Outline and Reading

◆ Divide-and-conquer paradigm (§10.1.1)
◆ Merge-sort (§10.1)
  ▪ Algorithm
  ▪ Merging two sorted sequences
  ▪ Merge-sort tree
  ▪ Execution example
  ▪ Analysis
◆ Generic merging and set operations (§10.2)
◆ Summary of sorting algorithms

Merge-Sort

Divide-and-Conquer

Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm. It uses a comparator. It has $O(n \log n)$ running time. Unlike heap-sort, it does not use an auxiliary priority queue. It accesses data in a sequential manner (suitable for sort data on a disk).

Merge-Sort Tree

An execution of merge-sort is depicted by a binary tree.

1. Each node represents a recursive call of merge-sort and stores
   ▪ unsorted sequence before the execution and its partition
   ▪ 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

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 $B$.

1. 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:

1. Divide: partition $S$ into two sequences $S_1$ and $S_2$ of about $n/2$ elements each.
2. Recur: recursively sort $S_1$ and $S_2$.
3. Conquer: merge $S_1$ and $S_2$ into a unique sorted sequence.

The algorithm has $O(n \log n)$ time complexity. The tree is balanced if the input data is random.

Algorithm $mergeSort(S, C)$

Input: sequence $S$ with $n$ elements, comparator $C$
Output: sequence $S$ sorted according to $C$

1. If $S.size() > 1$
   2. $(S_1, S_2) \leftarrow partition(S, n/2)$
   3. $mergeSort(S_1, C)$
   4. $mergeSort(S_2, C)$
   5. $S \leftarrow merge(S_1, S_2)$

Algorithm $merge(A, B)$

Input: sequences $A$ and $B$ with $n/2$ elements each.
Output: sorted sequence of $A \cup B$

1. $S \leftarrow$ empty sequence
2. while $\neg A.isEmpty() \land \neg B.isEmpty()$
   3. if $A.first().element < B.first().element$
      4. $S.insertLast(A.remove(A.first().))$
   5. else
      6. $S.insertLast(B.remove(B.first().))$
3. while $\neg A.isEmpty()$
   4. $S.insertLast(A.remove(A.first().))$
5. while $\neg B.isEmpty()$
   6. $S.insertLast(B.remove(B.first().))$
7. return $S$
Execution Example

Partition

7 2 9 4 3 8 6 1

Recursive call, partition

Recursive call, base case

Merge
Execution Example (cont.)

Recursive call, ..., base case, merge

7 2 9 4 | 3 8 6 1

7 2 | 9 4 → 2 4 7 9
9 4 | 2 4 7 9 → 1 3 6 8
7 2 | 2 4 7 9 → 1 3 6 8
9 4 | 1 3 6 8 → 1 2 3 4 6 7 8 9

Merge

7 2 9 4 | 3 8 6 1 → 1 2 3 4 6 7 8 9

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 $O(n \log n)$

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