Quick Sort:

Divide and conquer algorithm in which each array is divided into sub-arrays, solved separately, then merged.

In each trial run, a pivot is chosen, around which the function will sort the remaining list items. **The goal after each trial run, is to place the pivot in its appropriate position in the list, having all elements less than the pivot on its left-hand side, and the remaining elements on the right-hand side.**

In a nutshell, the main three steps utilized to perform this algorithm are as follows:

1- Choose a pivot

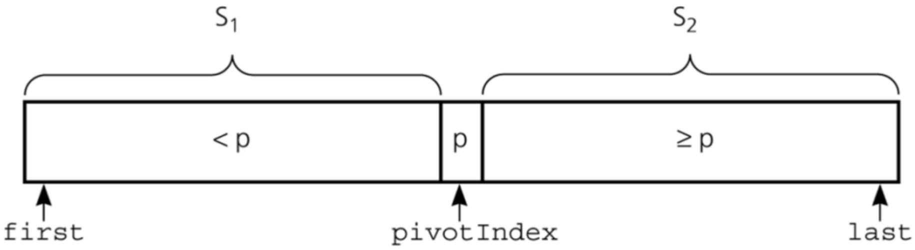
The choice of the pivot can be one of a myriad of choices. This choice heavily impacts the time complexity of the algorithm. It can be the first, last, middle, or a random element.

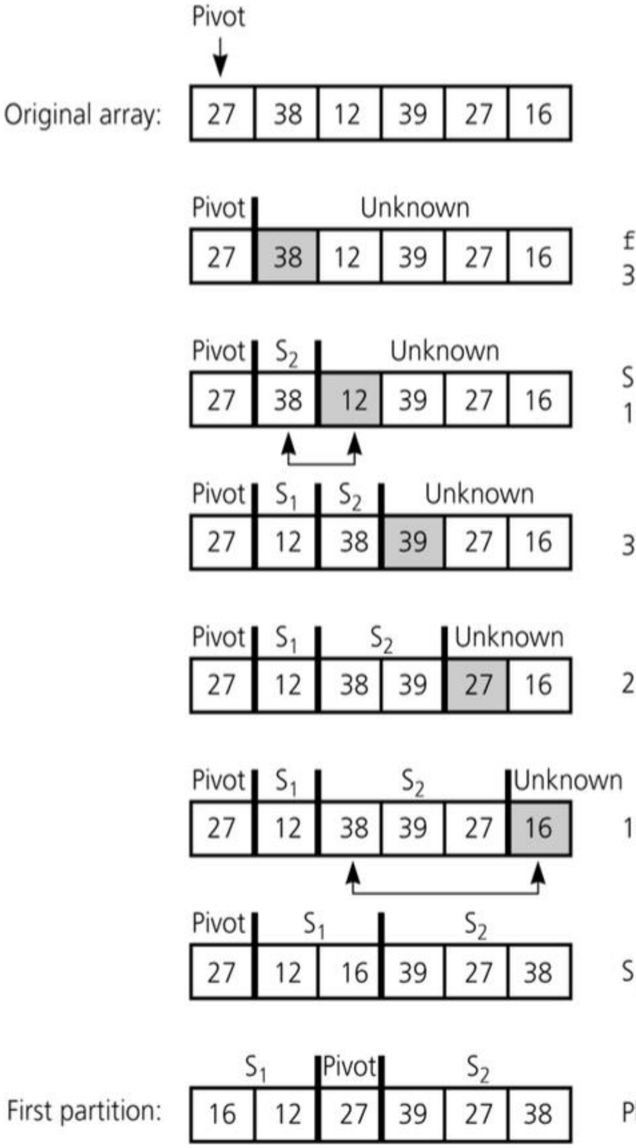
2- Partition the array

This step only ensures that array is properly arranged around the pivot.

3- Recursively call the function

The base case for this algorithm is when there is only one remaining element in the list, in which case it would already be sorted.





`firstUnknown` = 1 (points to 38)
38 belongs in S_2

S_1 is empty;
12 belongs in S_1, so swap 38 and 12

39 belongs in S_2

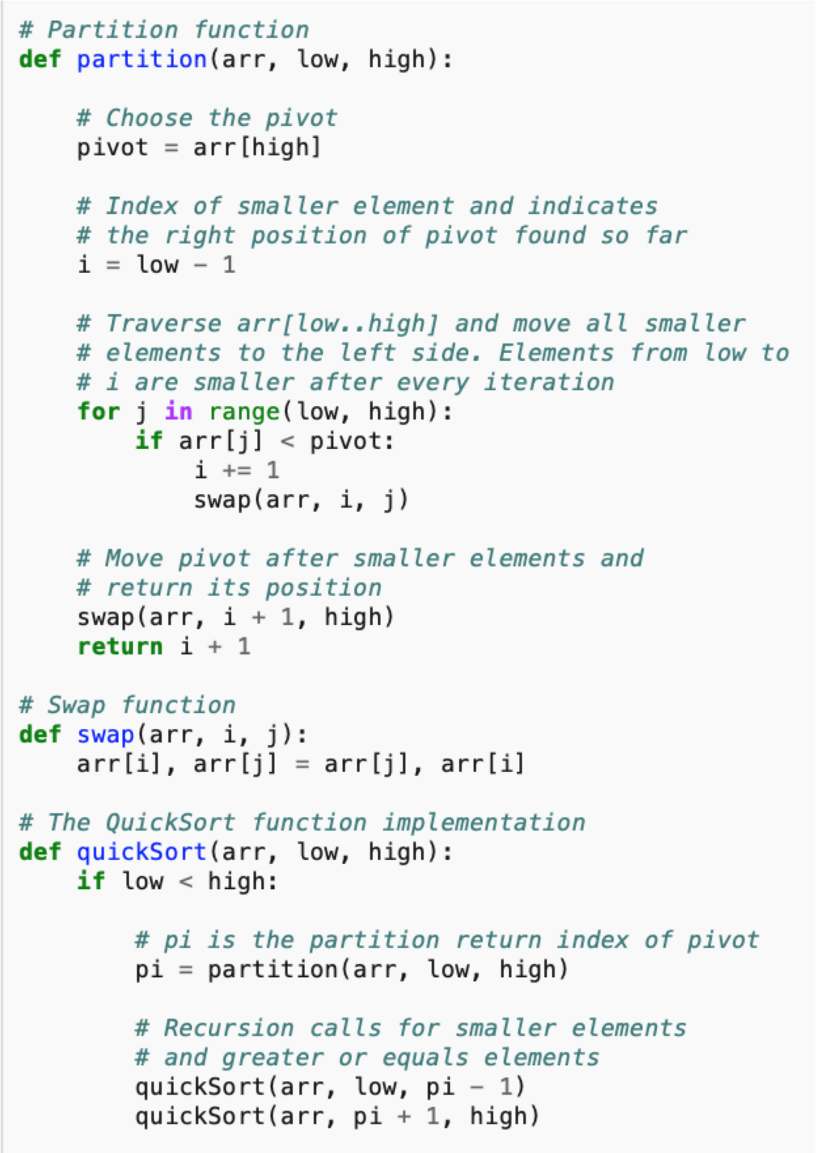
27 belongs in S_2

16 belongs in S_1, so swap 38 and 16

S_1 and S_2 are determined

Place pivot between S_1 and S_2

Algorithm Python Code Snippet:



```python
# Partition function
def partition(arr, low, high):

    # Choose the pivot
    pivot = arr[high]

    # Index of smaller element and indicates
    # the right position of pivot found so far
    i = low - 1

    # Traverse arr[low..high] and move all smaller
    # elements to the left side. Elements from low to
    # i are smaller after every iteration
    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            swap(arr, i, j)

    # Move pivot after smaller elements and
    # return its position
    swap(arr, i + 1, high)
    return i + 1

# Swap function
def swap(arr, i, j):
    arr[i], arr[j] = arr[j], arr[i]

# The QuickSort function implementation
def quickSort(arr, low, high):
    if low < high:

        # pi is the partition return index of pivot
        pi = partition(arr, low, high)

        # Recursion calls for smaller elements
        # and greater or equals elements
        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)
```

Time Complexity:

$$\boldsymbol{O(n * logn)}$$

Algorithm Complexity Explained:

Best Case

Random element chosen is the median of the list $\boldsymbol{O(n * logn)}$

Worst Case → Sorted Array in which first element is chosen as pivot

Inner loop → Executed i-1 times

Number of swaps: n*(n+1)/2-1 $\boldsymbol{O(n^2)}$

Number of comparisons: 1+2+...+n-1 = n*(n-1)/2 $\boldsymbol{O(n^2)}$

Average Case → $\boldsymbol{O(n * logn)}$