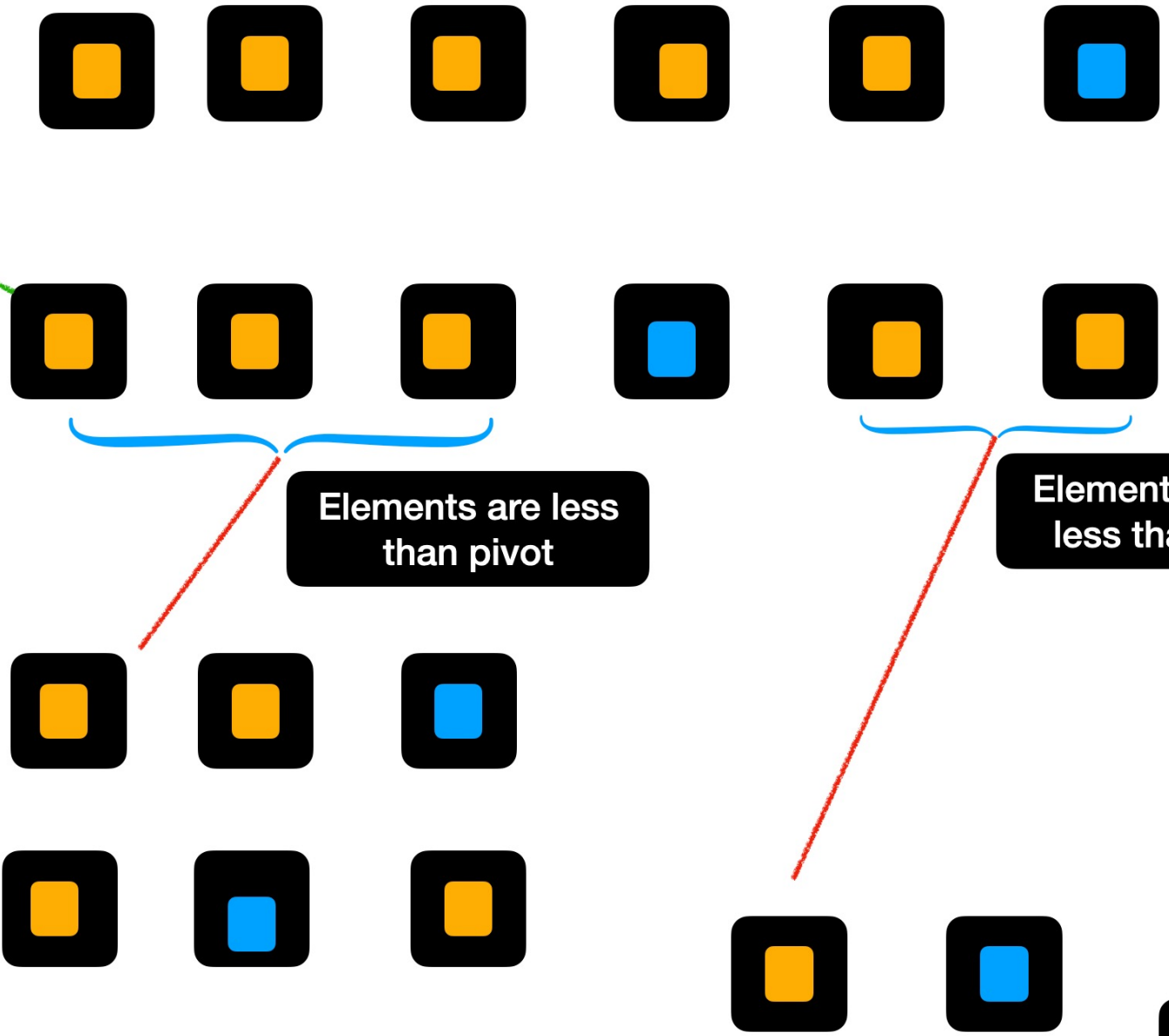
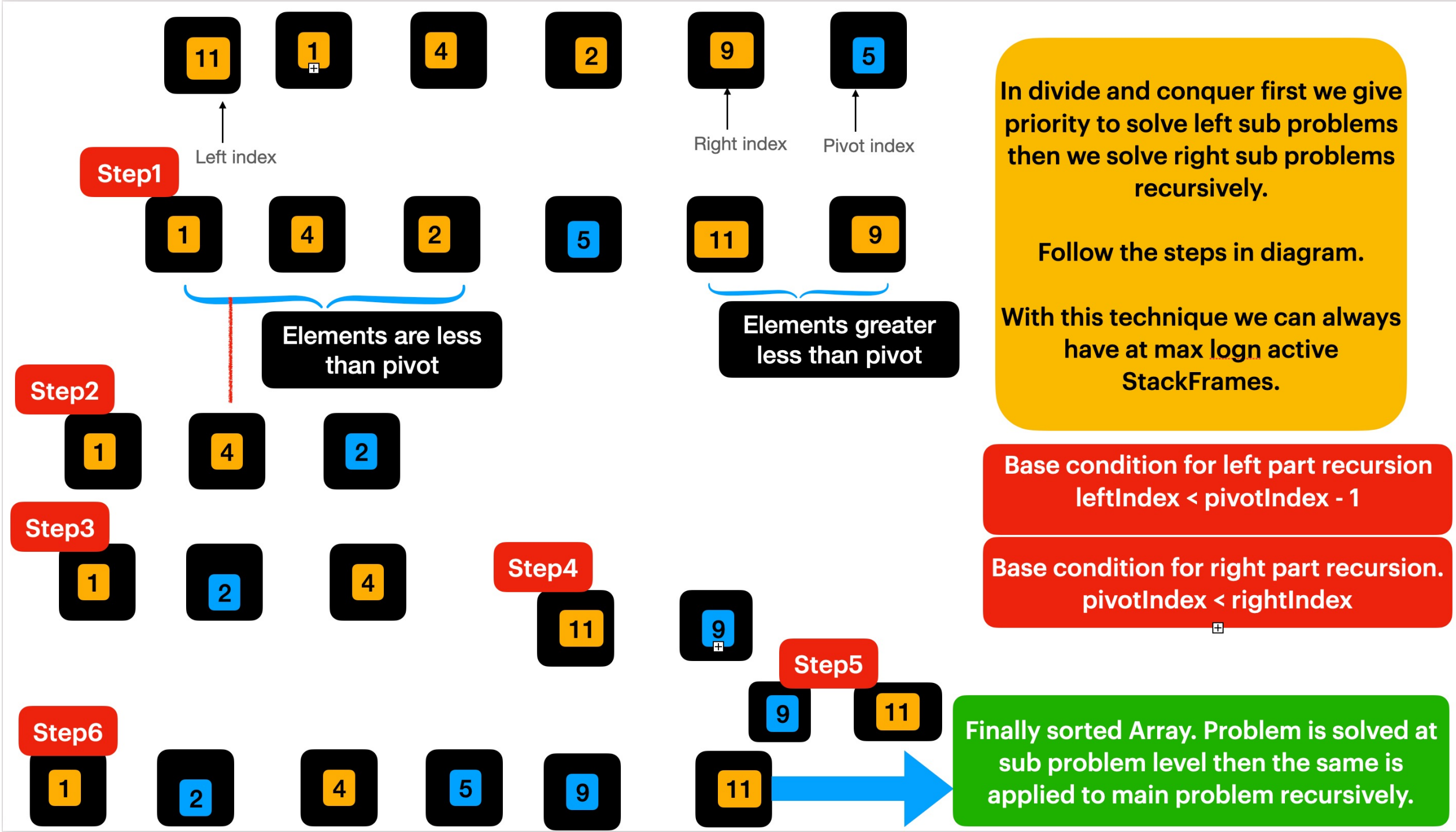
Quick Sort : Quick Sort uses the divide & conquer approach. Quick Sort chooses the pivot element within a collection of elements then do the sorting such that elements less than the pivot moved to left part and the elements greater than pivot moved to right part.
Same logic applies on left Part and Right Part recursively.

Elements are sorted with respect to pivot. This little observation makes QuickSort to do only logn comparisons in recursive calls.

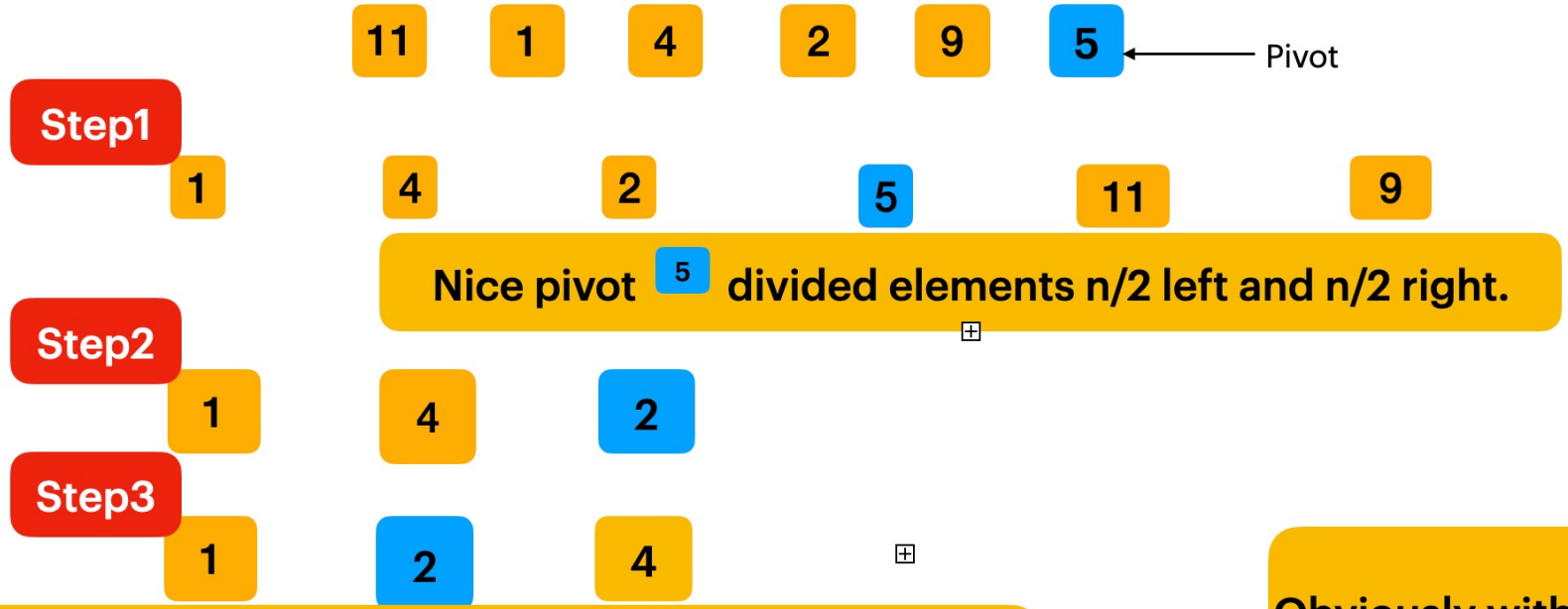


In Quick Sort choosing a pivot is a critical part. In each recursive step the pivot has to partition the elements half (i.e n/2 & n/2 for both left and right) . So that in the next recursive call either in leftPart or in rightPart takes only logn comparison.


11
1
4
2
9
5
Left index
Right index
Pivot index
Step1
1
4
2
5
11
9
Elements are less than pivot
Elements greater less than pivot
Step2
1
4
2
Step3
1
2
4
Step4
11
9
Step5
9
11
Step6
1
2
4
5
9
11
In divide and conquer first we give priority to solve left sub problems then we solve right sub problems recursively.
Follow the steps in diagram.
With this technique we can always have at max logn active StackFrames.
Base condition for left part recursion leftIndex < pivotIndex - 1
Base condition for right part recursion. pivotIndex < rightIndex
Finally sorted Array. Problem is solved at sub problem level then the same is applied to main problem recursively.

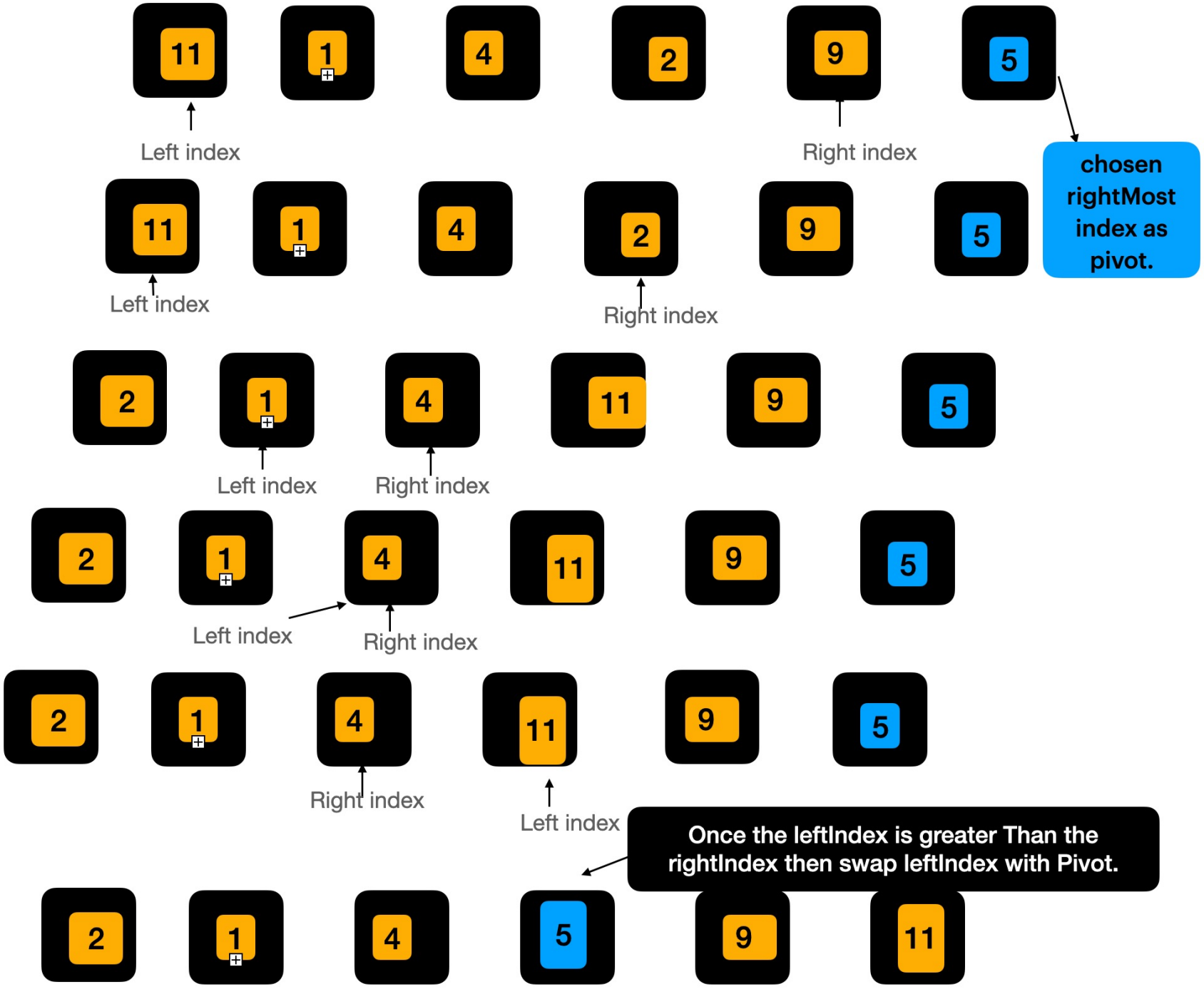
Choosing Pivot in Quick sort makes a critical role.
Let's consider for simplicity we chooses rightMost Index as Pivot.

Assume input elements are in unsorted order



Nice pivot 2 divided elements, n/2 left & n/2 right. There is no division of elements for sorting either at left or right part with respect to 2 so go to the parent pivot 5 to solve right part of the array.

Obviously with respect to pivot 9 the other element will be moved right , more or like division is n/2.



=> Quick sort uses InPlace Algorithm :
This is the advantage over the merge sort. QuickSort takes O(1) Space complexity for sort the elements.

while (leftIndex <= rightIndex) {

1. Make sure leftIndex points to lower index.

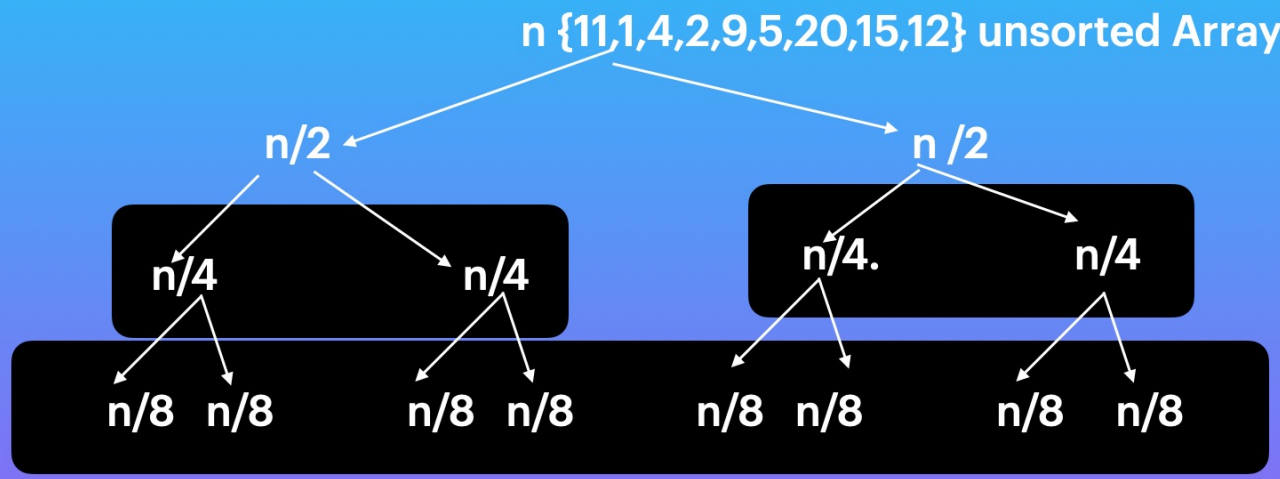
2. Make sure rightIndex points to higher index (pivotIndex - 1).

3. If the leftIndex value less than pivot then movie leftIndex to right.

4. If the rightIndex value greater than pivot move rightIndex to left.

5. When the leftIndex value > pivot && rightIndex value < pivot then swap
leftIndex++
rightIndex—
}

Let's figure out the TimeComplexity for QuickSort. If the elements are unsorted.

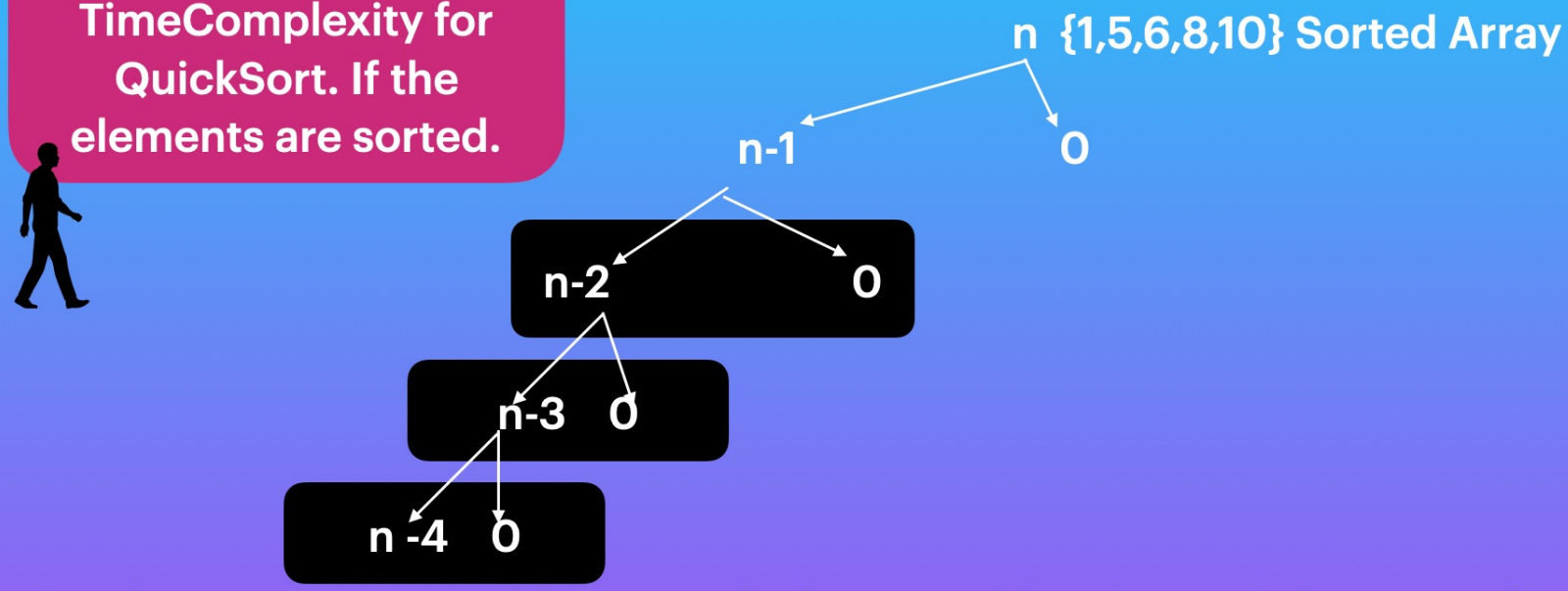


Now when the elements are unsorted {11,1,4,2,9,5} in a quick sort either in best or average Case => In each level pivot divides into n/2 left and n/2 right sub problems . In each sub problem altogether sort happens at left and right recursively.

As per diagram n = (leftPart Recursion) + (rightPart Recursion)
= n/2 * logn + n/2 * logn = 2 * n/2 * logn = nlogn

For n pivot values logn comparisons happens at each level so that Time Complexity is O(nlogn)

Let's figure out the TimeComplexity for QuickSort. If the elements are sorted.



Now when the elements are sorted {1,5,6,8,10}, If we consider always rightMost Index as pivot then in each level pivot divides into (n-1) left and 0 right sub problems .

In each sub problem sort happens only at left part recursively.

As per diagram n = (leftPart Recursion) + 0 = n(n-1) + n(n-2) +...+1 = n^2

For a sorted array QuickSort gives O(n^2) Time Complexity.

This can be addressed by taking proper pivot.

Pivot Techniques :

Usually for simplicity in a quick sort either we take leftMost or rightMost index a pivot but it leads to O(n^2) time complexity when the elements in an array are sorted so that choosing right pivot is always makes your algorithm best in QuickSort.

Here are the Pivot Techniques :

1. Always choose random index as a pivot index, it results into O(nlogn) time complexity.
(This technique is been used in our code example refer QuickSortApp.java)

2. If you are a good mathematician you can take a few elements in a given array as a sample, identify the behaviour of elements then choose the pivot .
(OR)
you can take last few elements find the median then choose the pivot.

3. You can also have dual pivots in your algorithm to improve the performance of a quick sort.

When to go for QuickSort ?

If most of the elements are already sorted then go with InsertionSort, which gives you best time complexity i.e O(n).

If the elements are Unsorted then you can prefer Quicksort with proper pivot technique, which gives you best time complexity i.e O(nlogn).

Finally On QuickSort

TimeComplexity :

Best Case or Average Case = O(nlogn)

Worst Case = O(n^2)

n^2 would be avoided if you go with proper pivot even in worst case.

Space Complexity :

As its following InPlace algorithm so that Space Complexity is O(1)

If you are more particular about active stack frames for quick sort Space Complexity would be O(logn)

Stable : Unstable

Recursive / NonRecursive => Recursive

Internal / External => Internal Sort

Comparison => Yes

Swaps => lögn (As we go with proper pivot)

How Come QuickSort gives best performance when both MergeSort & QuickSort gives same Time Complexity i.e O(nlogn) ?

Actually quick sort uses InPlace algorithm so the sort happens within the array. In CPU there is branch prediction feature , in simple branch prediction is nothing but maintaining the cache & guessing the next requested value.

This is what makes the difference between QuickSort and MergeSort.

As QuickSort performs sorting within the array(InPlace Algorithm) , the CPU branch prediction feature helps to execute faster.

Where as MergeSort uses out place algorithm, it results into effecting execution time.

Merge Sort work efficiently in external sorting. Always use QuickSort for internal Sorting , use MergeSort for external sorting.