Merge Sort

Outline and Reading

- Divide-and-conquer paradigm (§4.1.1)
- Merge-sort (§4.1.1)
  - Algorithm
  - Merging two sorted sequences
  - Merge-sort tree
  - Execution example
  - Analysis
- Generic merging and set operations (§4.2.1)
- Summary of sorting algorithms (§4.2.1)

Divide-and-Conquer

- Divide-and-conquer is a general algorithm design paradigm:
  - Divide: divide the input data into two disjoint subsets $S_1$ and $S_2$
  - Recur: solve the subproblems associated with $S_1$ and $S_2$.
  - Conquer: combine the solutions for $S_1$ and $S_2$ into a solution for $S$.
- The base case for the recursion are subproblems of size 0 or 1.

Merge-Sort

- Merge-sort on an input sequence $S$ with $n$ elements consists of three steps:
  - Divide: partition $S$ into two sequences $S_1$ and $S_2$ of about $n/2$ elements each.
  - Recur: recursively sort $S_1$ and $S_2$.
  - Conquer: merge $S_1$ and $S_2$ into a unique sorted sequence.

Merge-Sort Tree

- An execution of merge-sort is depicted by a binary tree:
  - Each node represents a recursive call of merge-sort and stores an unsorted sequence before the execution and its partition.
  - The root is the initial call.
  - The leaves are calls on subsequences of size 0 or 1.

Merging Two Sorted Sequences

- The conquer step of merge-sort consists of merging two sorted sequences $A$ and $B$ into a sorted sequence $S$ containing the union of the elements of $A$ and $B$.
- Merging two sorted sequences, each with $n/2$ elements and implemented by means of a doubly linked list, takes $O(n)$ time.

Algorithm $mergeSort(A, B)$

Input:
- sequences $A$ and $B$ with $n/2$ elements each
- $S$, a sequence containing the union of $A$ and $B$

Output:
- sorted sequence $S$ with $n$ elements

Output:

```
S ← empty sequence
while ¬LisaEmpty() ∧ ¬BisEmpty()
  if A.first() ≤ B.first()
    S.insertLast(A.remove(A.first()))
  else
    S.insertLast(B.remove(B.first()))
while ¬LisaEmpty()
  S.insertLast(A.remove(A.first()))
while ¬BisEmpty()
  S.insertLast(B.remove(B.first()))
return S
```
Execution Example

Partition

7 2 9 4 3 8 6 1

Recursive call, partition

7 2 9 4

Recursive call, base case

7 2 9 4

Merge
### Merge Sort

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

![Diagram](merge-sort-diagram.png)

**Execution Example (cont.)**
- Merge

![Diagram](merge-sort-diagram.png)

**Analysis of Merge-Sort**
- The height $h$ of the merge-sort tree is $O(\log n)$
  - at each recursive call we divide in half the sequence,
  - The overall amount or work done at the nodes of depth $i$ is $O(n)$
    - we partition and merge $2^i$ sequences of size $n/2^i$
    - we make $2^i + 1$ recursive calls
- Thus, the total running time of merge-sort is $O(n \log n)$

<table>
<thead>
<tr>
<th>depth</th>
<th>$n$</th>
<th>$n/2^i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$n$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$n/2$</td>
<td></td>
</tr>
<tr>
<td>$i$</td>
<td>$n/2^i$</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

**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>slow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for small data sets (&lt; 1K)</td>
</tr>
<tr>
<td>insertion-sort</td>
<td>$O(n^2)$</td>
<td>slow</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for small data sets (&lt; 1K)</td>
</tr>
<tr>
<td>heap-sort</td>
<td>$O(n \log n)$</td>
<td>fast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>in-place</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for large data sets (1K — 1M)</td>
</tr>
<tr>
<td>merge-sort</td>
<td>$O(n \log n)$</td>
<td>fast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>sequential data access</td>
</tr>
<tr>
<td></td>
<td></td>
<td>for huge data sets (&gt; 1M)</td>
</tr>
</tbody>
</table>