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 to $x$
  - $G$: elements greater than $x$
- Recur: sort $L$ and $G$
- Conquer: join $L$, $E$, and $G$

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

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. $L$, $E$, $G$ ← empty sequences
2. $x$ ← $S$.remove($p$)
3. while ¬$S$.isEmpty
   1. $y$ ← $S$.remove($S$.first())
   2. if $y < x$ $L$.insertLast($y$)
   3. else if $y = x$ $E$.insertLast($y$)
   4. else $G$.insertLast($y$)
5. 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

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
Execution Example (cont.)
Partition, recursive call, pivot selection

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

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

Execution Example (cont.)
Recursive call, pivot selection

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

Execution Example (cont.)
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 \( O(n^2) \).

Expected Running Time

- Consider a recursive call of quick-sort on a sequence of size \( x \).
  - Good call: the sizes of \( L \) and \( G \) are each less than \( \frac{3}{4}n \).
  - Bad call: one of \( L \) and \( G \) has size greater than \( \frac{3}{4}n \).

A call is good with probability \( \frac{1}{2} \).

Probabilistic Fact: The expected number of coin tosses required in order to get a head is \( 2^k \).

Hence, for a node of depth \( i \), we expect that
- \( \frac{i}{2} \) parent nodes are associated with good calls
- the size of the input sequence for the current call is at most \( \frac{1}{4}n \).

Thus, the expected running time of quick-sort is \( O(n \log n) \).

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 \( k \).
- the elements equal to the pivot have rank between \( k \) and \( h \).
- the elements greater than the pivot have rank greater than \( h \).
- The recursive calls consider
  - elements with rank less than \( k \).
  - elements with rank greater than \( h \).

Algorithm inPlaceQuickSort(S, l, r)

Input: sequence \( S \), ranks \( l \) and \( r \).
Output: sequence \( S \) with the elements of rank between \( l \) and \( r \) rearranged in increasing order.

if \( l \geq r \)
return

\( i \leftarrow \) a random integer between \( l \) and \( r \)
\( x \leftarrow S.eleRank(i) \)

\( (h, k) \leftarrow \) inPlacePartition\( (x) \)

inPlaceQuickSort\( (S, l, h - 1) \)

inPlaceQuickSort\( (S, k + 1, r) \)

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>