QUICK SORT

THIS IS ONCE AGAIN A DIVIDE AND CONQUER ALGORITHM WHICH PARTITIONS THE LIST AT EVERY STEP

THE PARTITION IS NOT BASED ON THE LENGTH OR AN ARTIFICIAL INDEX, IT'S BASED ON A PIVOT

THE PIVOT IS AN ELEMENT FROM THE LIST

THE LIST IS PARTITIONED WITH ALL ELEMENTS SMALLER THAN THE PIVOT ON ONE SIDE AND LARGER THAN THE PIVOT ON THE OTHER

THIS PIVOT PARTITION IS APPLIED TO ALL SUB-LISTS TILL THE LIST IS SORTED

QUICK SORT

FIRST WE CHOOSE A PIVOT TO PARTITION THE LIST

IN VERY FIRST ITERATION THE SUBLIST WHICH WE PARTITION IS THE ENTIRE LIST

PIVOT

6	5	4	2	1	10	3	7	8	9

THERE IS NO EXACT SCIENCE BEHIND THE PIVOT, USUALLY THE FIRST OR THE LAST ELEMENTS OF THE SUB-LIST ARE CHOSEN

QUICK SORT

FIRST WE CHOOSE A PIVOT TO PARTITION THE LIST

IN VERY FIRST ITERATION THE SUBLIST WHICH WE PARTITION IS THE ENTIRE LIST

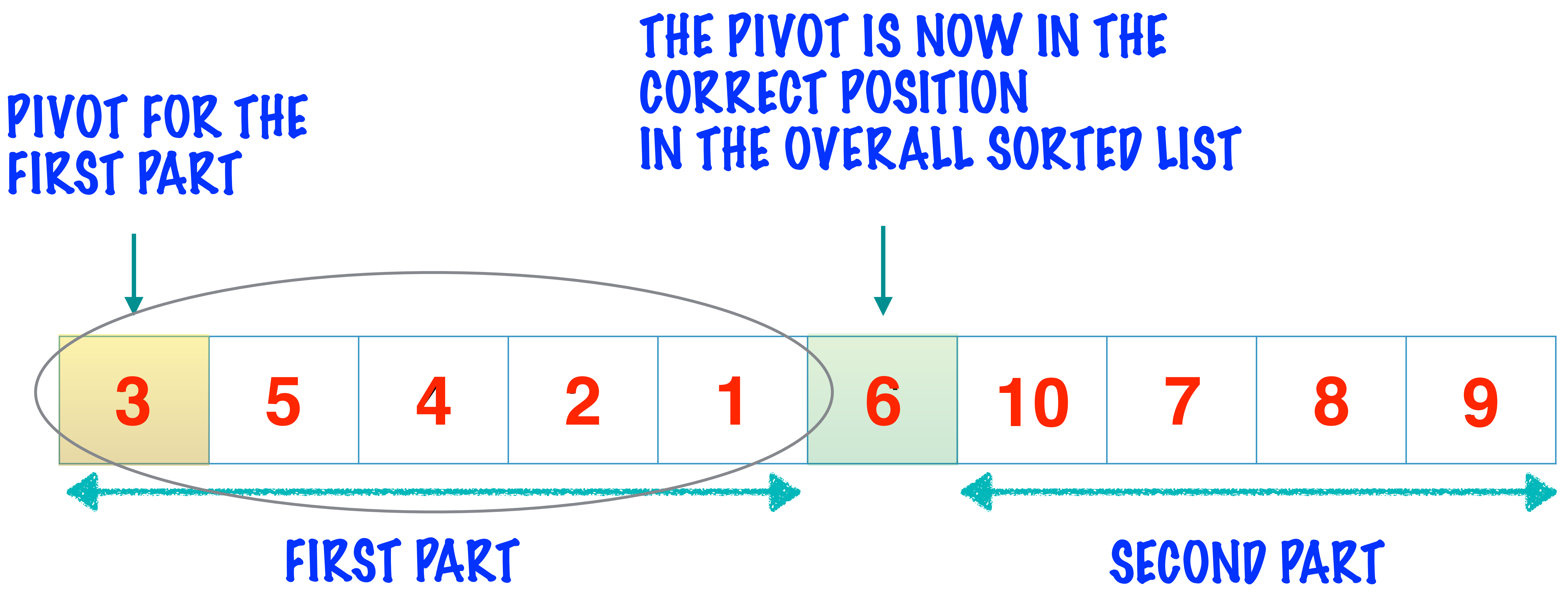
PIVOT

↓

6	5	4	2	1	10	3	7	8	9

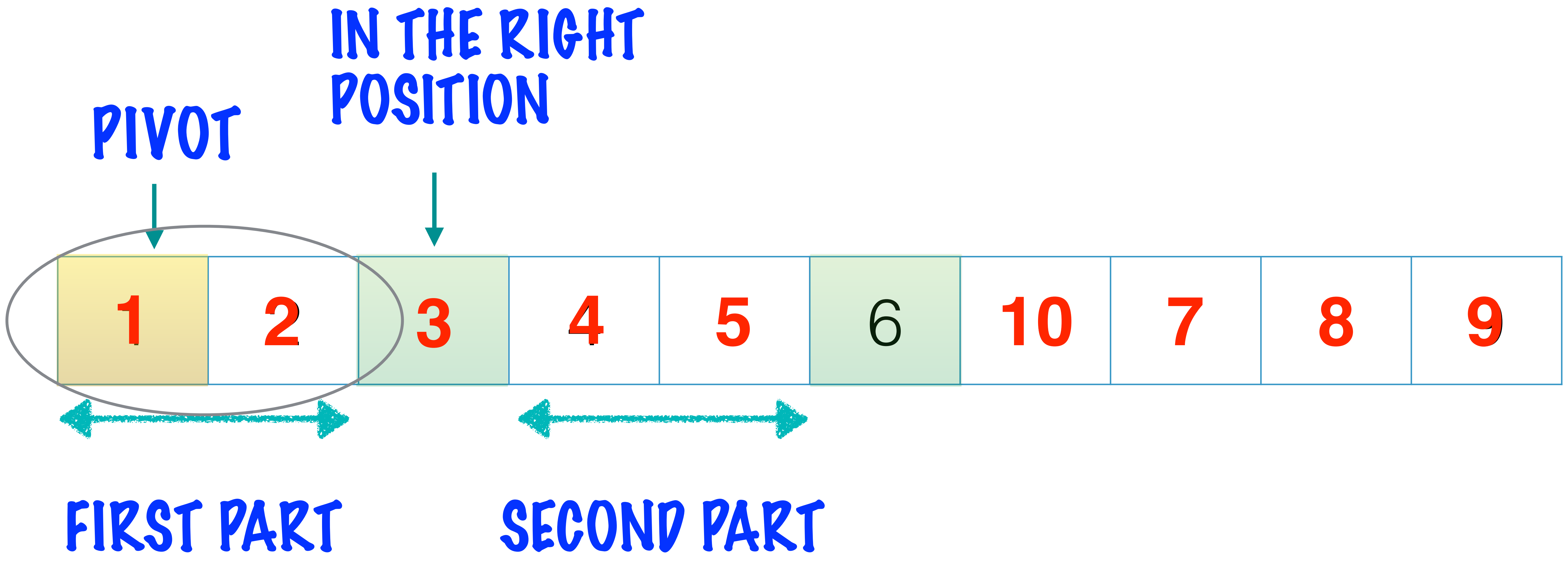
THERE IS NO EXACT SCIENCE BEHIND THE PIVOT, USUALLY THE FIRST OR THE LAST ELEMENTS OF THE SUB-LIST ARE CHOSEN

QUICK SORT



LET'S SORT THE FIRST PART OF THE LIST BEFORE MOVING ON TO THE SECOND

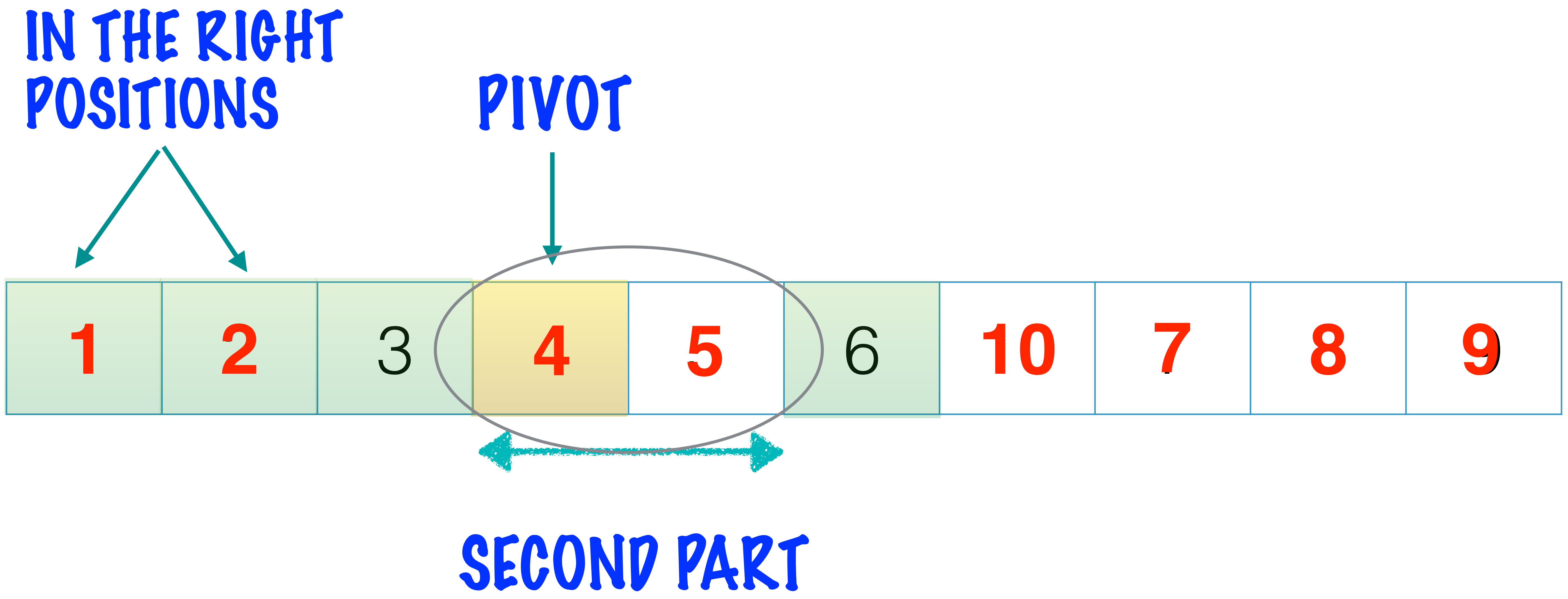
QUICK SORT



NOTE THAT ALL THAT IS NEEDED THAT THE ELEMENTS SMALLER THAN THE PIVOT MOVE TO THE LEFT, AND THOSE LARGER MOVE TO THE RIGHT

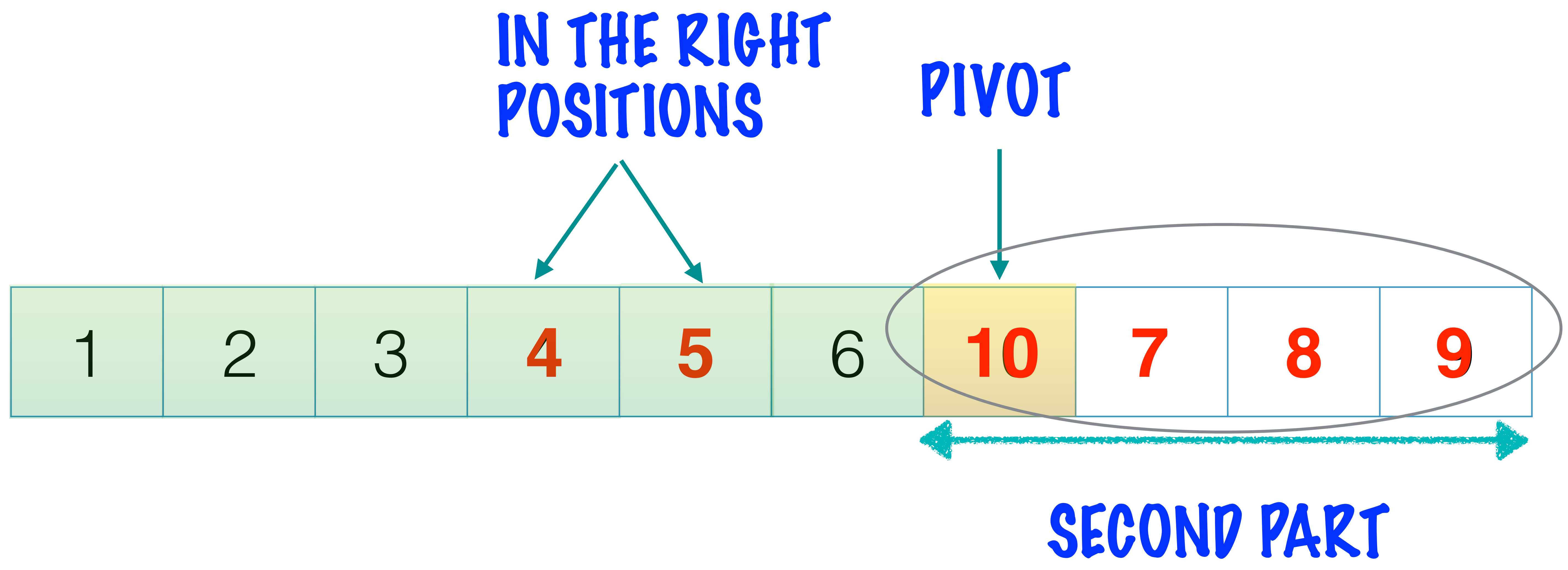
THEY NEED NOT BE IN ORDER ON EITHER SIDE OF THE PIVOT

QUICK SORT



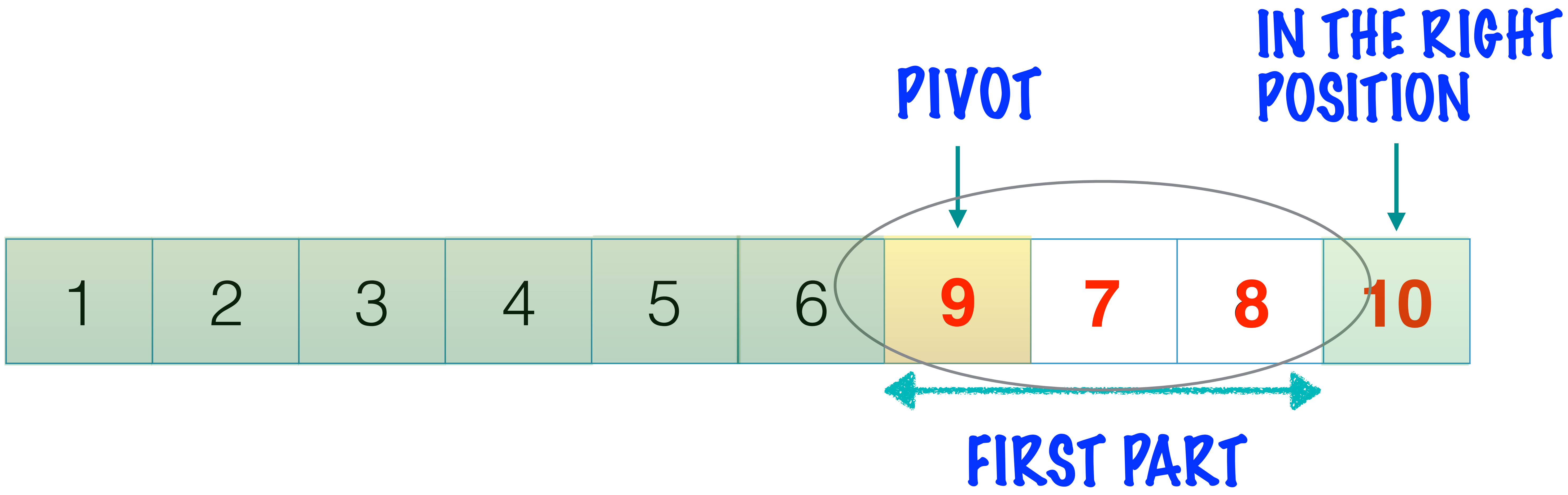
ITERATE TILL THE START AND END OF THE LIST MEET AT THE CENTER

QUICK SORT



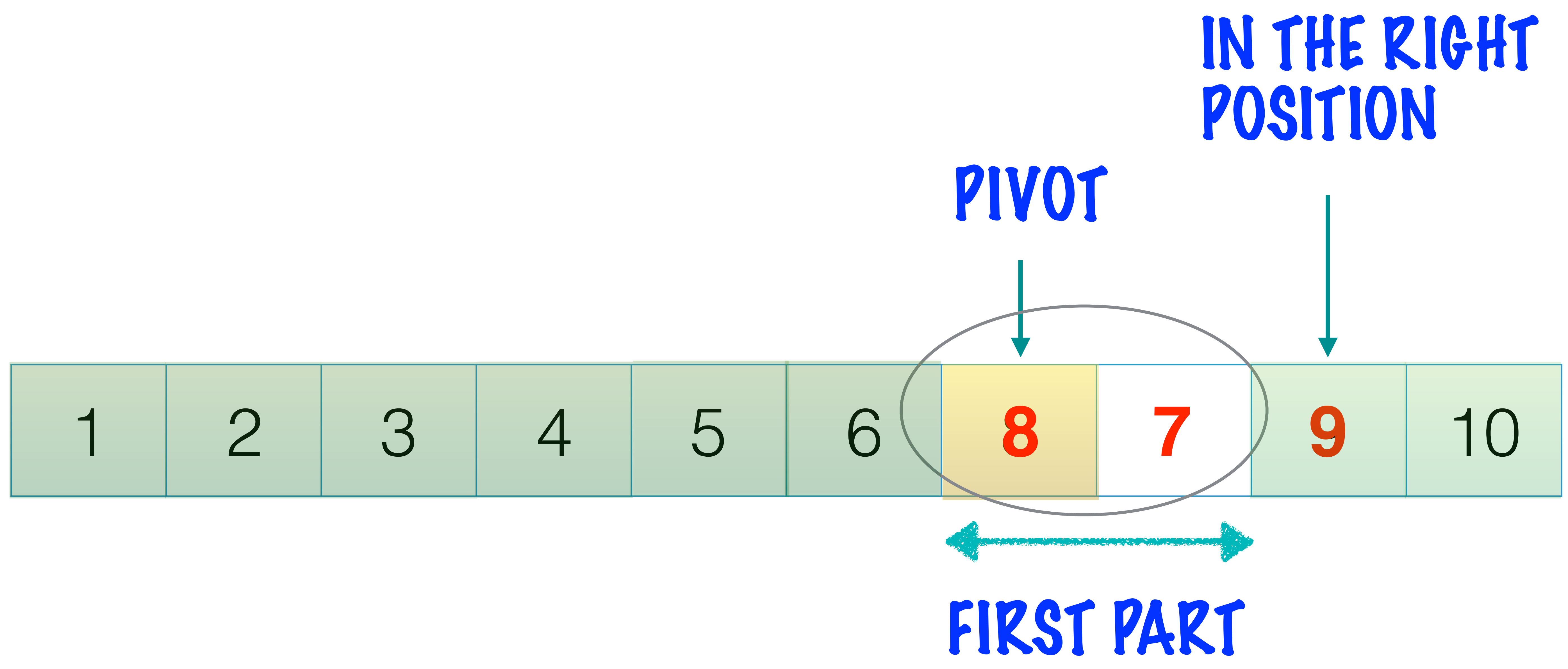
ONE-HALF OF THE LIST IS SORTED, NOW ON TO THE SECOND PART

QUICK SORT



LAST FEW ITERATIONS LEFT

QUICK SORT



QUICK SORT

1	2	3	4	5	6	7	8	9	10

A FULLY SORTED LIST!

QUICK SORT HAS 2 MAIN METHODS:

1: THE "PARTITION" METHOD WHICH FINDS A PIVOT AND MOVES ELEMENTS TO BEFORE OR AFTER THE PIVOT

2: THE "QUICKSORT" METHOD WHICH DOES THE RECURSIVE CALL TO SORT THE SUB-LISTS

THE "PARTITION"

```java
public static int partition(int[] listToSort, int low, int high) {
    int pivot = listToSort[low];
    int l = low;
    int h = high;
    while (l < h) {
        while (listToSort[l] <= pivot && l < h) {
            l++;
        }
        while (listToSort[h] > pivot) {
            h--;
        }
        if (l < h) {
            swap(listToSort, l, h);
        }
    }
    swap(listToSort, low, h);

    System.out.println("Pivot: " + pivot);
    print(listToSort);
    return h;
}
```

LOW AND HIGH SPECIFY INDICES WHICH DETERMINE WHAT PORTION OF THE LIST WE'RE WORKING ON

CHOOSE A PIVOT TO PARTITION THE LIST

MOVING FROM EITHER END OF THE LIST TOWARDS THE CENTER WE COMPARE THE ELEMENTS TO THE PIVOT

ELEMENTS LARGER THAN THE PIVOT ARE SWAPPED TO AFTER THE PIVOT AND SMALLER ELEMENTS MOVE BEFORE THE PIVOT

THE LOOP CONTINUES SO LONG AS THE INDICES DON'T CROSS ONE ANOTHER AT THE CENTER

SWAP THE PIVOT ELEMENT TO THE CORRECT POSITION IN THE LIST

THE "QUICKSORT"

STOP THE SORT IF THE LOWER INDEX IS NOT ACTUALLY LOWER THAN THE HIGHER INDEX

```
public static void quickSort(int[] listToSort, int low, int high) {
    if (low >= high) {
        return;
    }
    int pivotIndex = partition(listToSort, low, high);
    quickSort(listToSort, low, pivotIndex - 1);
    quickSort(listToSort, pivotIndex + 1, high);
}
```

FIND THE PIVOT

QUICKSORT THE PORTION OF THE LIST ON EITHER SIDE OF THE PIVOT

QUICK SORT USES DIVIDE AND CONQUER TO CREATE SMALLER PROBLEMS WHICH ARE EASIER TO TACKLE

JUST AS IN THE CASE OF OTHER DIVIDE AND CONQUER ALGORITHMS (E.G. MERGE SORT) THE COMPLEXITY HAS TO BE DERIVED

THE EXACT DERIVATION IS NOT REALLY RELEVANT TO PROGRAMMING INTERVIEWS

THE AVERAGE CASE COMPLEXITY OF QUICK SORT IS O(NLOGN)

QUICK SORT IS NOT ADAPTIVE

IT TAKES O(LOG N) EXTRA SPACE FOR THE CALL STACK IN THE RECURSION, THE WORST CASE FOR SPACE COMPLEXITY COULD BE O(N)

IT IS NOT A STABLE SORT