst pivot each time would mean you have n groups to iterate through, hence the $O(n^2)$ complexity.

Stability

Quicksort is not a stable algorithm because it exchanges non-adjacent elements.

Take the array `[7,3,7,4,5]` where we have two 7s, one at the 0th index and one at the 2nd index. In this case, if our pivot is the 4th and the last index, we will end up with `[7,4,7,7,5]` where the 7 from the 0th index is now at 3rd index, which means that the order has been reversed.