Recursive sorting algorithms

Quicksort

Quicksort is a divide and conquer algorithm. Quicksort first divides a large list into two smaller sub-lists: the low elements and the high elements. Quicksort can then recursively sort the sub-lists.

The steps are:

1. Pick an element, called a *pivot*, from the list.
2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

The base case of the recursion are lists of size zero or one, which never need to be sorted.

Quicksort

- The "worker method" of quicksort is to process partitions
  - first, a pivot is selected (e.g. the middle element)
  - this pivot is moved to safety (i.e. to the far right)
  - all values that are larger than this pivot value are moved to the left
  - all values that are smaller than this pivot value are moved to the right
  - the pivot is then moved back into place
- Processing a partition leaves the selected pivot in its final sorted position
- The newly created partitions are then processed recursively using quicksort
- Partitions are processed from left to right
  - processing the first partition produces a left and a right partition
  - processing the left partition produces a left-left and a left-right partition
  - etc
  - left partitions are processed until completion (i.e. partitions of one element)
- There can be a large number of (unnecessary) pivot swaps for small partitions (e.g. 2 or 3 elements). These inefficient swaps can be optimized by using Insertion Sort.

Quicksort – pseudocode part 1

```
quicksort(array, left, right):
        if left < right:
                pivot := partition(array, left, right)
                quicksort(array, left, pivot - 1)
                quicksort(array, pivot + 1, right)
```

Quicksort : Pseudocode part 2

// left is the index of the leftmost element of the subarray

// right is the index of the rightmost element of the subarray (inclusive)

// number of elements in subarray = right-left+1

```
partition(array, left, right)
        pivotIndex := choosePivot(array, left, right)
        pivotValue := array[pivotIndex]
        swap array[pivotIndex] and array[right]
        storeIndex := left
        for i from left to right - 1
                if array[i] < pivotValue
                        swap array[i] and array[storeIndex]
                        storeIndex := storeIndex + 1
        swap array[storeIndex] and array[right]
                // Move pivot to its final place
        return storeIndex
```

Merge Sort

Conceptually, a merge sort works as follows

- Divide the unsorted list into n sublists, each containing 1 element (a list of 1 element is considered sorted).
- Repeatedly merge sublists to produce new sublists until there is only 1 sublist remaining. This will be the sorted list.

Merge Sort pseudocode part 1

```
private static List<Integer> merge_sort(List<Integer> arr) {
    if (arr.size() <= 1) { // one or zero length lists are already sorted
        return arr;
    }
    //Split the list into a left and a right part
    List<Integer> left = new LinkedList<>();
    List<Integer> right = new LinkedList<>();
    split_list(arr, left, right);

    // sort the two parts
    left = merge_sort(left);
    right = merge_sort(right);

    // merge them again
    arr = merge_list(left, right);
    return arr;
}
```

Merge Sort pseudocode part 2

```
function merge(list left, list right)
    var list result
    while something left in both
        if first of left < first of right
                insert first of left into result
                remove first from left
        else
                insert first of right into result
                remove first from right
        end if
    end while
    insert the remaining elements from left or right into result
    return result
```