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 set in 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
  - Recur: recursively sort \( S_1 \) and \( S_2 \)
  - Conquer: merge \( S_1 \) and \( S_2 \) into a unique sorted sequence

Algorithm `mergeSort(S)`

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

```plaintext
if \( n > 1 \)
    \( (S_1, S_2) \leftarrow \text{partition}(S, n/2) \)
    \( S_1 \leftarrow \text{mergeSort}(S_1) \)
    \( S_2 \leftarrow \text{mergeSort}(S_2) \)
    \( S \leftarrow \text{merge}(S_1, S_2) \)
```

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 a total elements, partition takes \( O(n) \) time.

Algorithm `partition(S, k)`

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

```plaintext
S_1 \leftarrow \text{empty sequence}
S_2 \leftarrow \text{empty sequence}
pos \leftarrow \text{S.first}()
for \( i \leftarrow 1 \) to \( k \) do
    S_1.insertLast(pos.element())
    pos \leftarrow \text{S.after}(pos)
for \( i \leftarrow k+1 \) to \( n \) do
    S_2.insertLast(pos.element())
    pos \leftarrow \text{S.after}(pos)
return \((S_1, S_2)\)
```

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 a total elements, merge takes \( O(n) \) time.

Algorithm `merge(A, B)`

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

```plaintext
S \leftarrow \text{empty sequence}
while \( \neg A.isEmpty() \) \&\& \( \neg B.isEmpty() \)
    if \( A.first().element() < B.first().element() \)
        S.insertLast(A.remove(A.first()))
    else
        S.insertLast(B.remove(B.first()))
while \( \neg A.isEmpty() \)
    S.insertLast(A.remove(A.first()))
while \( \neg B.isEmpty() \)
    S.insertLast(B.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.

![Merge-Sort Tree Diagram]

Execution Example

Partition

![Execution Example Diagram]

Execution Example (cont.)

Recursive call, partition

![Execution Example (cont.) Diagram]
Execution Example (cont.)

Recursive call, partition

Execution Example (cont.)

Recursive call, base case

Execution Example (cont.)

Recursive call, base case
Merge Sort

Execution Example (cont.)

Recursive call, ..., base case, merge

Merge Sort version 1.2

13

20

14

32

15

5
Execution Example (cont.)

Recursive call, ..., merge, merge

Execution Example (cont.)

Merge

Merge-Sort Analysis

Algorithm mergeSort(S)
Input sequence S with n elements
Output sequence S sorted
if n > 1
    (S₁, S₂) ← partition(S, n/2)
    mergeSort(S₁)
    mergeSort(S₂)
    S ← merge(S₁, S₂)
return S
Merge-Sort Analysis

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

Algorithm mergeSort(S)

Input sequence $S$ with $n$ elements
Output sequence $S$ sorted
if $n > 1$
    $(S_1, S_2) \leftarrow \text{partition}(S, n/2)$
    mergeSort($S_1$)
    mergeSort($S_2$)
    $S \leftarrow \text{merge}(S_1, S_2)$
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 $O(n)$
  - 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 $2n \log n$, or $O(n \log n)$

<table>
<thead>
<tr>
<th>depth</th>
<th>size</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>$2c$</td>
</tr>
<tr>
<td>1</td>
<td>$n/2$</td>
<td>$2cn$</td>
</tr>
<tr>
<td>$i$</td>
<td>$n/2^i$</td>
<td>$2cn$</td>
</tr>
</tbody>
</table>

The Recursion Tree

For solving divide-and-conquer recurrence relations:

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

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

### Divide-and-Conquer

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:

\[
T(n) = 2T(n/2) + bn = 2^2T(n/2^2) + 2^2bn = 2^3T(n/2^3) + 3^2bn = 2^4T(n/2^4) + 4^2bn = \ldots
\]

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!
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: are constants you pick.

1. if \( f(n) = O(n^{\log_b a}) \), then \( T(n) = \Theta(n^{\log_b a}) \)
2. if \( f(n) = \Theta(n^{\log_b a} \log^k n) \), then \( T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \)
3. if \( f(n) = \Omega(n^{\log_b a} \log^{k+\epsilon} n) \), 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 \( f(n) = O(n^{\log_b a}) \), then \( T(n) = \Theta(n^{\log_b a}) \)
2. if \( f(n) = \Theta(n^{\log_b a} \log^k n) \), then \( T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \)
3. if \( f(n) = \Omega(n^{\log_b a} \log^{k+\epsilon} n) \), 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: \( \log_b a = 2 \), 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 \( f(n) = O(n^{\log_b a}) \), then \( T(n) = \Theta(n^{\log_b a}) \)
2. if \( f(n) = \Theta(n^{\log_b a} \log^k n) \), then \( T(n) = \Theta(n^{\log_b a} \log^{k+1} n) \)
3. if \( f(n) = \Omega(n^{\log_b a} \log^{k+\epsilon} n) \), 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: \( \log_b a = 1 \), 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 \( f(n) = O(n^{\log_b a - \epsilon}) \), then \( T(n) = \Theta(n^{\log_b a}) \)
2. If \( f(n) = \Theta(n^{\log_b a} \cdot \log^k n) \), then \( T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n) \)
3. If \( f(n) = \Omega(n^{\log_b a + \epsilon}) \), then \( T(n) = \Theta(f(n)) \), provided \( a(n/b)^{\log_b a} \leq \delta f(n) \) for some \( \delta < 1 \).

Example:
\[ T(n) = T(n/3) + n \log n \]
Solution: \( \log_a n = 0 \), so case 3 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 \( f(n) = O(n^{\log_b a - \epsilon}) \), then \( T(n) = \Theta(n^{\log_b a}) \)
2. If \( f(n) = \Theta(n^{\log_b a} \cdot \log^k n) \), then \( T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n) \)
3. If \( f(n) = \Omega(n^{\log_b a + \epsilon}) \), then \( T(n) = \Theta(f(n)) \), provided \( a(n/b)^{\log_b a} \leq \delta f(n) \) for some \( \delta < 1 \).

Example:
\[ T(n) = 8T(n/2) + n^2 \]
Solution: \( \log_a n = 3 \), 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 \( f(n) = O(n^{\log_b a - \epsilon}) \), then \( T(n) = \Theta(n^{\log_b a}) \)
2. If \( f(n) = \Theta(n^{\log_b a} \cdot \log^k n) \), then \( T(n) = \Theta(n^{\log_b a} \cdot \log^{k+1} n) \)
3. If \( f(n) = \Omega(n^{\log_b a + \epsilon}) \), then \( T(n) = \Theta(f(n)) \), provided \( a(n/b)^{\log_b a} \leq \delta f(n) \) for some \( \delta < 1 \).

Example:
\[ T(n) = 9T(n/3) + n^3 \]
Solution: \( \log_a n = 2 \), 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 \( f(n) = O(n^{d-\epsilon}) \), then \( T(n) = \Theta(n^d) \)
2. if \( f(n) = \Theta(n^{d} \log^k n) \), then \( T(n) = \Theta(n^d \log^{k+1} n) \)
3. if \( f(n) = \Omega(n^{d+\epsilon}) \), then \( T(n) = \Theta(f(n)) \)

provided \( a/b < \Theta(f(n)) \) 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 \( f(n) = O(n^{d-\epsilon}) \), then \( T(n) = \Theta(n^d) \)
2. if \( f(n) = \Theta(n^{d} \log^k n) \), then \( T(n) = \Theta(n^d \log^{k+1} n) \)
3. if \( f(n) = \Omega(n^{d+\epsilon}) \), then \( T(n) = \Theta(f(n)) \)

provided \( a/b < \Theta(f(n)) \) 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:

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

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

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

\[ \vdots \]

\[ = a^nT(n) + \sum_{i=0}^{n-1} a^i f(n/b^i) \]

\[ = a^nT(n) + \sum_{i=0}^{n-1} a^i f(n/b^i) \]

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

---