Quick-Sort

Quick-Sort (§ 10.2)

Quick-sort is a randomized sorting algorithm based on the divide- and conquer paradigm:

- **Divide:** pick a random element \( x \) (called pivot) and partition \( S \) into
  - \( L \) elements less than \( x \)
  - \( E \) elements equal \( x \)
  - \( G \) elements greater than \( x \)
- **Recur:** sort \( L \) and \( G \)
- **Conquer:** join \( L \), \( E \) and \( G \)

Partition

We partition an input sequence as follows:

- We remove, in turn, each element \( y \) from \( S \) and
- We insert \( y \) into \( L \), \( E \) or \( G \), depending on the result of the comparison with the pivot \( x \)
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes \( O(1) \) time
- Thus, the partition step of quick-sort takes \( O(n) \) time

Algorithm `partition(S, p)`

**Input** sequence \( S \), position \( p \) of pivot

**Output** subsequences \( L \), \( E \), \( G \) of the elements of \( S \) less than, equal to, or greater than the pivot, resp.

1. \( L, E, G \leftarrow \) empty sequences
2. \( x \leftarrow S\.remove(p) \)
3. while \( \neg S\.isEmpty() \)
   1. \( y \leftarrow S\.remove(S\.first()) \)
   2. if \( y < x \)
      1. \( L\.insertLast(y) \)
   3. else if \( y = x \)
      1. \( E\.insertLast(y) \)
   4. else \( y > x \)
      1. \( G\.insertLast(y) \)
4. return \( L, E, G \)

Quick-Sort Tree

An execution of quick-sort is depicted by a binary tree

- Each node represents a recursive call of quick-sort and stores
  - Unsorted sequence before the execution and its pivot
  - Sorted sequence at the end of the execution
- The root is the initial call
- The leaves are calls on subsequences of size 0 or 1
Execution Example

Pivot selection

7 2 9 4 3 7 6 1

Partition, recursive call, pivot selection

2 4 3 1

Recursive call, ..., base case, join

2 4 3 1 → 1 2 3 4

1 → 1

4 3 → 3 4

4 → 4
Execution Example (cont.)

- Recursive call, pivot selection
- Partition, ..., recursive call, base case
- Join, join

Worst-case Running Time

- The worst case for quick-sort occurs when the pivot is the unique minimum or maximum element.
- One of $L$ and $G$ has size $n - 1$ and the other has size 0.
- The running time is proportional to the sum $n + (n - 1) + \ldots + 2 + 1$.
- Thus, the worst-case running time of quick-sort is $O(n^2)$. 

Depth time

<table>
<thead>
<tr>
<th>Depth</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$n$</td>
</tr>
<tr>
<td>1</td>
<td>$n-1$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>$n-1$</td>
<td></td>
</tr>
</tbody>
</table>
Expected Running Time

Consider a recursive call of quick-sort on a sequence of size \( s \).

- **Good call:** the sizes of \( L \) and \( G \) are each less than \( 3s/4 \).
- **Bad call:** one of \( L \) and \( G \) has size greater than \( 3s/4 \).

A call is good with probability \( 1/2 \).

1/2 of the possible pivots cause good calls.

<table>
<thead>
<tr>
<th>Good call</th>
<th>Bad call</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ 1 2 3 4 5 6 7 8 9 10 11 12 ]</td>
<td>[ 13 14 15 16 ]</td>
</tr>
</tbody>
</table>

In-Place Quick-Sort

Quick-sort can be implemented to run in-place.

In the partition step, we use replace operations to rearrange the elements of the input sequence such that:

- the elements less than the pivot have rank less than \( h \).
- the elements equal to the pivot have rank between \( h \) and \( k \).
- the elements greater than the pivot have rank greater than \( k \).

The recursive calls consider:

- elements with rank less than \( h \).
- elements with rank greater than \( k \).

**Algorithm inPlaceQuickSort(S, I, r)**

- **Input** sequence \( S \), ranks \( I \) and \( r \).
- **Output** sequence \( S \) with the elements of rank between \( I \) and \( r \) rearranged in increasing order.
  - if \( I \geq r \)
    - return
  - \( i \leftarrow \text{a random integer between} \; I \) and \( r \)
  - \( x \leftarrow S.\text{elemAtRank}(i) \)
  - \( (h, k) \leftarrow \text{inPlacePartition}(x) \)
  - \( \text{inPlaceQuickSort}(S, I, h - 1) \)
  - \( \text{inPlaceQuickSort}(S, k + 1, r) \)

In-Place Partitioning

Perform the partition using two indices to split \( S \) into \( L \) and \( E \cup G \) (a similar method can split \( E \cup G \) into \( E \) and \( G \)).

Repeat until \( j \) and \( k \) cross:

- Scan \( j \) to the right until finding an element \( \geq x \).
- Scan \( k \) to the left until finding an element \( < x \).
- Swap elements at indices \( j \) and \( k \).

\[ 3 \; 2 \; 5 \; 1 \; 0 \; 7 \; 3 \; 5 \; 9 \; 2 \; 7 \; 9 \; 8 \; 9 \; 7 \; 6 \; 9 \] (pivot = 6)
## Summary of Sorting Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection sort</td>
<td>$O(n^2)$</td>
<td>in-place, slow (good for small inputs)</td>
</tr>
<tr>
<td>insertion sort</td>
<td>$O(n^2)$</td>
<td>in-place, slow (good for small inputs)</td>
</tr>
<tr>
<td>quick sort</td>
<td>$O(n \log n)$ expected</td>
<td>in-place, randomized, fastest (good for large inputs)</td>
</tr>
<tr>
<td>heap sort</td>
<td>$O(n \log n)$</td>
<td>in-place, fast (good for large inputs)</td>
</tr>
<tr>
<td>merge-sort</td>
<td>$O(n \log n)$</td>
<td>sequential data access, fast (good for huge inputs)</td>
</tr>
</tbody>
</table>

### Java Implementation

```java
public static void quickSort(Object[] S, Comparator c) {
    if (S.length < 2) return; // the array is already sorted in this case
    quickSortStep(S, c, 0, S.length-1); // recursive sort method
}

private static void quickSortStep(Object[] S, Comparator c, int leftBound, int rightBound) {
    if (leftBound >= rightBound) return; // the indices have crossed
    Object temp; // temp object used for swapping
    Object pivot = S[rightBound];
    int leftIndex = leftBound; // will scan rightward
    int rightIndex = rightBound-1; // will scan leftward
    while (leftIndex <= rightIndex) { // scan right until larger than the pivot
        while ( (leftIndex <= rightIndex) && (c.compare(S[leftIndex], pivot)<=0) )
            leftIndex++;
        while ( (rightIndex >= leftIndex) && (c.compare(S[rightIndex], pivot)>=0))
            rightIndex--;
        if (leftIndex < rightIndex) { // both elements were found
            temp = S[rightIndex];
            S[rightIndex] = S[leftIndex]; // swap these elements
            S[leftIndex] = temp;
        }
    }
    temp = S[rightBound]; // swap pivot with the element at leftIndex
    S[rightBound] = S[leftIndex];
    S[leftIndex] = temp; // the pivot is now at leftIndex, so recur
    quickSortStep(S, c, leftBound, leftIndex-1, rightBound);
}
```

Only works for distinct elements.