Quick Sort

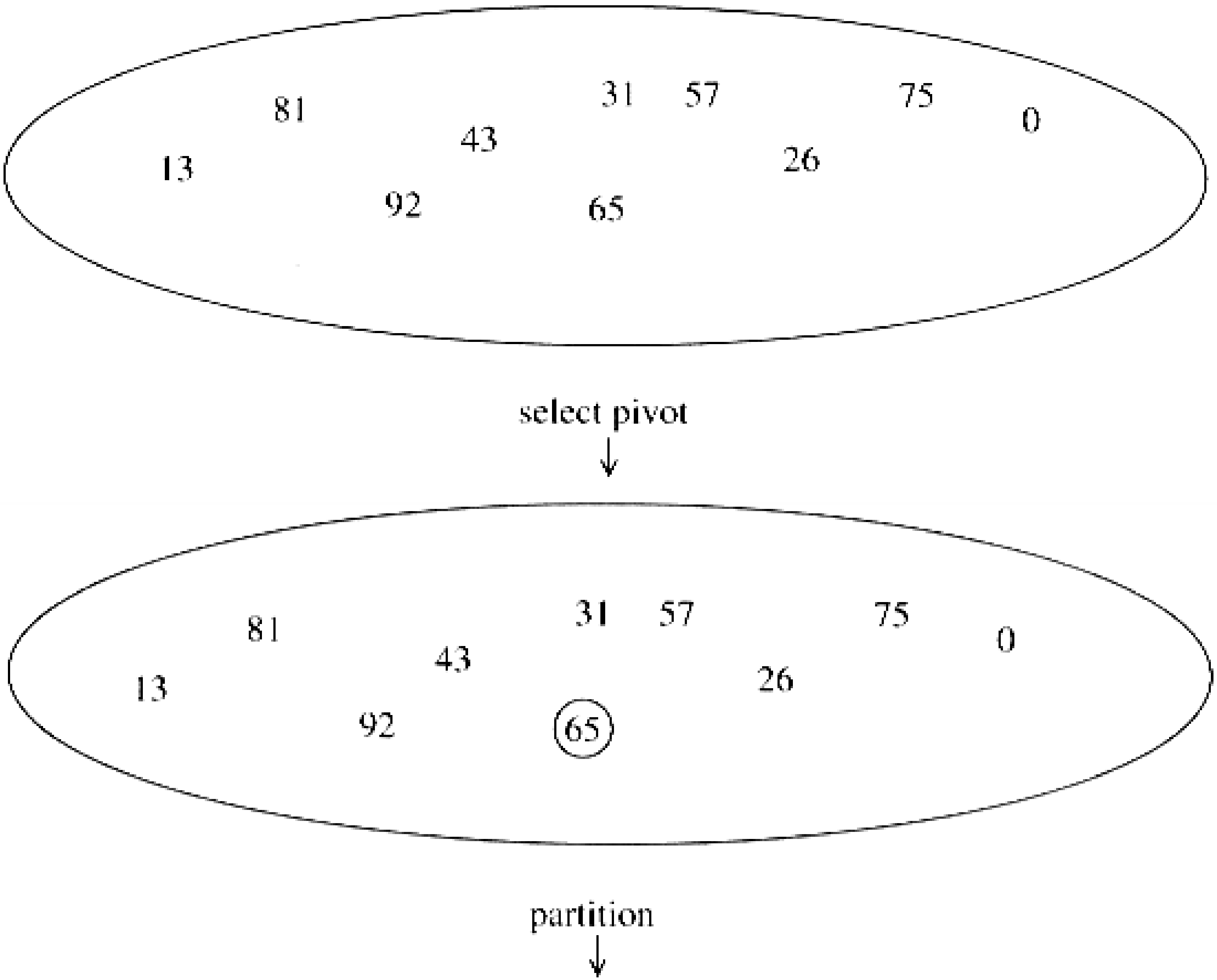
Quick Sort

- Quick Sort is based on the **Divide and Conquer** approach
- Fastest known sorting algorithm in practice
- Average case: **O(N log N)**
- Worst case: **$O(N^2)$**

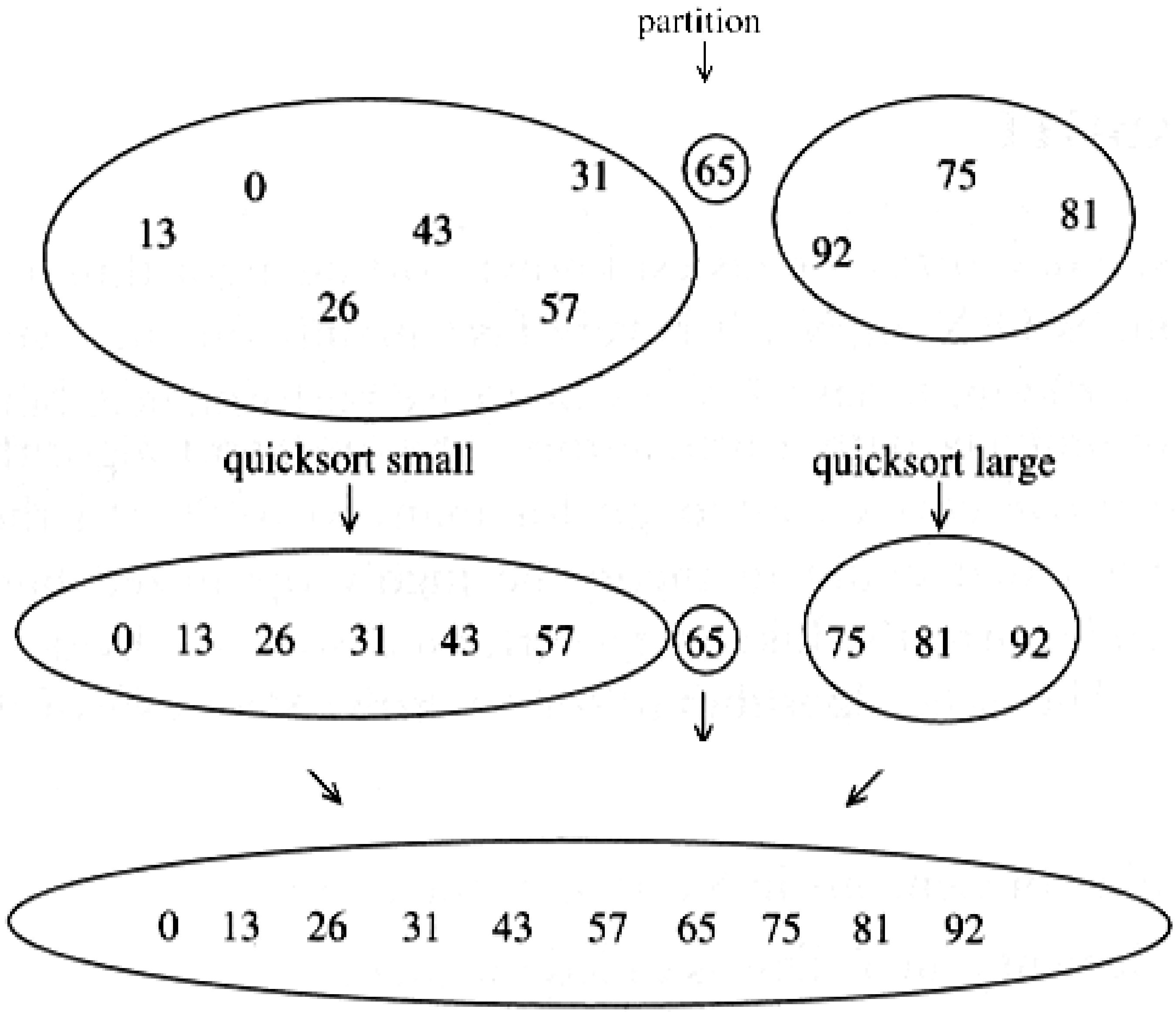
Quick Sort: Main Idea

1. Pick any element **v** called the **pivot** in **S**.
2. Partition the elements in **S** except **v** into two disjoint groups:
   1. $S_1 = \{ x \in S - \{ v \} \mid x \leq v \}$
   2. $S_2 = \{ x \in S - \{ v \} \mid x \geq v \}$
3. $\text{QuickSort}(S_1) + v + \text{QuickSort}(S_2)$

Quick Sort: Example



Example of Quick Sort...



Quick Sort Algorithm

- Array is divided into subarrays by selecting a pivot element(element selected from the array).
- While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
- The left and right subarrays are also divided using the same approach.
- This process continues until each subarray contains a single element.
- At this point, elements are already sorted.

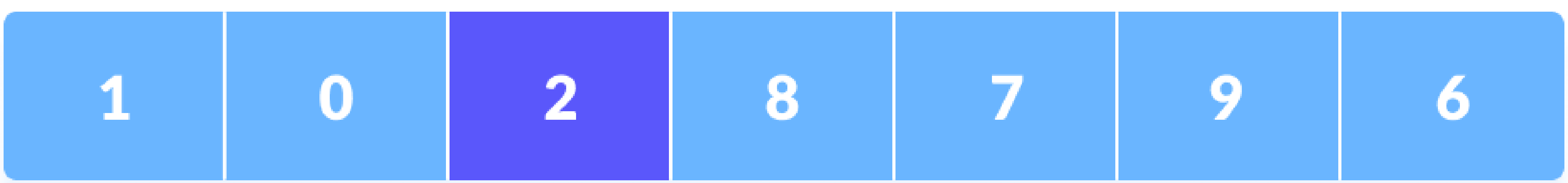
Quick Sort Algorithm

1. **Select the Pivot Element**
   - **There are different variations of quicksort where the pivot element is selected from different positions.**
     - **Rightmost element of the array**
     - **Leftmost element of the array**
     - **Any random element from the array**
     - **Median element of the array**
     - **…**

8	7	6	1	0	9	2

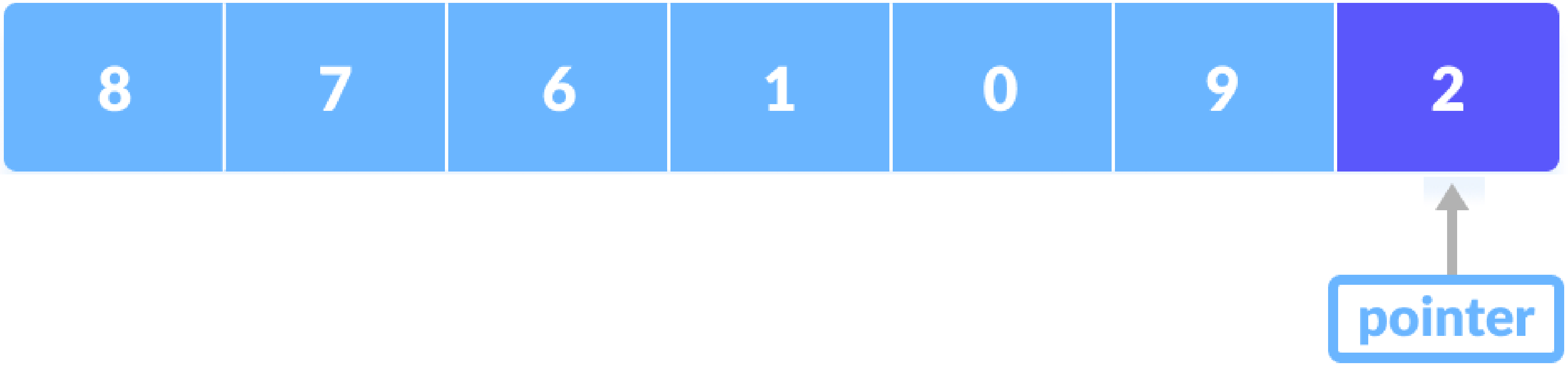
Quick Sort Algorithm

2. Rearrange the Array
   a) Elements of the array are rearranged so that elements that are **smaller** than the **pivot** are put on the **left** and the elements **greater** than the **pivot** are put on the **right**.



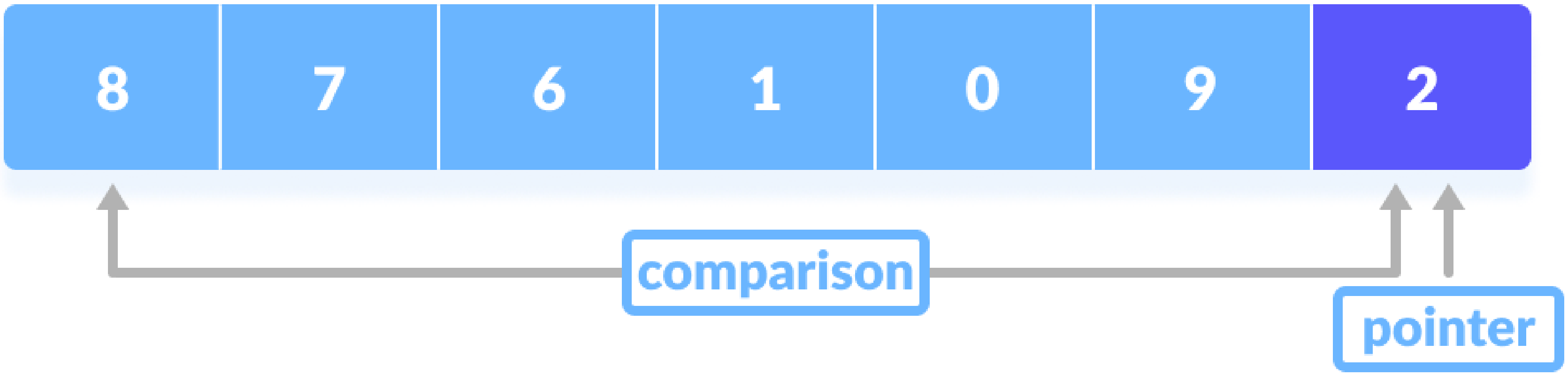
Quick Sort Algorithm

- Select the **Pivot** Element
- A **pointer** is fixed at the **pivot** element.

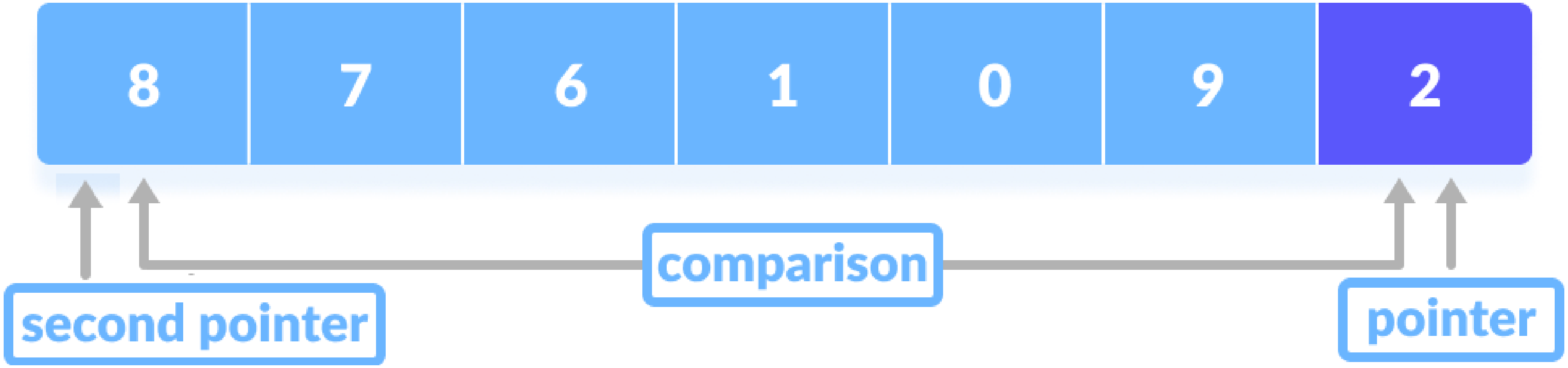


Quick Sort Algorithm

- The pivot element is **compared** with the **elements beginning** from the **first index**.

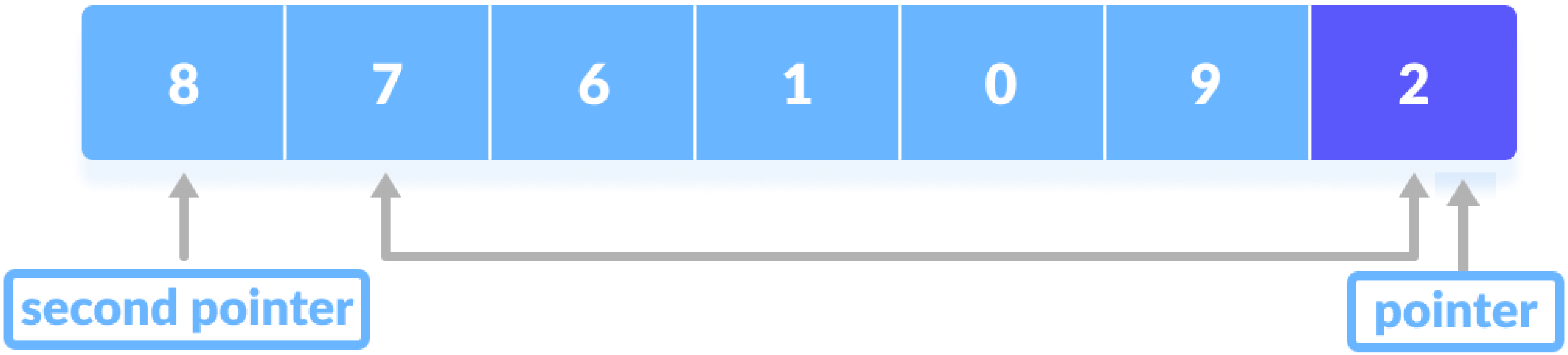


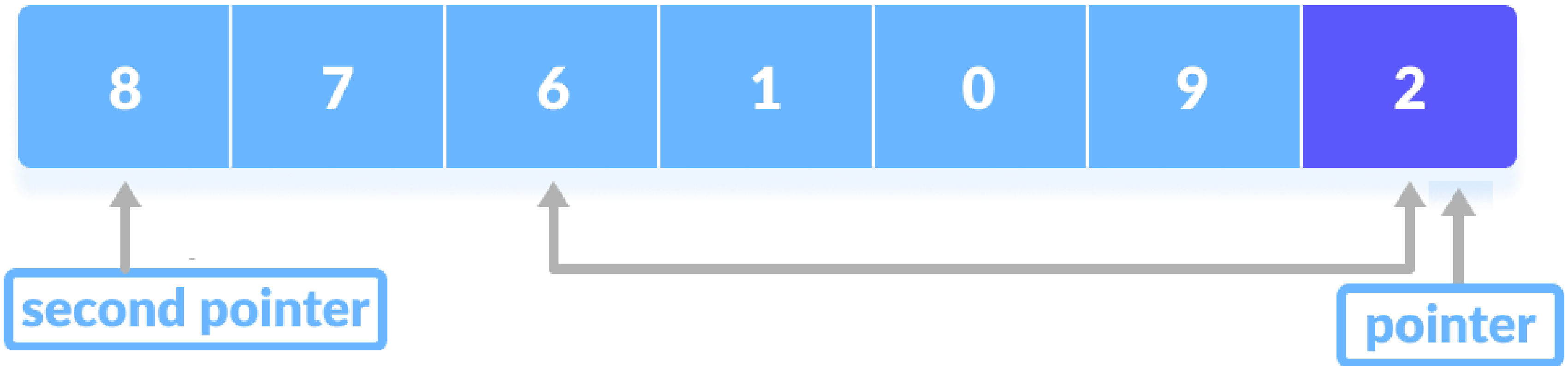
- If the element is **greater** than the **pivot** element, a **second pointer** is set for that element.



Quick Sort Algorithm

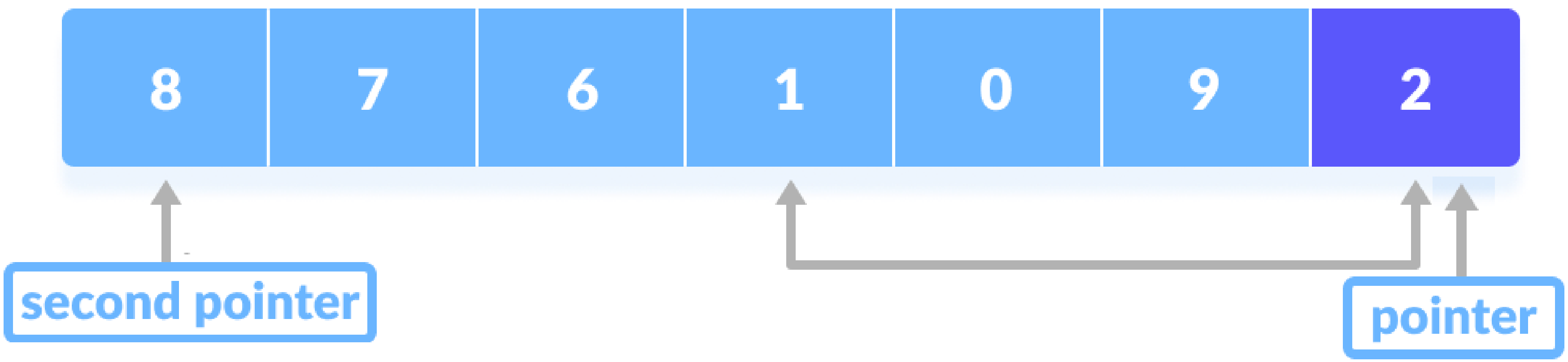
- Now, pivot is compared with other elements.
  - If the element greater than the pivot element, and second pointer is already set then go to the next element.





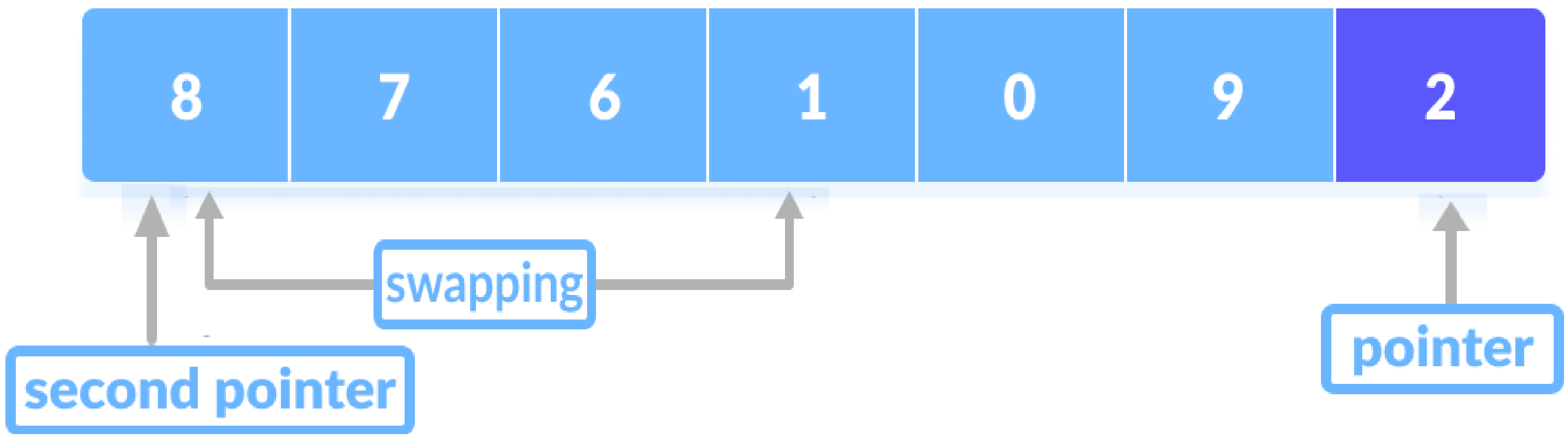
Quick Sort Algorithm

- If an element **smaller** than the **pivot element** is found, then the **smaller element** is **swapped** with the **greater element found earlier.**



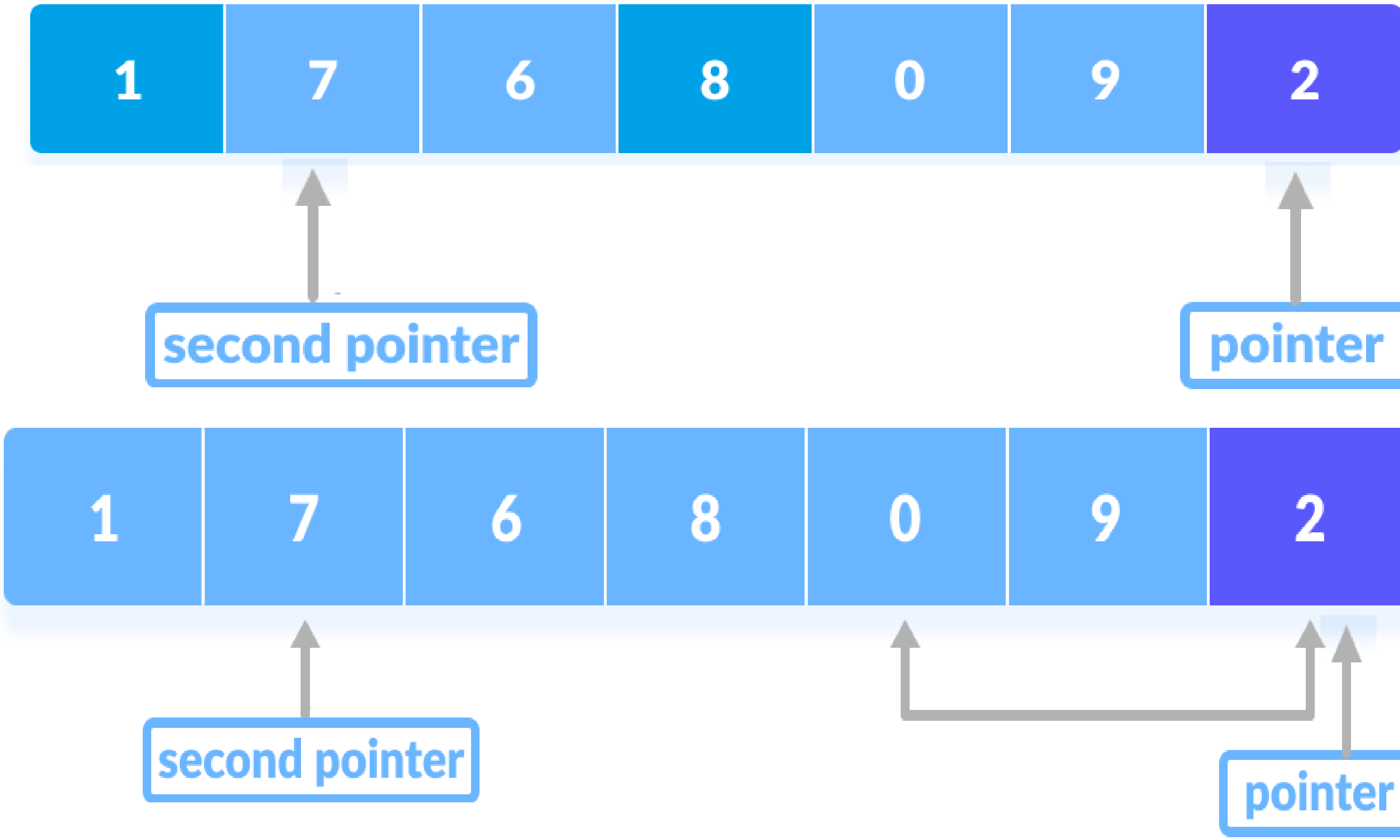
Quick Sort Algorithm

- **Smaller element** is **swapped** with the **greater element found earlier.**



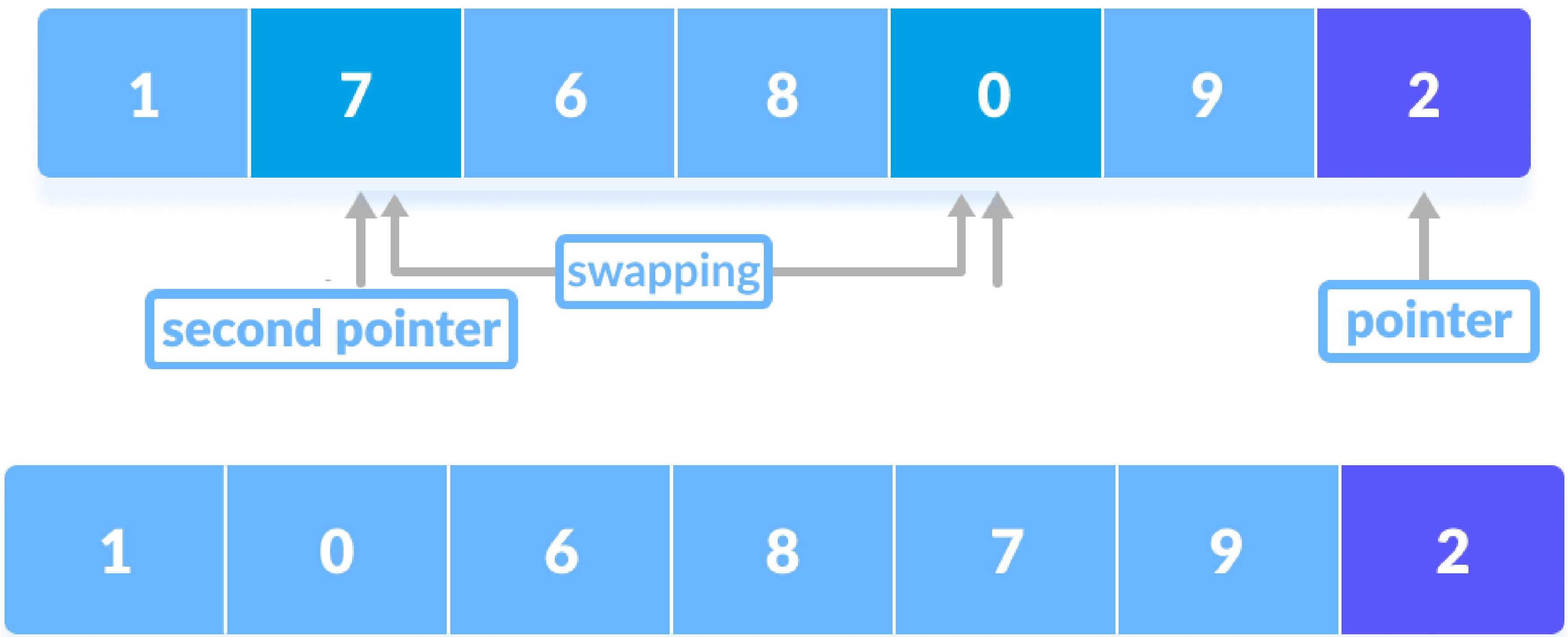
Quick Sort Algorithm

- The process is repeated to **set the next greater element** as the **second pointer**. And, **swap** it with **another smaller element.**



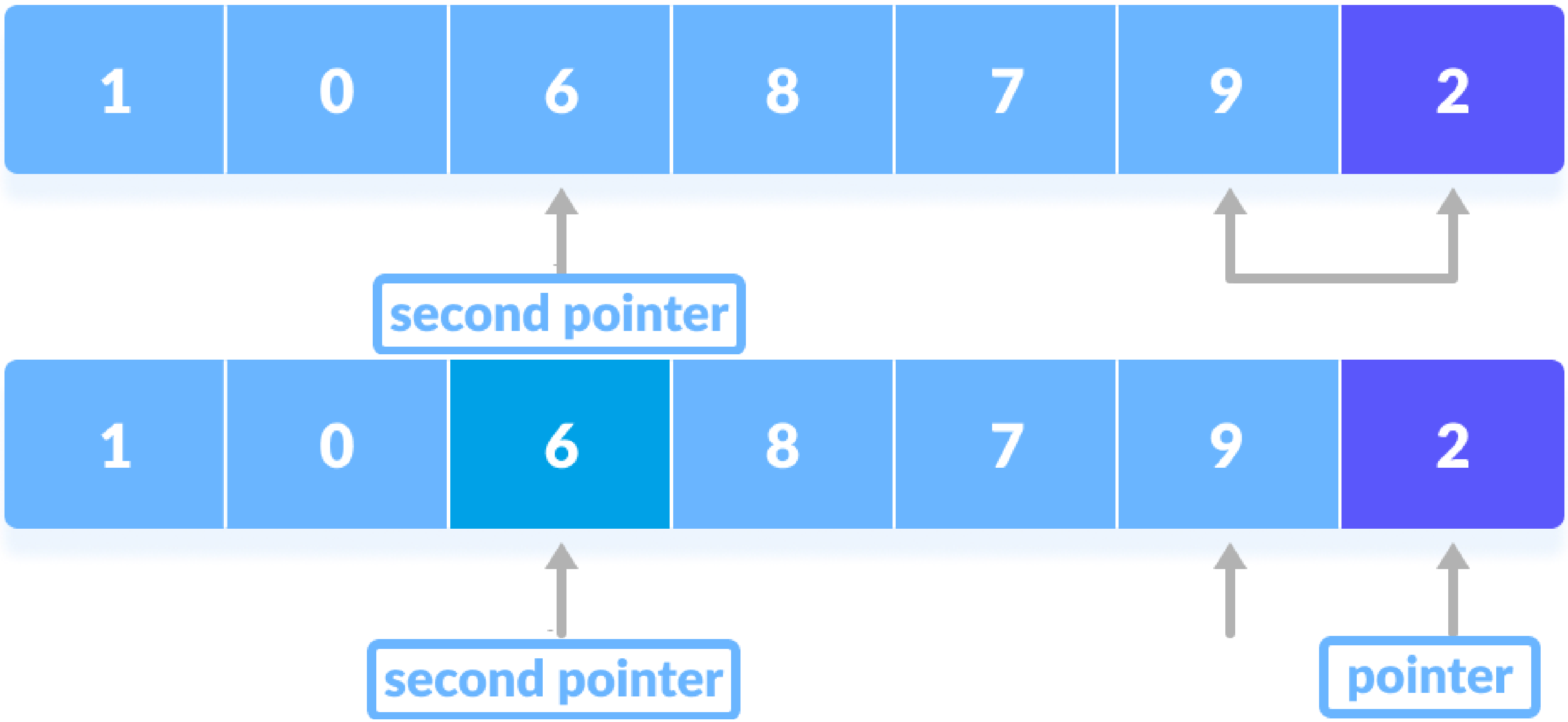
Quick Sort Algorithm

- The process is repeated to **set the next greater element** as the **second pointer**. And, **swap** it with **another smaller element.**

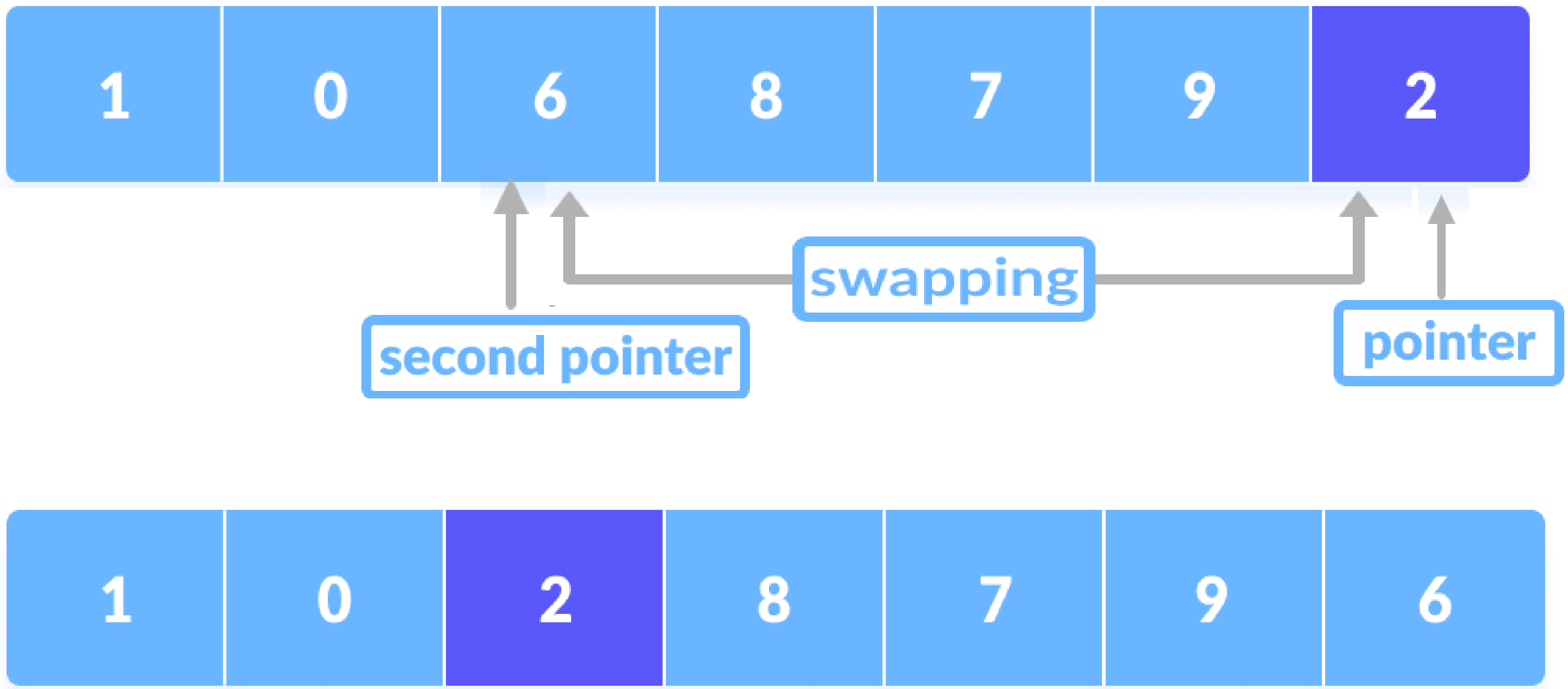


Quick Sort Algorithm

- The process goes on until the **second last** element is reached



- Finally, the pivot element is swapped with the second pointer.



Quick Sort Algorithm

3. Divide Subarrays

Pivot elements are again chosen for the left and the right sub-parts separately and step 2 is repeated.

Select pivot element in each half and put at correct place using recursion

Quick Sort

```
// Quick sort in C
#include <stdio.h>
// function to swap elements
void swap(int *a, int *b)
{     int  t = *a;
      *a = *b;
      *b = t;
}
```

Quick Sort

```
// function to find the partition position
int partition(int array[], int low, int high)
{  // select the rightmost element as pivot
  int pivot = array[high];
   // pointer for greater element
  int i = (low - 1);
```

Quick Sort

```
 // traverse each element of the array
 // compare them with the pivot
 for (int j = low; j < high; j++)
{   if (array[j] <= pivot)
   {  // if element smaller than pivot is found
      // swap it with the greater element
      // pointed by i
     i++;
```

Quick Sort

```
        // swap element at i with element at j
      swap(&array[i], &array[j]);
    }
  }
//swap pivot element with greater element at i
  swap(&array[i + 1], &array[high]);
    // return the partition point
  return (i + 1);
}
```

Quick Sort

```
void quickSort(int array[], int low, int high) {
  if (low < high) {
     // find the pivot element such that
    // elements smaller than pivot are on
    // left of pivot
    // elements greater than pivot are on
    // right of pivot
    int pi = partition(array, low, high);
```

Quick Sort

```
    // recursive call on the left of pivot
    quickSort(array, low, pi - 1);

    // recursive call on the right of pivot
    quickSort(array, pi + 1, high);
  }
}
```

Quick Sort

```
// function to print array elements
void printArray(int array[], int size)
{  for (int i = 0; i < size; ++i) {
    printf("%d  ", array[i]);
  }
  printf("\n");
}
```

Quick Sort

```
int main( )
{  int data[] = {8, 7, 2, 1, 0, 9, 6};
   int n = sizeof(data) / sizeof(data[0]);
   printf("Unsorted Array\n");
   printArray(data, n);
   // perform quicksort on data
   quickSort(data, 0, n - 1);
   printf("Sorted array in ascending order: \n");
   printArray(data, n);
}
```

Quick Sort Algorithm in Python

```
# Quick sort in Python
# function to find the partition position
def partition(array, low, high):
  # choose the rightmost element as pivot
  pivot = array[high]
  # pointer for greater element
  i = low - 1
  # traverse through all elements
  # compare each element with pivot
```

```
for j in range(low, high):
    if array[j] <= pivot:
      # if element smaller than pivot is found
      # swap it with the greater element pointed
      # by i
      i = i + 1
      # swapping element at i with element at j
      (array[i], array[j]) = (array[j], array[i])
```

```
# swap the pivot element with the greater
# element specified by i
 (array[i + 1], array[high]) = (array[high],
                                array[i + 1])
 # return the position from where partition
 # is done
 return i + 1
```

```
# function to perform quicksort
def quickSort(array, low, high):
  if low < high:
    # find pivot element such that element smaller than
    # pivot are on the left element greater than pivot
    # are on the right
    pi = partition(array, low, high)
    # recursive call on the left of pivot
    quickSort(array, low, pi - 1)
    # recursive call on the right of pivot
    quickSort(array, pi + 1, high)
```

```
data = [38,47,942,61,10,90,46,66,53,83,16 ]
print("Unsorted Array")
print(data)
size = len(data)
quickSort(data, 0, size - 1)
print('Sorted Array in Ascending Order:')
print(data)
```

Unsorted Array

[38, 47, 942, 61, 10, 90, 46, 66, 53, 83, 16]

Sorted Array in Ascending Order:

[10, 16, 38, 46, 47, 53, 61, 66, 83, 90, 942]

Analysis of Quick Sort

Worst Case Complexity [Big-O]: $O(n^2)$

- **It occurs when the pivot element picked is either the greatest or the smallest element.**
- **This condition leads to the case in which the pivot element lies in an extreme end of the sorted array.**
- **One sub-array is always empty and another sub-array contains n - 1 elements.**
- **However, the quicksort algorithm has better performance for scattered pivots.**

Best Case Complexity [Big-omega]: O(n*log n)

- **It occurs when the pivot element is always the middle element or near to the middle element.**

Average Case Complexity [Big-theta]: O(n*log n)

- **It occurs when the above conditions do not occur.**

Space Complexity

- **The space complexity for quicksort is O(log n).**

Visual Demo of Quick Sort

https://visualgo.net/en/sorting