Practical 08
Advanced Sort – Quick Sort

1. Quick Sort

***Background:** Quick Sort (sometimes called partition-exchange sort) is an efficient sorting algorithm, serving as a systematic method for placing the elements of a random access file or an array in order. Developed by British computer scientist Tony Hoare in 1959 and published in 1961, it is still a commonly used algorithm for sorting. When implemented well, it can be about two or three times faster than its main competitors, Merge Sort and Heapsort.*

Quick Sort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation is defined. In efficient implementations it is not a stable sort, meaning that the relative order of equal sort items is not preserved. Quick Sort can operate in-place on an array, requiring small additional amounts of memory to perform the sorting.

(Reference: https://en.wikipedia.org/wiki/Quicksort)

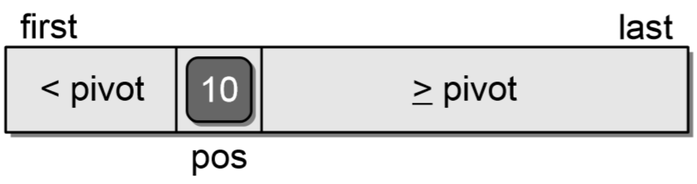
The following code is a partial in-place implementation of the Quick Sort algorithm using recursion:

```python
# Sorts a Python list in ascending order using the quick sort
# algorithm
def quickSort( theList ):
    n = len( theList )
    recQuickSort( theList, 0, n-1 )


# The recursive "in-place" implementation
def recQuickSort( theList, first, last ):
    # Check the base case (range is trivially sorted)
    if first >= last:
        return
    else:
        # Partition the list and obtain the pivot position
        pos = partitionSeq( theList, first, last )

        # Repeat the process on the two sublists
        recQuickSort( theList, first, pos - 1 )
        recQuickSort( theList, pos + 1, last )
```

The implementation of the `partitionSeq()` function is not included above. Basically, the function re-arranges the items within the sequence by correctly positioning the pivot value within the sequence and placing all the items that are less than the pivot to the left and all the items that are greater or equal to the right as shown below:



Refer to the lecture notes for Topic 06 – Advanced Sort (slide no. 33 to 37) to see an explanation of how `partitionSeq()` works. Then, complete the implementation of the `partitionSeq()` given below, by providing the rest of the required code.

```
# Partitions the list using the first key as the pivot
def partitionSeq(theList, first, last):
    # Save a copy of the pivot value.
    pivot = theList[first]   # first element of range is pivot

    # Find the pivot position and move the elements around it
    left  = ___________         # will scan rightward
    right = ___________         # will scan leftward
    while left <= right:
        # Scan until reaches value equal or larger than pivot (or
        # right marker)
        while left <= right and theList[left] < pivot:
            ___________

        # Scan until reaches value equal or smaller than pivot (or
        # left marker)
        while left <= right and theList[right] > pivot:
            ___________

        # Scans did not strictly cross
        if left <= right:
            # swap values
            _________________________________________________________

            # Shrink range (Recursion: Progress towards base case)
            left += 1
            ___________

    # Put the pivot in the proper position (marked by the right index)
    _________________________________________________________

    # Return the index position of the pivot value.
    return ____________


# Test code
list_of_numbers = [12, 7, 9, 24, 7, 29, 5, 3, 11, 7]

print('Input List:', list_of_numbers)
quickSort(list_of_numbers)
print('Sorted List:', list_of_numbers)
```

Sample Output:

```
Input List: [12, 7, 9, 24, 7, 29, 5, 3, 11, 7]
Sorted List: [3, 5, 7, 7, 7, 9, 11, 12, 24, 29]

Process finished with exit code 0
```

- End of Practical --