Merge Sort

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:
  - **Like heap-sort**: 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 to sort data on a disk).
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 \texttt{mergeSort}(S)

**Input** sequence \( S \) with \( n \) elements
**Output** sequence \( S \) sorted according to \( C \)

if \( S\text{.size}() > 1 \)
  \( (S_1, S_2) \leftarrow \text{partition}(S, n/2) \)
  \texttt{mergeSort}(S_1)
  \texttt{mergeSort}(S_2)

\( S \leftarrow \text{merge}(S_1, S_2) \)

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 \texttt{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 \( A\text{.first}().element() < B\text{.first}().element() \)
    \( S\text{.addLast}(A\text{.remove}(A\text{.first}())) \)
  else
    \( S\text{.addLast}(B\text{.remove}(B\text{.first}())) \)

while \( \neg A\text{.isEmpty()} \)
  \( S\text{.addLast}(A\text{.remove}(A\text{.first}())) \)

while \( \neg B\text{.isEmpty()} \)
  \( S\text{.addLast}(B\text{.remove}(B\text{.first}())) \)

return \( S \)
Java Merge Implementation

```java
/** Merge contents of arrays S1 and S2 into properly sized array S. */
public static <K> void merge(K[], K[], Comparator<K> comp) {
    int i = 0, j = 0;
    while (i + j < S.length) {
        if (i == S2.length || (i < S1.length && comp.compare(S1[i], S2[j]) < 0))
            S[i+j] = S1[i++]; // copy ith element of S1 and increment i
        else
            S[i+j] = S2[j++]; // copy jth element of S2 and increment j
    }
}
```

Java Merge-Sort Implementation

```java
/** Merge-sort contents of array S. */
public static <K> void mergeSort(K[], S, Comparator<K> comp) {
    int n = S.length;
    if (n < 2) return; // array is trivially sorted
    // divide
    int mid = n/2;
    K[] S1 = Arrays.copyOfRange(S, 0, mid); // copy of first half
    K[] S2 = Arrays.copyOfRange(S, mid, n); // copy of second half
    // conquer (with recursion)
    mergeSort(S1, comp); // sort copy of first half
    mergeSort(S2, comp); // sort copy of second half
    // merge results
    merge(S1, S2, S, comp); // merge sorted halves back into original
}
```
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
Execution Example (cont.)

Recursive call, partition

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

Recursive call, base case

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

© 2014 Goodrich, Tamassia, Goldwasser  Merge Sort  11

Execution Example (cont.)

Recursive call, base case

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

© 2014 Goodrich, Tamassia, Goldwasser  Merge Sort  12
Execution Example (cont.)

**Merge**

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

```
1 3 8 6
1 6
1
```

```
2 4 9
2 7
2
```

**Recursive call, ..., base case, merge**

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

```
8 3
8
```

```
6 1
6
```

© 2014 Goodrich, Tamassia, Goldwasser
Execution Example (cont.)

- Merge

7 2 9 4 | 3 8 6 1

- Recursive call, ..., merge, merge

7 2 9 4 | 3 8 6 1

© 2014 Goodrich, Tamassia, Goldwasser
Execution Example (cont.)

Merge

<table>
<thead>
<tr>
<th>7 2 9 4</th>
<th>3 8 6 1</th>
<th>1 2 3 4 6 7 8 9</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>7 2</th>
<th>9 4</th>
<th>2 4 7 9</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>7</th>
<th>2</th>
<th>2 7</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>9 4</th>
<th>4 9</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>3 8</th>
<th>3 8</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>6 1</th>
<th>1 6</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>7</th>
<th>2</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>9</th>
<th>4</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>3</th>
<th>8</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>6</th>
<th>1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>1</th>
</tr>
</thead>
</table>

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>