14. Sorting Algorithms :: Quick Sort :: Introduction

Although merge sort is significantly more time-efficient than selection and insertion sort, a disadvantage is that it requires the creation of temporary lists containing copies of the elements in the list being sorted:

```
-- Otherwise, split the list into two halves: a left half and a right half.
-- Recursively merge sort each half.
$list_L$ ← list[0..list.size / 2 - 1]
$list_R$ ← list[list.size / 2..list.size - 1]
```

Quick sort is another sorting algorithm that also has $O(n\ lg\ n)$ complexity, but does not require temporary lists, i.e., quick sort works by reordering the elements within the list itself.

Similar to merge sort—where the primary operation is merging sorted sublists that were formed by splitting a larger list into two halves—quick sort has a primary operation which **partitions** the list being sorted into two smaller sublists. For example, consider this list:

$list = \{4, 2, 7, 3, 5, 13, 11, 8, 6, 2\}$

The goal of the partitioning operation is to split *list* into two lists we shall refer to as $list_\mathrm{L}$ and $list_\mathrm{R}$, where all of the elements in $list_\mathrm{L}$ are less than the elements in $list_\mathrm{R}$. For example, one partitioning of *list* could be:

$list_\mathrm{L} = \{2, 2, 3\}$ $list_\mathrm{R} = \{7, 5, 13, 11, 8, 6, 4\}$

Of course, there are many such partitions, for example, another one would be:

$list_\mathrm{L} = \{4, 2, 2, 3, 5\}$ $list_\mathrm{R} = \{13, 11, 8, 6, 7\}$

14. Sorting Algorithms :: Quick Sort :: The Pivot Element

Note one difference between merge sort and quick sort: in merge sort, we split the larger list into two smaller lists which are roughly the same size, i.e., they will each contain exactly $n/2$ elements if n is even, or the left sublist will have one fewer element than the right sublist when n is odd.

In quick sort, depending on how we partition, it is possible to obtain a left sublist which has two elements, while the right sublist ends up with ten elements. That is fine, as long as the rule which requires that all of the elements in the left sublist be less than the elements in the right sublist is satisfied.

So how do we partition? The strategy is to choose some element in the list as the **pivot** element. Partitioning then proceeds to form sublists $list_\text{L}$ and $list_\text{R}$ such that all of the elements in $list_\text{L}$ are less than than the elements in $list_\text{R}$.

The next question, then, is which element do we choose as *pivot*? It turns out for quick sort that this is a vitally important choice because a bad choice can cause quick sort's time complexity to degenerate to $O(n^2)$ which is no better than insertion and selection sort.

For this reason there are various strategies for selecting the pivot; the simplest one is to simply choose the first element in the list.

Once *pivot* is selected, the partitioning procedure reorders elements in the list to form the two sublists such that the elements in the left sublist are all less than or equal to *pivot* and the ones in the right sublist are greater than *pivot*.

14. Sorting Algorithms :: Quick Sort :: Partitioning Pseudocode

Here is the pseudocode for the partition procedure. Note that it partitions a sublist of *list*, i.e., it partitions *list*[*fromIndex* .. *toIndex*].

```
Method partition(In: List<T> list; In: fromIndex; In: toIndex) → Integer
    -- Choose the first element of list[fromIndex..toIndex] to be the pivot
    pivot ← list[fromIndex]

    -- leftIndex and rightIndex are indices into list that move toward each other as
    -- the partitioning proceeds. leftIndex starts at the left end of the sublist and
    -- rightIndex starts at the right end.
    leftIndex ← fromIndex - 1, rightIndex ← toIndex + 1

    -- Partitioning ends when leftIndex and rightIndex "cross".
    While leftIndex < rightIndex Do

        -- Move leftIndex rightward until an element that is greater than or equal
        -- to pivot is reached.
        leftIndex++
        While list[leftIndex] < pivot Do: leftIndex++

        -- Move rightIndex leftward until an element that is less than or equal to
        -- pivot is reached.
        rightIndex--
        While list[rightIndex] > pivot Do: rightIndex--
```

14. Sorting Algorithms :: Quick Sort :: Partitioning Pseudocode

```
        -- When the above while loops terminate, if leftIndex and rightIndex have not
        -- "crossed", then leftIndex will be the index of an element that belongs in
        -- the right half of the partition and rightIndex will be the index of an
        -- element that belongs in the left half. So we swap the elements.
        If leftIndex < rightIndex Then: swap(list, leftIndex, rightIndex)
    End While
    -- When the while loop terminates, the elements in list[fromIndex..
    -- rightIndex - 1] are all less than the elements in list[rightIndex..toIndex].
    -- We have now essentially partitioned list into a left half and a right half.
    -- We return rightIndex which will be used in the main quick sort method.
    Return rightIndex
End Method partition
```