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 design paradigm:
  - Divide: divide the input data $S$ into two (or more) 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 $S$.
- Base case: directly solve and do not divide for "small" subproblem sizes (typically 0 or 1).

- Merge-sort is a sorting algorithm based on divide-and-conquer.
- Like heap-sort:
  - $O(n \log n)$ running time
- Unlike heap-sort:
  - No auxiliary priority queue
  - Accesses data sequentially (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.
- **Recurs**: recursively sort \( S_1 \) and \( S_2 \).
- **Conquer**: merge \( S_1 \) and \( S_2 \) into a unique sorted sequence.

**Algorithm mergeSort**

Input: sequence \( S \) with \( n \) elements
Output: sequence \( S \) sorted

\[
\begin{align*}
S_1, S_2 &\leftarrow \text{partition}(S, n/2) \\
S_1 &\leftarrow \text{mergeSort}(S_1) \\
S_2 &\leftarrow \text{mergeSort}(S_2) \\
S &\leftarrow \text{merge}(S_1, S_2) \\
\text{return} S
\end{align*}
\]

**Partitioning a Sequence**

The divide step of merge-sort consists of partitioning input sequence \( S \).
- Use doubly linked list with head and tail pointer.
- Then all sequence ADT operations take \( O(1) \) time.
- With \( n \) total elements, partition takes \( O(n) \) time.

**Algorithm partition**

Input: sequence \( S \) with \( n \) items; \( k \), partition size
Output: partition of \( S \) into \( S_1 \) of size \( k \) and \( S_2 \) of size \( n-k \)

\[
\begin{align*}
S_1 &\leftarrow \text{empty sequence} \\
S_2 &\leftarrow \text{empty sequence} \\
pos &\leftarrow S.\text{first}() \\
\text{for } i &\leftarrow 1 \text{ to } k \text{ do} \\
S_1.\text{insertLast}(pos.\text{element}()) \\
pos &\leftarrow S.\text{after}(pos) \\
\text{for } i &\leftarrow k+1 \text{ to } n \text{ do} \\
S_2.\text{insertLast}(pos.\text{element}()) \\
pos &\leftarrow S.\text{after}(pos) \\
\text{return } (S_1, S_2)
\end{align*}
\]

**Merging Two Sorted Sequences**

The conquer step of merge-sort consists of merging two sorted sequences \( A \) and \( B \).
- Use doubly linked list with head and tail pointer.
- Then all sequence ADT operations take \( O(1) \) time.
- With \( n \) total elements, merge takes \( O(n) \) time.

**Algorithm merge**

Input: sequence \( A \) and \( B \), both sorted, with \( n \) total items combined
Output: sorted sequence of \( A \cup B \)

\[
\begin{align*}
S &\leftarrow \text{empty sequence} \\
\text{while } \neg A.\text{isEmpty}() \land \neg B.\text{isEmpty}() \\
\text{if } A.\text{first}().\text{element}() \prec B.\text{first}().\text{element}() \\
S.\text{insertLast}(A.\text{remove}(A.\text{first}())) \\
\text{else} \\
S.\text{insertLast}(B.\text{remove}(B.\text{first}())) \\
\text{while } \neg A.\text{isEmpty}() \\
S.\text{insertLast}(A.\text{remove}(A.\text{first}())) \\
\text{while } \neg B.\text{isEmpty}() \\
S.\text{insertLast}(B.\text{remove}(B.\text{first}())) \\
\text{return } S
\end{align*}
\]
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

Recursive call, partition
Execution Example (cont.)

Merge Sort version 1.3

Execution Example (cont.)

Recursive call, ..., base case, merge

Execution Example (cont.)

Merge Sort version 1.3
**Execution Example (cont.)**

- Recursive call, ..., merge, merge

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

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

```
9 4 | 4 9
```

```
1 8 | 3 8
```

```
6 1 | 1 8
```

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

```
| 9 4 | 4 9
```

```
| 1 8 | 3 8
```

```
| 6 1 | 1 8
```

```
1 3 6 8
```

```
3 8 6 1
```

```
6 1 8
```

```
1 8
```

```
3 8 6 1
```

```
6 1 8
```

```
1 8
```

```
3 8 6 1
```

```
6 1 8
```

```
1 8
```

```
3 8 6 1
```

```
6 1 8
```

```
1 8
```

```
```

**Merge-Sort Analysis**

- Use recurrence equation.

```
Algorithm mergeSort(S)
Input sequence S with n elements
Output sequence S sorted
if n > 1
    (S, S1) ↔ partition(S, n/2)
    S1 ↔ mergeSort(S1)
    S2 ↔ mergeSort(S2)
    S ↔ merge(S1, S2)
return S
```

Merge-Sort Analysis

- Use recurrence equation.
- \( T(0) = T(1) = 2 \)
- \( T(n) = cn + T(n/2) + T(n/2) + 2cn = 2cn + 2T(n/2) \)
- \( c \) is a constant.

Algorithm `mergeSort(S)`

```
Input sequence S with n elements
Output sequence S sorted
```

```java
if n > 1
    (S1, S2) ← partition(S, n/2)
    S1 ← mergeSort(S1)
    S2 ← mergeSort(S2)
S ← merge(S1, S2)
return S
```

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 \( O(n/2^i) \)
  - 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 about \( 2cn \log n \), or \( O(n \log n) \)

```
<table>
<thead>
<tr>
<th>depth</th>
<th>calls</th>
<th>size</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>n</td>
<td>2cn</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>n/2</td>
<td>2cn</td>
</tr>
<tr>
<td>i</td>
<td>2^i</td>
<td>n/2^i</td>
<td>2cn</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

The Recursion Tree

- For solving divide-and-conquer recurrence relations:

\[
T(n) = \begin{cases} 
    b & \text{if } n < 2 \\
    2T(n/2) + bn & \text{if } n \geq 2 
\end{cases}
\]

```
<table>
<thead>
<tr>
<th>depth</th>
<th>size</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>n</td>
<td>bn</td>
</tr>
<tr>
<td>1</td>
<td>n/2</td>
<td>bn</td>
</tr>
<tr>
<td>i</td>
<td>n/2^i</td>
<td>bn</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

Total cost = \( bn + bn \log n \)
(last level plus all previous levels)

Merge Sort version 1.3
### 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>

### Divide-and-Conquer

![Divide-and-Conquer Diagram]

- Analysis can be done using recurrence equations
- What would recurrence equation look like for this tree?
Recurrence Equation Analysis

The conquer step of merge-sort consists of merging two sorted sequences, each with \( n/2 \) elements and implemented by means of a doubly linked list, takes at most \( bn \) steps, for some constant \( b \).

The basis case \( (n < 2) \) takes 2 steps.

Therefore, if we let \( T(n) \) denote the running time of merge-sort:

\[
T(n) = \begin{cases} 
2 & \text{if } n < 2 \\
2T(n/2) + bn & \text{if } n \geq 2 
\end{cases}
\]

We can therefore analyze the running time of merge-sort by finding a closed form solution to the above equation.

- That is, a solution that has \( T(n) \) only on the left-hand side.

Iterative Substitution

In the iterative substitution, or "plug-and-chug," technique, we iteratively apply the recurrence equation to itself and see if we can find a pattern:

\[
T(n) = 2T(n/2) + bn \\
= 2(2T(n/2^2) + bn) \\
= 2^2T(n/2^2) + 2bn \\
= 2^3T(n/2^3) + 3bn \\
= 2^4T(n/2^4) + 4bn \\
= ... \\
= 2^iT(n/2^i) + ibn
\]

Note that base, \( T(1)=2 \), case occurs when \( n/2^i=1 \) (Or \( i = \log n \)).

So, \( T(n) = 2n + bn \log n \)

Thus, \( T(n) = O(n \log n) \).

Solving recurrence equations

- Recurrence Trees (already shown)
- Iterative Substitution (already shown)
- Guess-and-Test Method (in book)
- Master Method (next)

- does not apply to all recurrence equations!
Master Method

Many divide-and-conquer recurrence equations have the form:
\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

The Master Theorem: Note: \( a, k, \varepsilon \) are constants you pick.
1. if \( f(n) = O(n^{\varepsilon k}) \), then \( T(n) = \Theta(n^{\varepsilon k}) \)
2. if \( f(n) = \Theta(n^{\varepsilon k} \log^a n) \), then \( T(n) = \Theta(n^{\varepsilon k} \log^{a+1} n) \)
3. if \( f(n) = \Omega(n^{\varepsilon k + \delta}) \), then \( T(n) = \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Master Method, Example 1

The form:
\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

The Master Theorem:
1. if \( f(n) = O(n^{\varepsilon k}) \), then \( T(n) = \Theta(n^{\varepsilon k}) \)
2. if \( f(n) = \Theta(n^{\varepsilon k} \log^a n) \), then \( T(n) = \Theta(n^{\varepsilon k} \log^{a+1} n) \)
3. if \( f(n) = \Omega(n^{\varepsilon k + \delta}) \), then \( T(n) = \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Example:
\[ T(n) = 4T(n/2) + n \]
Solution: \( \log_a 4 = 2 \), so case 1 says \( T(n) = O(n^2) \).

Master Method, Example 2

The form:
\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

The Master Theorem:
1. if \( f(n) = O(n^{\varepsilon k}) \), then \( T(n) = \Theta(n^{\varepsilon k}) \)
2. if \( f(n) = \Theta(n^{\varepsilon k} \log^a n) \), then \( T(n) = \Theta(n^{\varepsilon k} \log^{a+1} n) \)
3. if \( f(n) = \Omega(n^{\varepsilon k + \delta}) \), then \( T(n) = \Theta(f(n)) \), provided \( af(n/b) \leq \delta f(n) \) for some \( \delta < 1 \).

Example:
\[ T(n) = 2T(n/2) + n \log n \]
Solution: \( \log_a 2 = 1 \), so case 2 says \( T(n) = O(n \log^2 n) \).
Master Method, Example 3

- **The form:**
  \[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
  \end{cases} \]

- **The Master Theorem:**
  1. If \( f(n) \) is \( O(n^{\log_b a}) \), then \( T(n) = \Theta(n^{\log_b a}) \).
  2. If \( f(n) \) is \( \Theta(n^{\log_b a} \log^k n) \), then \( T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \).
  3. If \( f(n) \) is \( \Omega(n^{\log_b a} \log^k n) \), then \( T(n) = \Theta(f(n)) \),
     provided \( (a/b)^\delta \leq \delta \) for some \( \delta > 1 \).

- **Example:**
  \[ T(n) = T(n/3) + n \log n \]

Solution: \( \log_a a = 1 \), so case 1 says \( T(n) = O(n \log n) \).

Master Method, Example 4

- **The form:**
  \[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
  \end{cases} \]

- **The Master Theorem:**
  1. If \( f(n) \) is \( O(n^{\log_b a}) \), then \( T(n) = \Theta(n^{\log_b a}) \).
  2. If \( f(n) \) is \( \Theta(n^{\log_b a} \log^k n) \), then \( T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \).
  3. If \( f(n) \) is \( \Omega(n^{\log_b a} \log^k n) \), then \( T(n) = \Theta(f(n)) \),
     provided \( (a/b)^\delta \leq \delta \) for some \( \delta > 1 \).

- **Example:**
  \[ T(n) = 8T(n/2) + n^2 \]

Solution: \( \log_a a = 3 \), so case 1 says \( T(n) = O(n^3) \).

Master Method, Example 5

- **The form:**
  \[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
  \end{cases} \]

- **The Master Theorem:**
  1. If \( f(n) \) is \( O(n^{\log_b a}) \), then \( T(n) = \Theta(n^{\log_b a}) \).
  2. If \( f(n) \) is \( \Theta(n^{\log_b a} \log^k n) \), then \( T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \).
  3. If \( f(n) \) is \( \Omega(n^{\log_b a} \log^k n) \), then \( T(n) = \Theta(f(n)) \),
     provided \( (a/b)^\delta \leq \delta \) for some \( \delta > 1 \).

- **Example:**
  \[ T(n) = 9T(n/3) + n^3 \]

Solution: \( \log_a a = 2 \), so case 3 says \( T(n) = O(n^3) \).
Master Method, Example 6

The form:

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

The Master Theorem:

1. if \( f(n) = O(n^{\log_b a - \varepsilon}) \), then \( T(n) = \Theta(n^{\log_b a}) \)
2. if \( f(n) = \Theta(n^{\log_b a \log \log n}) \), then \( T(n) = \Theta(n^{\log_b a \log \log n}) \)
3. if \( f(n) = \Omega(n^{\log_b a + \varepsilon}) \), then \( T(n) = \Theta(f(n)) \), provided \( a^{\log_b c} \leq \delta f(n) \) for some \( \delta < 1 \).

Example:

\[ T(n) = T(n/2) + 1 \quad \text{(binary search)} \]

Solution: \( \log_b a = 0 \), so case 2 says \( T(n) = O(\log n) \).

Master Method, Example 7

The form:

\[ T(n) = \begin{cases} 
  c & \text{if } n < d \\
  aT(n/b) + f(n) & \text{if } n \geq d 
\end{cases} \]

The Master Theorem:

1. if \( f(n) = O(n^{\log_b a - \varepsilon}) \), then \( T(n) = \Theta(n^{\log_b a}) \)
2. if \( f(n) = \Theta(n^{\log_b a \log \log n}) \), then \( T(n) = \Theta(n^{\log_b a \log \log n}) \)
3. if \( f(n) = \Omega(n^{\log_b a + \varepsilon}) \), then \( T(n) = \Theta(f(n)) \), provided \( a^{\log_b c} \leq \delta f(n) \) for some \( \delta < 1 \).

Example:

\[ T(n) = 2T(n/2) + \log n \quad \text{(heap construction)} \]

Solution: \( \log_b a = 1 \), so case 1 says \( T(n) = O(n) \).

Iterative “Proof” of the Master Theorem

Using iterative substitution, let us see if we can find a pattern:

\[ T(n) = aT(n/b) + f(n) \]

\[ = a[aT(n/b^2) + f(n/b)] + bn \]

\[ = a^2T(n/b^2) + af(n/b) + f(n) \]

\[ = a^3T(n/b^3) + a^2f(n/b^2) + af(n/b) + f(n) \]

\[ = \cdots \]

\[ = a^{\log_b n}T(1) + \sum_{k=0}^{\log_b n} a^k f(n/b^k) \]

\[ = a^{\log_b n}T(1) + \sum_{k=0}^{\log_b n} a^k f(n/b^k) \]

We then distinguish the three cases as:

- The first term is dominant
- Each part of the summation is equally dominant
- The summation is a geometric series