Intuition: Given an array arr[]: {3 6 1 4 2 5}, we choose an element (let's call it pivot) and place it at its correct position. We recursively call this function for the left and right parts where the elements are / maybe in wrong positions (The faith that our function will solve for the left and right sub-parts is the leap of faith).

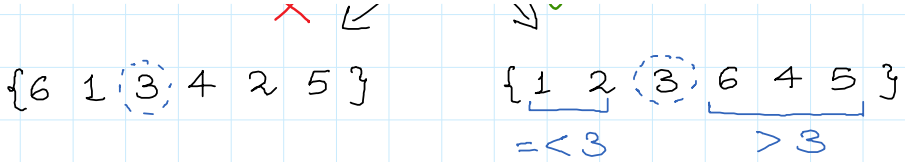
Example: arr[]: {3 6 1 4 2 5}
pivot

Let's say we keep the first element as our pivot. We must place 3 at its correct position. Find out how many elements are smaller than pivot. In our case, there are 2 elements 1 and 2 that are smaller than 3, so the correct position of 3 is index 2.

Note: Placing the pivot at its correct position also means that all elements to its left must be smaller and all elements to its right must be greater than the pivot (in no particular order).

arr[]: {3 6 1 4 2 5}

{6 1 3 4 2 5} {1 2 3 6 4 5}



$\Rightarrow$ arr[] = {1 2 3 6 4 5}

Call 1 Call 2

Code : Main code (Simple Recursive Calls)

1. Base Case : if low exceeds high.
2. Partition the array and return the correct index of the pivot element.
3. Repeat for the left half.
4. Repeat for the right half.

Partitioning Code

1. For a given array in the range arr[s.....e].
2. Take the pivot element as arr[s].
3. Count all element smaller than pivot.
4. Place/Swap pivot element with arr[s + count]
5. Start checking all elements with i=s & j=e till i < pivot & j > pivot. If arr[i] > pivot and arr[j] < pivot then swap them.

```cpp
void quickSort(int arr[], int s, int e) {
    if(s >= e) {
        return;
    }

    // The arr[partIndex] will now be at its correct position
    int partIndex = partition(arr, s, e);
    cout << "partIndex = " << partIndex << endl;

    // call the function for the left part
    quickSort(arr, s, partIndex-1);
    // call it for the right part
    quickSort(arr, partIndex+1, e);
}
```

```cpp
int partition(int arr[], int s, int e) {
    int pivot = arr[s];
    // count elements smaller than pivot
    int count = 0;
    for(int i=s+1;i<=e;i++) {
        if(arr[i] <= pivot)
            count++;
    }
    int pivotIndex = s + count;
    swap(arr[s], arr[pivotIndex]);
    // now arr[pivotIndex] is correctly placed
    int i = s, j = e;
    while(i < pivotIndex && j > pivotIndex) {
        if(arr[i] > arr[pivotIndex] && arr[j] <= arr[pivotIndex])
            swap(arr[i++], arr[j--]);
        else if(arr[i] < arr[pivotIndex]) {
            i++;
        }
        else {
            j--;
        }
    }
    return pivotIndex;
}
```

Homework :

① Is quick sort, an in-place sorting algorithm ?

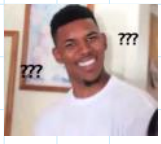
Ans: Yes. We are not creating secondary arrays and performing swaps in the original array.

② Is quick sort a stable sorting algorithm ?

Ans : Refer Here for detailed explanation

Time and Space Complexity

Time Complexity - Average Case : $O(n \log n)$
Worst Case : $O(n^2)$



Space Complexity - $O(n)$

Time Complexity of Recursive functions baadme padhaenge.

Hamari baat maan lo.