Project Final Report

On

Quicksort Algorithm Implementation In Python

Presented By:

Serial Number	Name	ID
01	S. M. Sanaul Haque	20212020010
02	Farjana Yeasmin Mohona	20212013010
03	S. M. Rokibul Islam	20212024010

Presented To:

Md. Shymon Islam

Lecturer

Department of CSE

North Western University, Khulna.

Table Of Contents

Acknowledgement

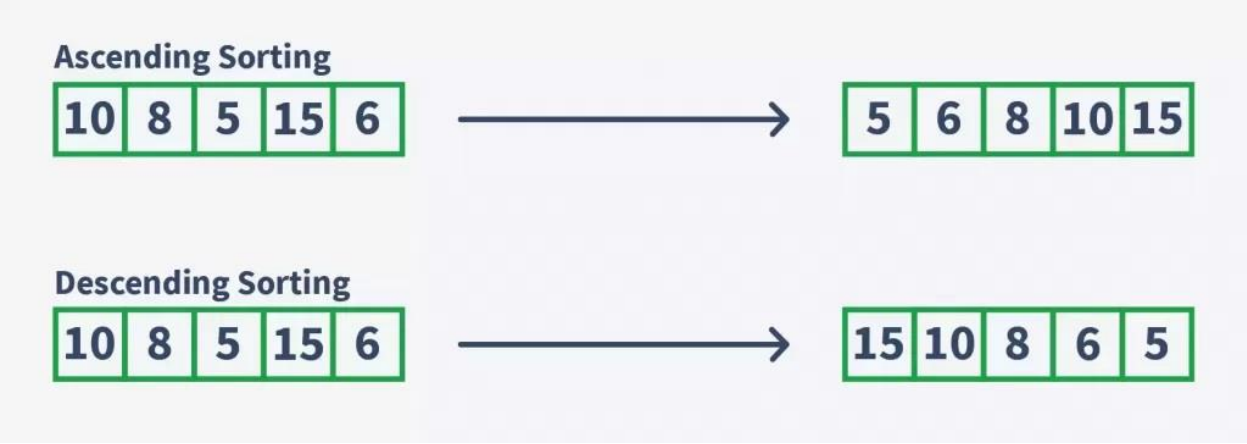
This project has been done as part of our course for the **CSE-2204**. I would like to thank my teacher, **Md. Shymon Islam** for his advice and inputs on the project. Many thanks to my group member and friends as well, which spent countless hours to listen and provide feedbacks.

Overview

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.

Consider a list with the elements – 10, 8, 5, 15, 6 in it. We have to arrange these elements in both ascending and descending order.

- How are we going to perform this arrangement? Simple answer – by using a **Sorting algorithm**. A sorting algorithm is a method of reorganizing the elements in a meaningful order. In both the above-given examples, we have implemented a sorting algorithm to get the desired results.



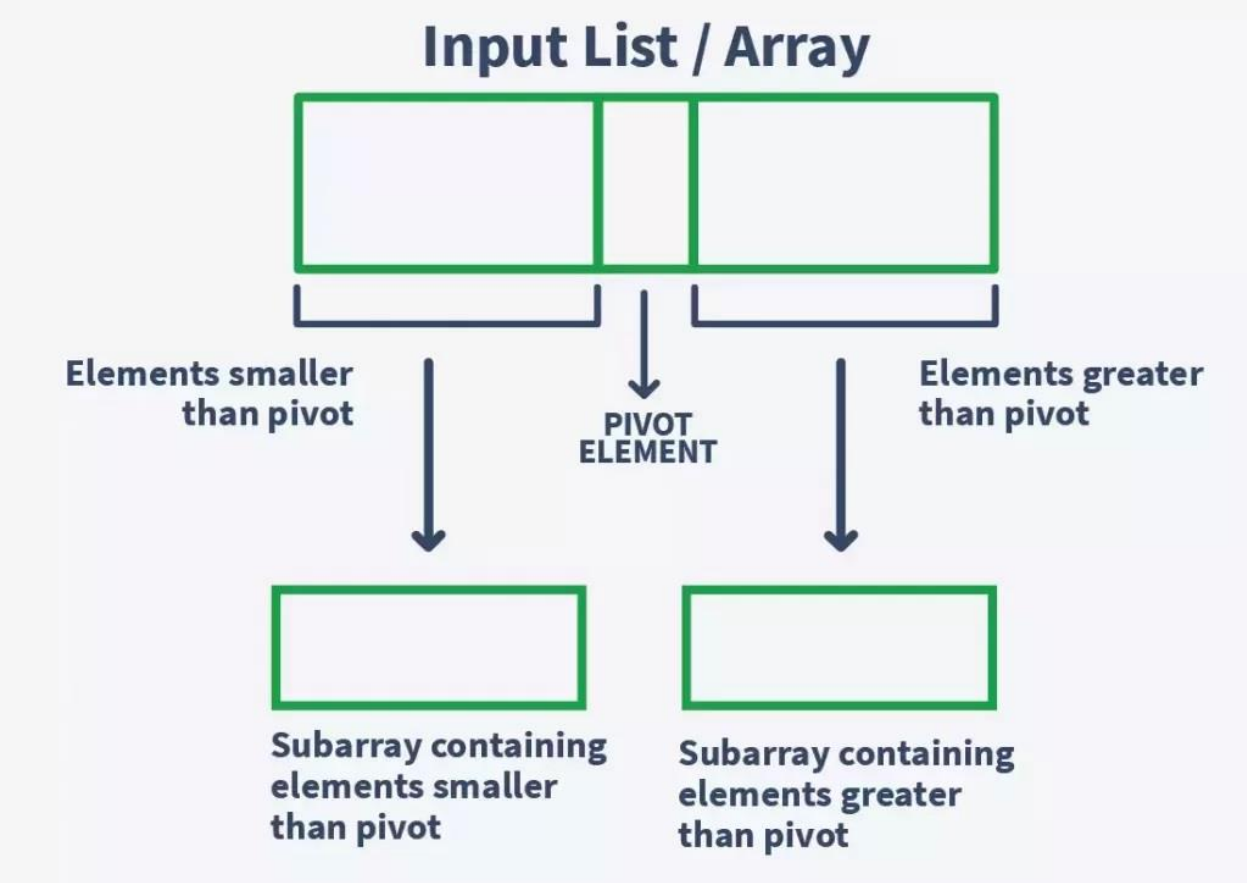
There are multiple algorithms that can be used for sorting. And, here, we will discuss Quick sort.

What is Quick Sort?

Quick sort, also known as partition-exchange sort, is an in-place sorting algorithm. It is a divide-and-conquer algorithm that works on the idea of selecting a pivot element and dividing the array into two subarrays around that pivot.

In quick sort, after selecting the pivot element, the array is split into two subarrays. One subarray contains the elements smaller than the pivot element, and the other subarray contains the elements greater than than the pivot element.

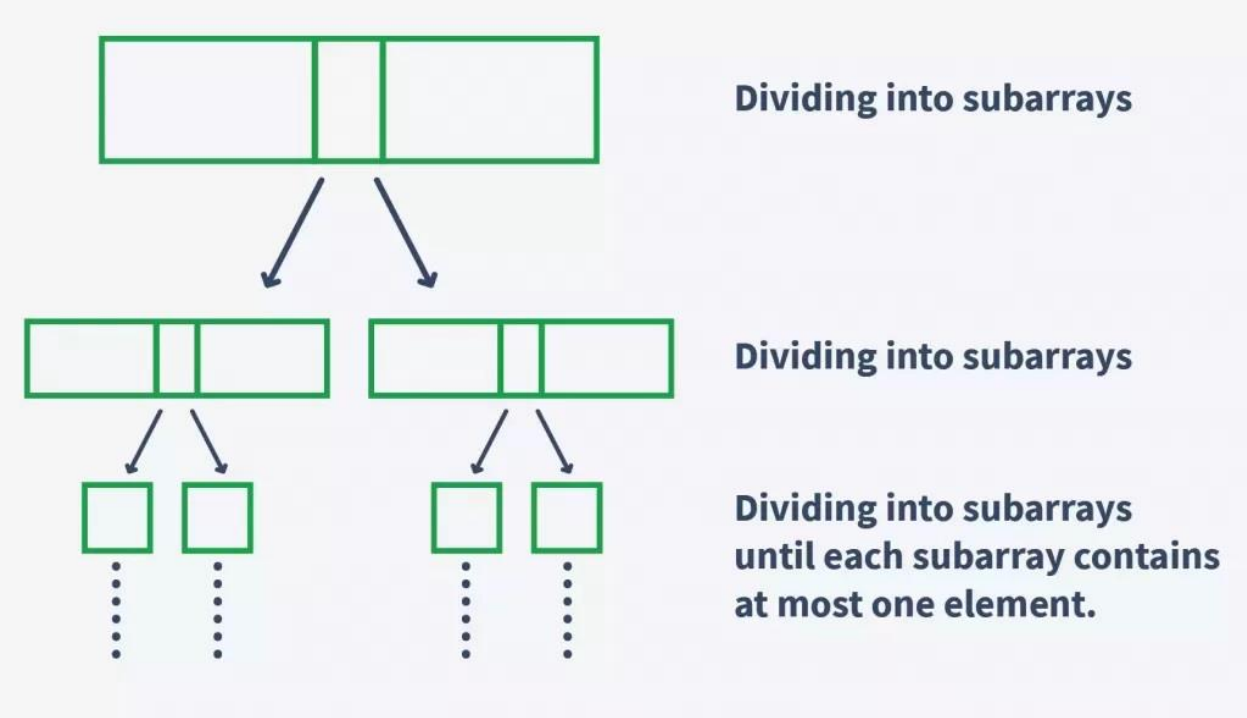
Below given is a representation of what the input array looks like after first iteration of quick sort:



At every iteration of quick sort, the chosen pivot element is placed at its correct position that it should acquire in the sorted array. This makes sure that each chosen pivot element is at its correct sorted position.

This process of selecting the pivot element and dividing the array into subarrays is carried out recursively until all the elements in the array are sorted. In other words, each subarray is divided further until each subarray consists of only one element. All these subarrays are then combined to form a single sorted array. Since we know before partitioning of the array, the pivot is placed

at its correct position, dividing the array to one single element places each element at its correct sorted position, and thus, combining all of them gives us a sorted array.



This way, with each iteration of quick sort, the problem reduces by 2 steps making quick sort a faster way to sort an array.

One interesting thing to note here is that we can choose any element as our pivot, be it the first element, last element, or any random element from the array.

To better understand the quick sort algorithm, imagine this. Owing to your good conduct, your class teacher gives you the responsibility of a monitor. One day, for a fest, your teacher calls you up and asks you to help her in arranging the students in height-wise order.

There are many ways in which you can arrange the students, you can pick every student and show them their places one-by-one, but this would take too much time if there are a lot of students (say 500), and you have to finish your job as soon as possible. So, one of the fastest ways could be asking the students to arrange themselves. **For example**, the shortest student knows that he has to stand at the front. Same for the tallest student and the students with medium height.

In this way, every student can find his/her correct position by comparing his/her height with that of students before and after him/her.

- Similarly, in quick sort, every element arranges itself at its correct position to sort the given array. Here, the pivot element is placed at its correct sorted position, and hence it is

the element that we know is definitely sorted. And thus, subarrays are divided around the pivot element.

How Does Quick Sort Algorithm Work?

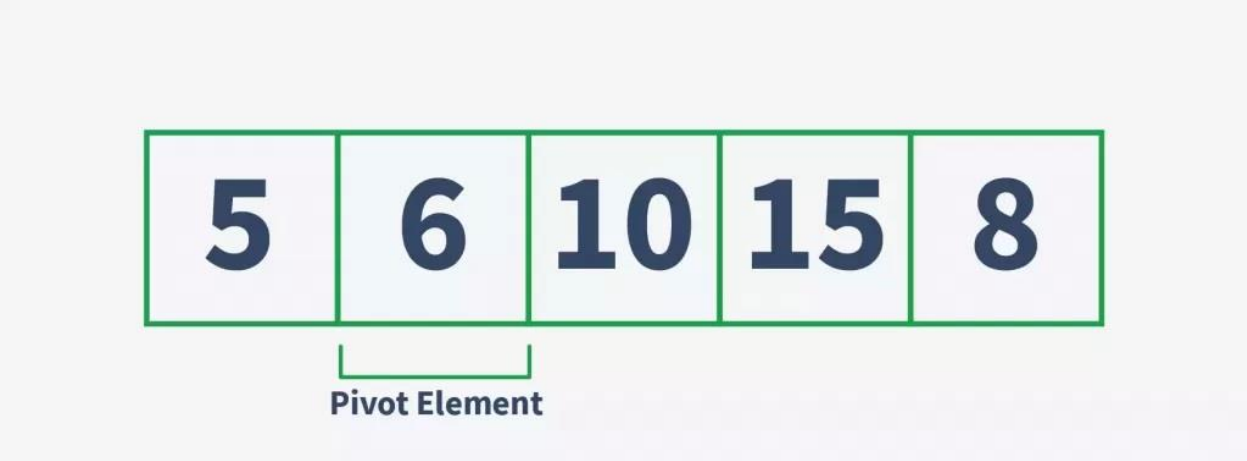
Recall the list/array that had the elements – 10, 8, 5, 15, 6 in it. To sort these elements in ascending order using quick sort, let us follow the under-given steps –

Step – 1:

Select the pivot element, here for the sake of simplicity, we are selecting the rightmost element as the pivot.

Step – 2:

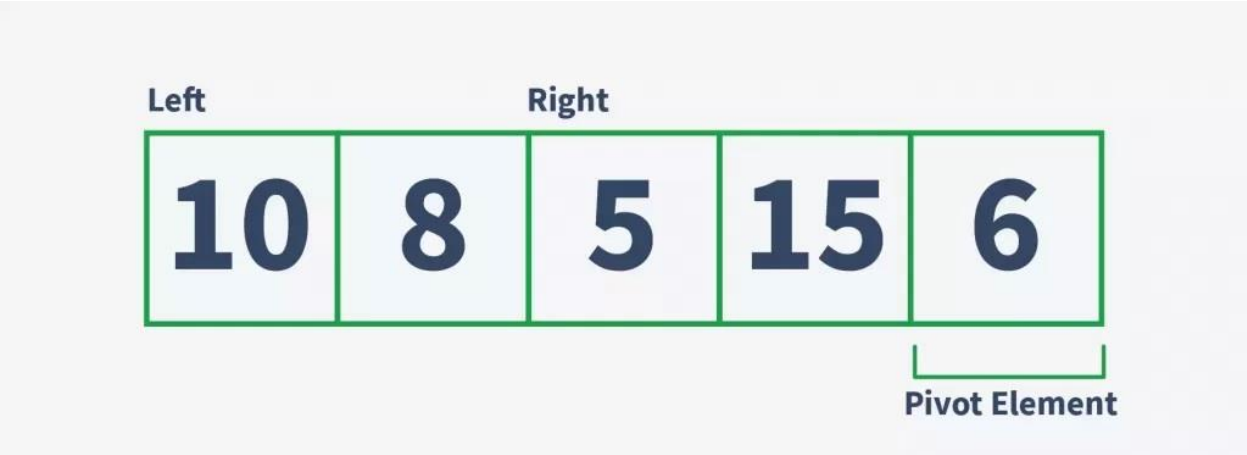
Rearrange the elements of the array in a way that all the elements smaller than the pivot are on its left and that all the greater ones are on the right. They need not be sorted. After the rearrangement, the array would look like this –



Here, 6 is our pivot element, and all the elements greater than itself are on the right side, while the elements smaller than the pivot are on its left side.

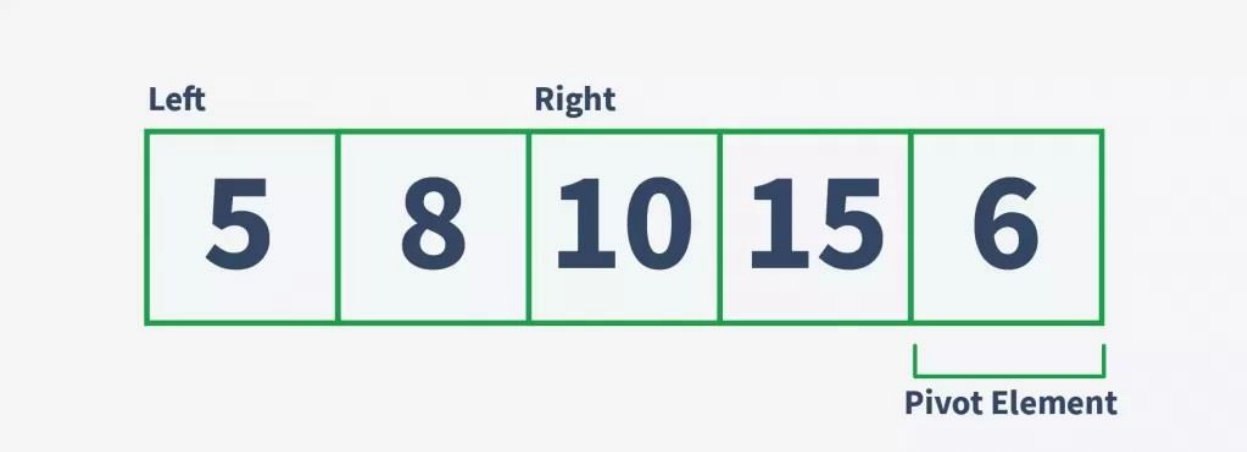
Now let us understand how did this rearrangement happen –

- Take two pointer variables that point to the left & right of the array (with pivot excluded). Let's call them left & right respectively.



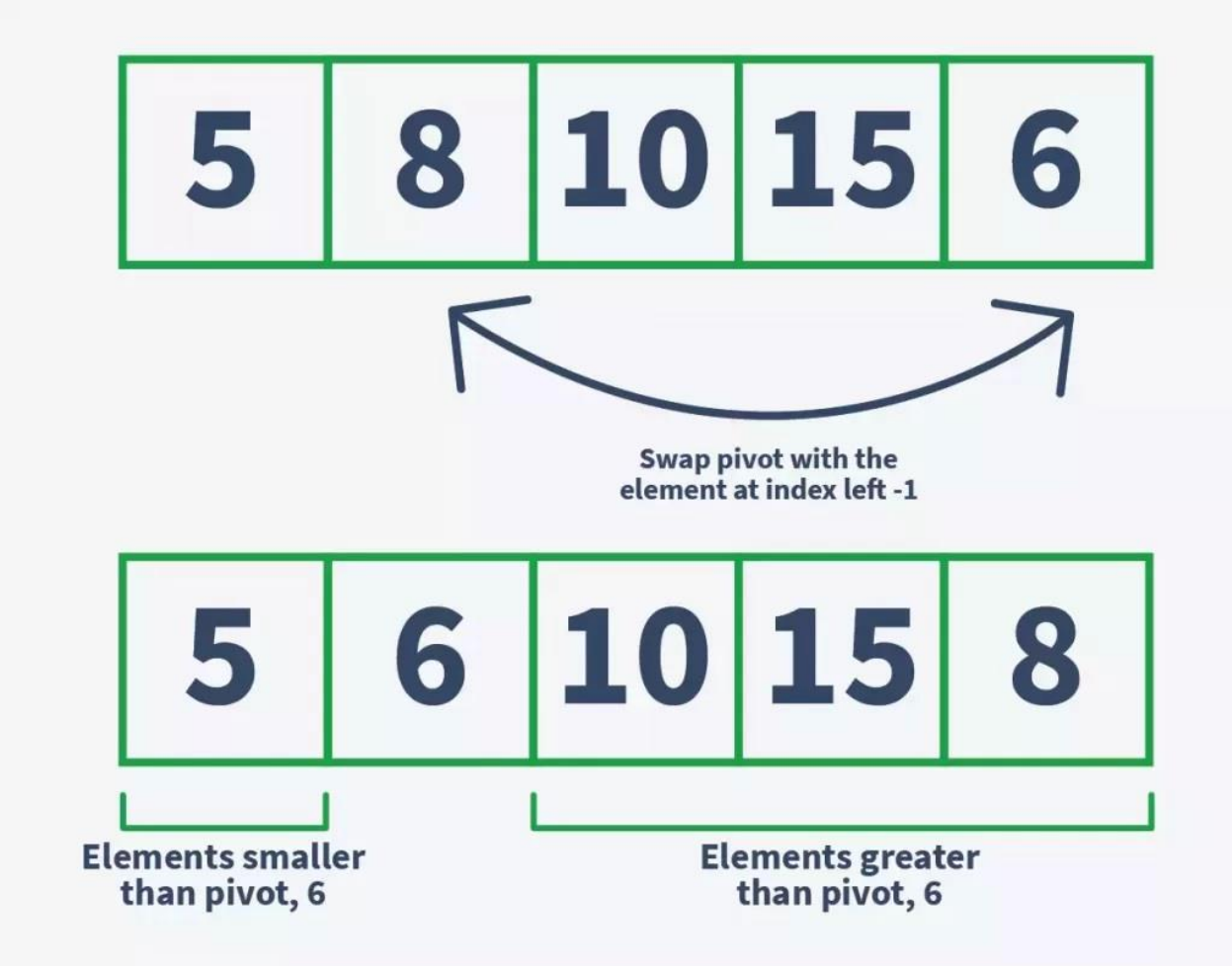
- Under a loop, increment the left pointer until a value greater than pivot is found. In this case, since 10 is already greater than 6, we move to step c.
- Under the same loop, increment the **right** pointer until a value smaller than pivot is found. Here, a value smaller than pivot is 5.

- Now since the conditions given in steps b and c are simultaneously fulfilled, swap the value at index **left** with the value at index **right** of the given array. This helps us in moving the elements smaller than the pivot element to the left of it, and the greater ones to the right of it.



Here, 5 & 10 are swapped.

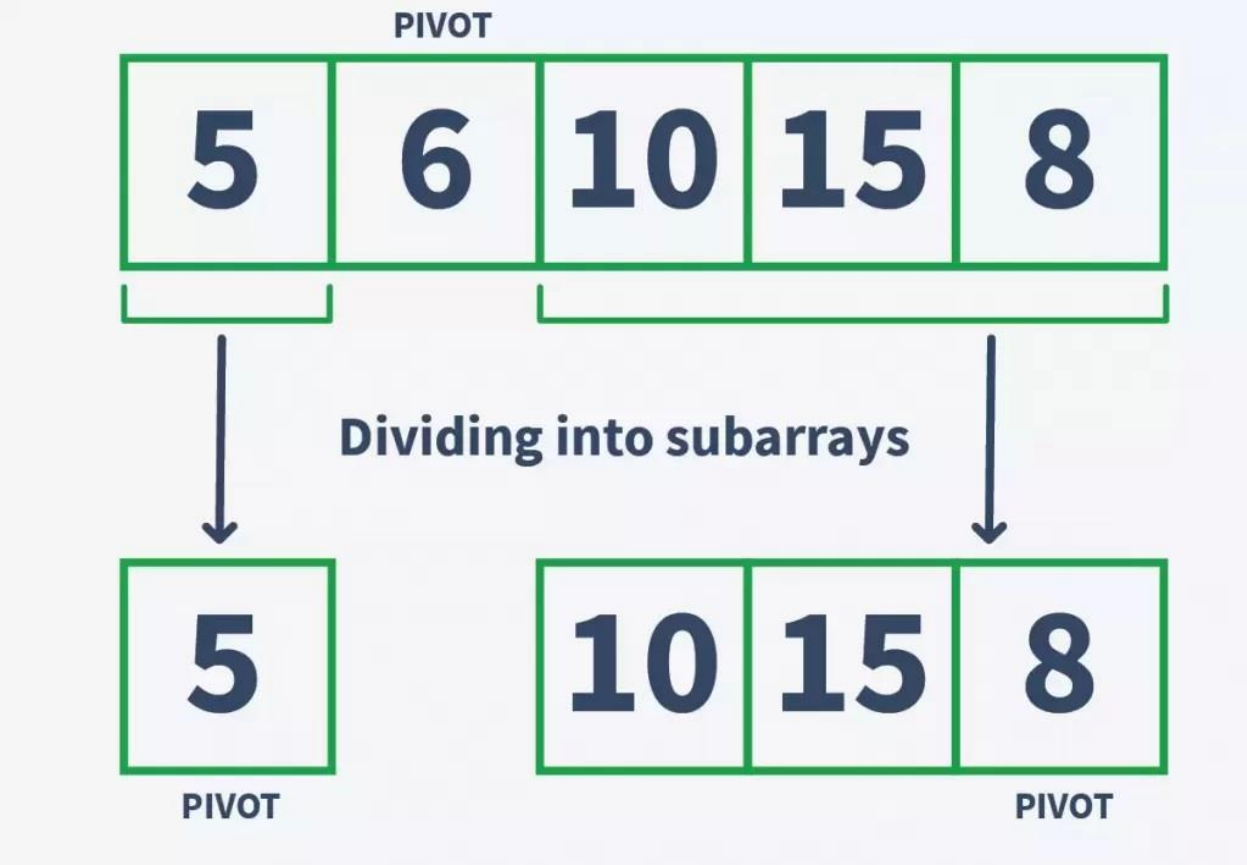
- Repeat steps b, c and d until the **left** index is greater than the **right** index.
- Here, the execution of the loop is stopped because **left > right**. Now, swap the value at pivot with the value at the index **left** or **right**.



We have placed 6 after 5, and before 10, 15, and 8. This way, we have placed 6 (pivot element) at such a position that elements smaller than it are on its left, and the elements greater than it are on the right. Notice that 6 is at its correct sorted position now.

Step – 3:

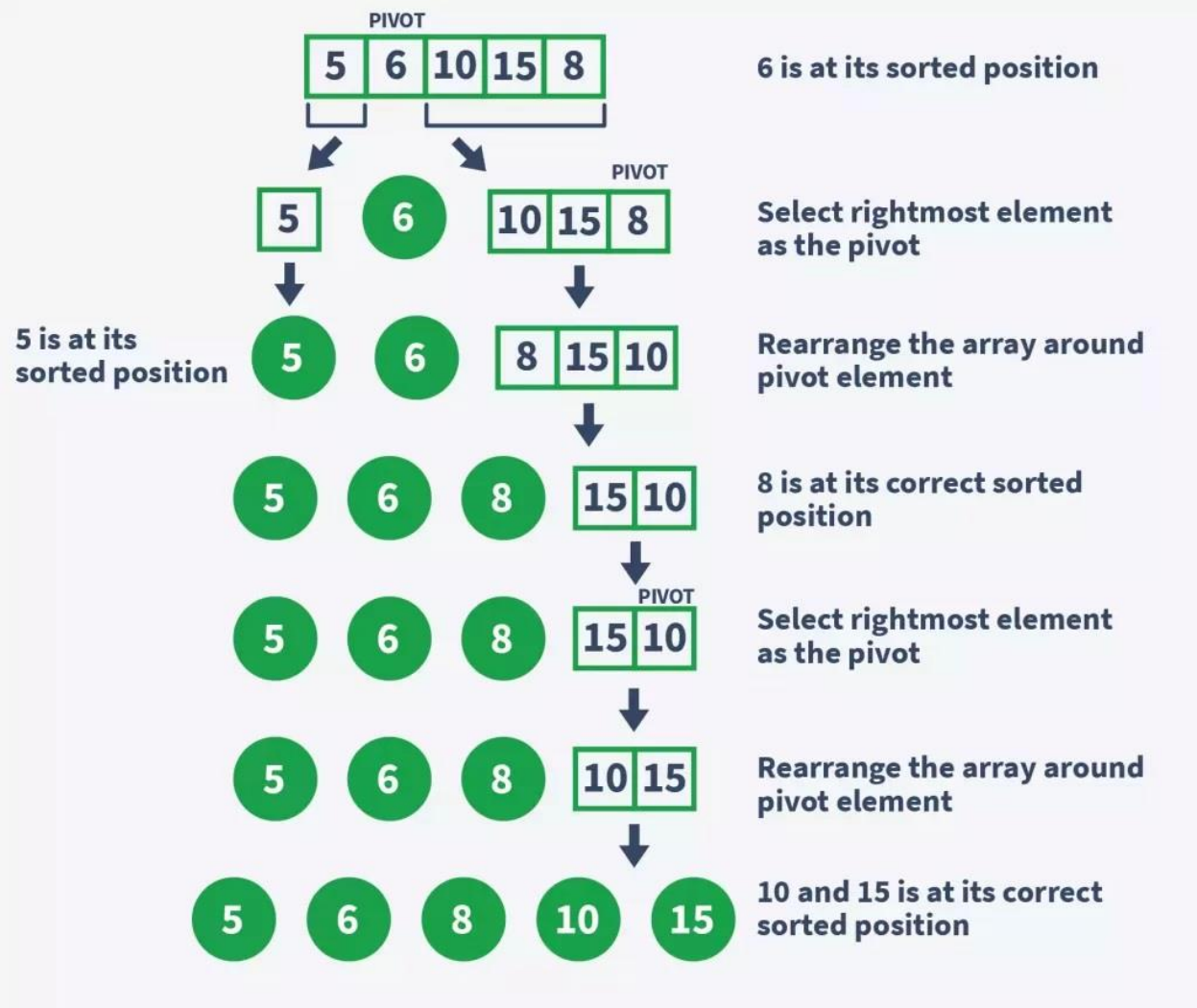
In this step, the sublist is divided into two parts – sublist before the pivot element, and sublist after the pivot element.



Step – 4:

Repeat the above-given steps recursively until all the elements of the array are at their correct sorted position.

The following diagram shows the working of the quick sort algorithm.



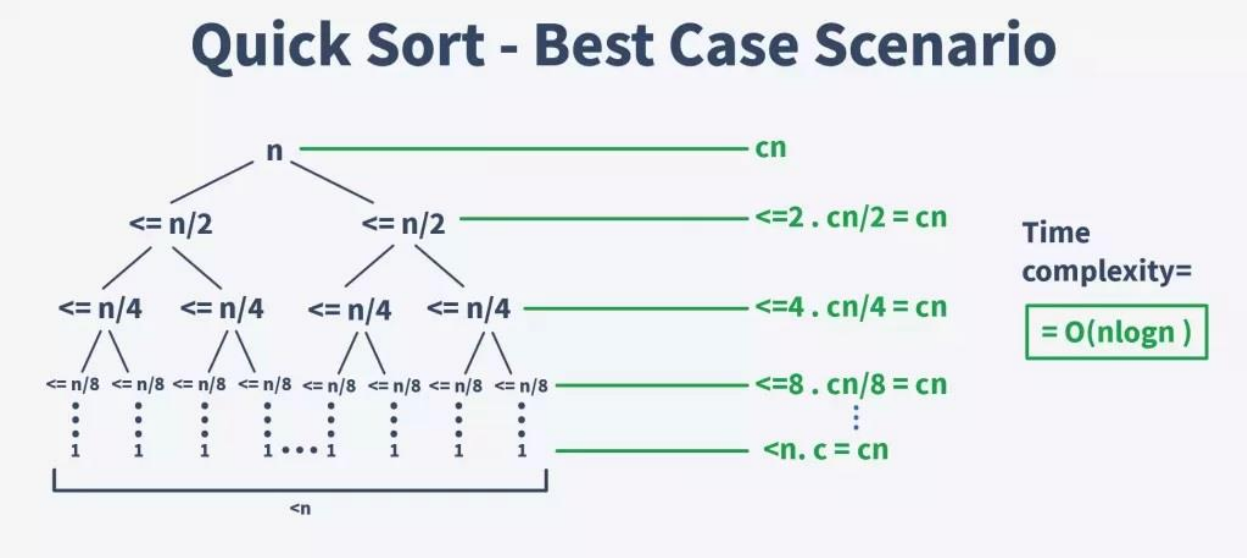
Complexity Analysis of Quick Sort

(a) 1. Time Complexity of Quick Sort

Best-Case:

The best-case occurs when the pivot is almost in the middle of the array, and the partitioning is done in the middle. So, let us assume that we have a total of n elements in the array, and we are dividing the array from the middle.

In this case, with each partition, we will have at most **n/2** elements. And we have to perform the partition until only one element is left in each subarray. The following is the tree representation of the given condition



Notice that the above-given tree is a binary tree. And for a binary tree, we can know that its height is equal to **logn.** Therefore, the complexity of this part is **O(logn).**

Also, the time complexity of the **partition()** function would be equal to **O(n)**. This is because, under the function, we are iterating over the array to swap rearranging the array around the pivot element.

Therefore, from the above results, we can conclude that the best-case time complexity for the quick sort algorithm is **O(nlogn).**

Worst Case:

The Worst-case occurs when either of the two partitions is unbalanced. This generally happens when the greatest or smallest element is selected as the pivot.

In this case, the pivot lies at the extreme of the array, and one subarray is always empty while the other contains **n – 1** elements.

This way, in the first call, the array is divided into two subarrays of 0 & n – 1 elements respectively. For the second call, the array with n – 1 elements is divided into two subarrays of 0 & n – 2 elements respectively. This continues till only one element is left.

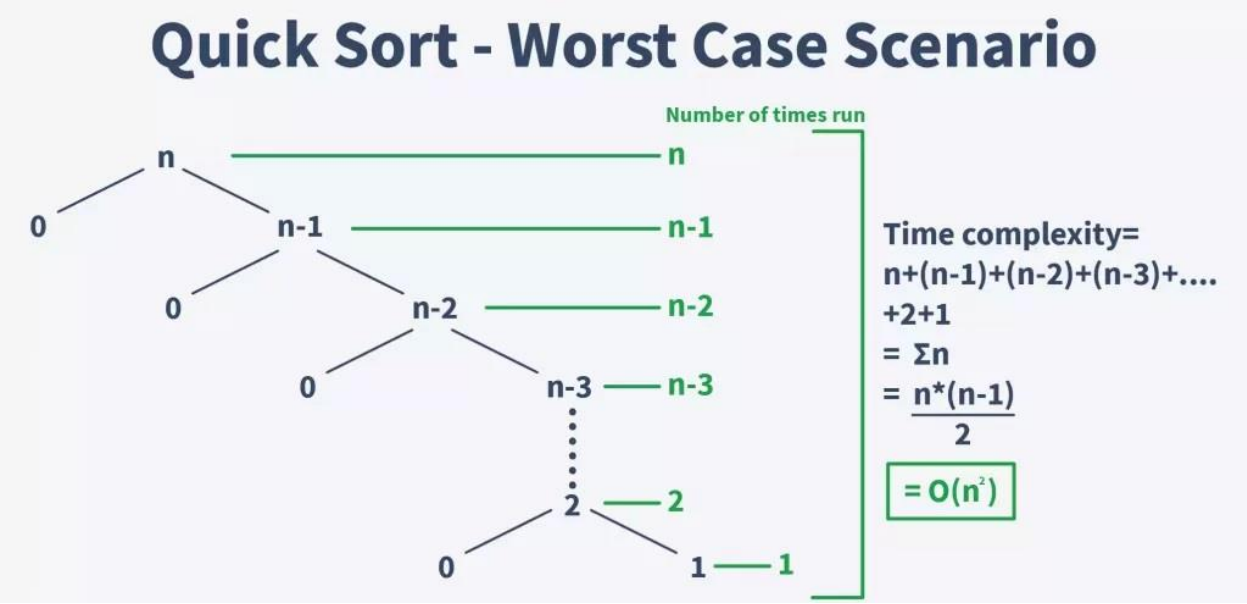
The time complexity, in this case, would be the sum of all the complexities at each step.

T(quicksort) = T(n) + T(n – 1) + T(n – 2) + …….. + 2 + 1

T(quicksort) = O((n * (n – 1))/2)

T(quicksort) = O(n ^ 2)

The following is the tree representation of this condition –



(b) 2. Space Complexity

In quick sort, the time complexity is calculated on the basis of space used by the recursion stack. In the worst case, the space complexity of quick sort is O(n) because in the worst case, n recursive calls are made. And, the average space complexity of a quick sort algorithm is equal to **O(logn)**.

Some Common Questions and Answers

(c) 1. Where is quick sort used?

- Quick sort is used in separating Nth smallest or the largest element from an array.
- Being a fast sorting algorithm, quick sort is used by various departments in information retrieval.
- Quick sort is used for numerical computing where the efficient algorithms use priority queues and in turn quick sort for achieving accuracy.
- It is used where fast searching and/or sorting is the priority.

(d) 2. Is Quick sort a stable algorithm?

An algorithm is called stable if the relative order of two equal elements is preserved. Therefore, quick sort is not a stable algorithm because ordering is done on the basis of the pivot's position.

(e) 3. Why Quick sort runs faster than merge sort?

Quick sort is an in-place algorithm. But in merge sort, a new temporary array is required to merge the sorted subarrays. Therefore, quick sort is faster than merge sort.

Moreover, in quick sort, the worst case can be avoided by randomly choosing an element as the pivot, and performing sorting around it. Thus, a properly implemented quick sort has a time complexity of O(nlogn) in the average case.

(f) 4. Why Quick sort is preferred for array?

Quick sort is cache-friendly, which means it does not create unnecessary buffers in the cache memory and slow the system down. It is because, when used for arrays, quick sort has a good locality of reference

Conclusion

- Quick Sort method sorts the elements using the Divide and Conquer approach.
- Quick Sort has an average O(nLogn) complexity.
- Quick Sort can be implemented in both recursive and iterative way.
- Quick Sort is in-place, cache-friendly and also a tail-recursive algorithm