Merge Sort

7 2 | 9 4  →  2 4 7 9

7 | 2  →  2 7

7 → 7  2 → 2

9 | 4  →  4 9

9 → 9  4 → 4
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 \( S \) in 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 is a sorting algorithm based on the divide-and-conquer paradigm
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

**Algorithm** $mergeSort(S, C)$

*Input* sequence $S$ with $n$ elements, comparator $C$

*Output* sequence $S$ sorted according to $C$

if $S.size() > 1$

$(S_1, S_2) \leftarrow partition(S, n/2)$

$mergeSort(S_1, C)$

$mergeSort(S_2, C)$

$S \leftarrow merge(S_1, S_2)$
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** \( \text{merge}(A, B) \)

**Input** sequences \( A \) and \( B \) with \( n/2 \) elements each

**Output** sorted sequence of \( A \cup B \)

\[
S \leftarrow \text{empty sequence}
\]

while \( \neg A.\text{isEmpty()} \land \neg B.\text{isEmpty()} \)

if \( A.\text{first().element()} < B.\text{first().element()} \)

\( S.\text{insertLast}(A.\text{remove(A.first())}) \)

else

\( S.\text{insertLast}(B.\text{remove(B.first())}) \)

while \( \neg A.\text{isEmpty()} \)

\( S.\text{insertLast}(A.\text{remove(A.first())}) \)

while \( \neg B.\text{isEmpty()} \)

\( S.\text{insertLast}(B.\text{remove(B.first())}) \)

return \( S \)
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
  - 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
Execution Example

Partition

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

2/28/2002 1:16 AM Merge Sort 7
Execution Example (cont.)

Recursive call, partition

7 2 9 4 | 3 8 6 1

1 2 3 4 6 7 8 9
Execution Example (cont.)

 Recursive call, partition

```
7 2 9 4 3 8 6 1
```

```
7 2 | 9 4
```

```
7 | 2
```

```
7 2 9 4
```

```
3 8 6 1
```

2/28/2002 1:16 AM
Execution Example (cont.)

Recursive call, base case

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

```
7 2 | 9 4
```

```
7 | 2
```

```
7 \rightarrow 7
```

```
7 \rightarrow 2
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```

```
7 \rightarrow 7
```
Execution Example (cont.)

Recursive call, base case

```
[7 2 9 4 | 3 8 6 1]
```

```
[7 2 | 9 4]
```

```
[7 | 2]
```

```
7 → 7
```

```
7 → 7
```

```
7 → 7
```

```
7 → 7
```

```
1 2 3 4 6 7 8 9
```
Execution Example (cont.)

Merge Sort

7 2 9 4 | 3 8 6 1

7 2 9 4

7 2 → 2 7

7 → 7  2 → 2

Merge Sort

12
Execution Example (cont.)

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

Merge Sort

2/28/2002 1:16 AM
Execution Example (cont.)

Recursive call, ..., merge, merge

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

```
7 2 | 9 4 → 2 4 7 9
```

```
7 | 2 → 2 7
```

```
9 4 → 4 9
```

```
9 → 9
```

```
4 → 4
```

```
3 8 → 3 8
```

```
3 → 3
```

```
8 → 8
```

```
6 1 → 1 6
```

```
6 → 6
```

```
1 → 1
```

2/28/2002 1:16 AM
Execution Example (cont.)

Merge Sort

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

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

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

```
7 → 7
2 → 2
9 → 9
4 → 4
3 → 3
8 → 8
6 → 6
1 → 1
```
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</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>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>