Quick-Sort

Outline and Reading
- Quick-sort (§4.3)
  - Algorithm
  - Partition step
  - Quick-sort tree
  - Execution example
- Analysis of quick-sort (4.3.1)
- In-place quick-sort (§4.8)
- Summary of sorting algorithms

Quick-Sort
- 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

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

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.

while \(-\neg S.isEmpty()\)
  \( y \leftarrow S.remove(S.first())\)
  if \( y < x \)
    \( L.insertLast(y)\)
  else if \( y = x \)
    \( E.insertLast(y)\)
  else \( G.insertLast(y)\)
return \( L \), \( E \), \( G \)
Execution Example (cont.)
- Partition, recursive call, pivot selection

\[
\begin{array}{c}
2 & 4 & 3 & 1 \\
\end{array}
\]

\[
\begin{array}{c}
7 & 2 & 9 & 4 & 3 & 7 & 6 & 1 \\
\end{array}
\]

Execution Example (cont.)
- Partition, recursive call, base case

\[
\begin{array}{c}
2 & 4 & 3 & 1 \\
\end{array}
\]

\[
\begin{array}{c}
7 & 2 & 9 & 4 & 3 & 7 & 6 & 1 \\
\end{array}
\]

Execution Example (cont.)
- Recursive call, ..., base case, join

\[
\begin{array}{c}
2 & 4 & 3 & 1 \rightarrow 1 & 2 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
7 & 2 & 9 & 4 & 3 & 7 & 6 & 1 \\
\end{array}
\]

Execution Example (cont.)
- Recursive call, pivot selection

\[
\begin{array}{c}
2 & 4 & 3 & 1 \rightarrow 1 & 2 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
7 & 2 & 9 & 4 & 3 & 7 & 6 & 1 \\
\end{array}
\]

Execution Example (cont.)
- Partition, ..., recursive call, base case

\[
\begin{array}{c}
2 & 4 & 3 & 1 \rightarrow 1 & 2 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
7 & 2 & 9 & 4 & 3 & 7 & 6 & 1 \\
\end{array}
\]

Execution Example (cont.)
- Join, join

\[
\begin{array}{c}
2 & 4 & 3 & 1 \rightarrow 1 & 2 & 3 & 4 \\
\end{array}
\]

\[
\begin{array}{c}
7 & 2 & 9 & 4 & 3 & 7 & 6 & 1 \\
\end{array}
\]
**Expected Running Time**

- Consider a recursive call of quick-sort on a sequence of size $n$.
  - **Good call:** the sizes of $L$ and $G$ are each less than $3n/4$.
  - **Bad call:** one of $L$ and $G$ has size greater than $3n/4$.

**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 $a$.
  - the elements equal to the pivot have rank between $a$ and $b$.
  - the elements greater than the pivot have rank greater than $b$.
- The recursive calls consider:
  - elements with rank less than $a$.
  - elements with rank greater than $b$.

**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>$O(n \log n)$</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>