Optimization Problems

- **Optimization Problems**
  - Find an optimal solution from many possible solutions
  - NOTE: optimal solution may not be unique
- Applications of optimization problems:
  - EDA: *electronic design automation*
  - control, operation, our daily lives ...
- Approaches in this lecture:
  - **Dynamic Programming**: (usually bottom up)
    - divide the problem into subproblems, which overlap
      - unlike D&C, where subproblems are disjoint
    - solve each subproblem just once
    - record the best solution in a table
  - **Greedy Algorithms**: (usually top-down)
    - make a choice that look best at the moment
    - Do NOT always yield an optimal solution.
      - But sometimes they do
Dynamic Programming*

- DP solves problems by combining the solution to subproblems
  - subproblems overlap (subproblems share subproblems)
    * Unlike divide and conquer, where subproblems are disjoint
- DP solve each subproblem once and store the results in a table

* “programming” means using a tabular solution to an optimization problem
Dynamic Programming

- Four steps
  - 1. Characterize the structure of an optimal solution.  
    - *optimal substructure*
  - 2. Recursively define the value of an optimal solution
  - 3. Compute the value of an optimal solution,
    - typically in a *bottom-up* fashion.
    - memorize solutions to many *overlapping subproblems*
  - 4. Construct an optimal solution from step 3.

- Example optimization problems
  - Rod cutting
  - Longest common subsequence
  - Optimal Binary Search Tree

Rod Cutting

- How to cut steel rods into pieces in order to maximize the revenue
  - Each cut is free.
  - Rod lengths are always an integral number of inches.
- Input: A rod of length \( n \) and a prices table \( p_i \), for \( i = 1, 2, ..., n \).

- Output: maximum revenue \( r_n \) for rods whose lengths sum to \( n \),  
  - computed as the sum of the prices for the individual rods

- FFT: how many different ways to cut an \( n \)-inch rod? What is the complexity of exhaustive search? 
  - assume all cut are integers
Example

- Price Table: Fig. 15.1

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_i$</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>20</td>
<td>24</td>
<td>30</td>
</tr>
</tbody>
</table>

- 8 different ways to cut an 4-inch rod: 2+2 is optimal

- optimal solutions $n = 1\sim 8$

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_i$</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2 + 2</td>
<td>2 + 3</td>
<td>6</td>
</tr>
<tr>
<td>optimal solution</td>
<td>(no cuts)</td>
<td>(no cuts)</td>
<td>2</td>
<td>2 (no cuts)</td>
<td>3</td>
<td>2 + 2</td>
<td>2 + 3</td>
<td>(no cuts)</td>
<td>1 + 6 or 2 + 2 + 3</td>
</tr>
</tbody>
</table>

Note:
- If $p_n$ is large enough, optimal solution might require no cut

Step 1: Optimal Substructure

- Idea: Divide this optimization problems into subproblems
- suppose $r_n = \text{optimal revenue of rod of length } n$
- we can obtain $r_n$ by taking the maximum of
  - $p_n$: the price we get by not cutting,
  - $p_1+r_{n-1}$: maximum revenue from a 1-inch rod and $n-1$ inches rod,
  - $p_2+r_{n-2}$: maximum revenue from a 2-inch rod and a $n-2$ inches rod
  - ...
  - $p_{n-1} + r_1$
- so, $r_n = \max (p_n, p_1+r_{n-1}, p_2 + r_{n-2}, \ldots, p_{n-1} + r_1)$
- Q: does this approach guartantee to find the optimal solution?
  - can you find a better solution that is NOT included in these $n$ subproblems?
Step 1: Optimal Substructure (2)

- **Optimal Substructure:**
  - To solve the original problem of size \( n \), solve subproblems on smaller sizes.
  - The optimal solution to the original problem incorporates optimal solutions to the subproblems.
  - We may solve the subproblems independently.
  - For rod cutting, if \( r_n \) is optimal solution for length \( n \), then the solution to its subproblem (\( r_{n-i} \)) is also an optimal cut of length \( n-i \)

- **Proof:**
  - if \( r_{n-i} \) is NOT an optimal cut, then replace it by an optimal cut
  - this violate our assumption that \( r_n \) is an optimal cut

- **NOTE:** this statement is NOT reversible
  - putting small optimal solutions together does NOT give a overall optimal solution

![Diagram](image)

Step 2: Recursive Solution

- However, we do not know what was the last cut \( i \)
  - so we must try all possible values of last cut
  - we are guaranteed to find the optimal solution after trying all \( i \)
- Recursive version of the equation for \( r_n \)

\[
r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})
\]
Step 3: Compute Opt. Solution (Top-Down)

- A direct implementation: CUT-ROD returns the optimal revenue $r_n$

```plaintext
CUT-ROD(p, n)
if n == 0
    return 0
q = -∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n - i))
return q
```

- Works but inefficient
  - calls itself repeatedly, even on subproblems already solved.
- Example: $n=4$ Solve the subproblem for
  - size 2 twice,
  - size 1 four times,
  - size 0 eight times.

Complexity Analysis

- $T(n)$ equal the number of calls to CUT-ROD

```plaintext
T(n) = \begin{cases} 
1 & \text{if } n = 0, \\
1 + \sum_{j=0}^{n-1} T(j) & \text{if } n > 1. 
\end{cases}
```

- $T(n) = 2^n$
  - Exponential growth!
Top-down with Memorization

- Solve recursively, but store each result in an array, \( r[n] \)
  - time-memory trade-off.

\[
\text{MEMOIZED-CUT-Rod}(p, n) \\
\text{let } r[0..n] \text{ be a new array} \quad \text{// initialize } r \\
\text{for } i = 0 \text{ to } n \\
\quad r[i] = -\infty \\
\text{return } \text{MEMOIZED-CUT-Rod-Aux}(p, n, r)
\]

\[
\text{MEMOIZED-CUT-Rod-Aux}(p, n, r) \\
\quad \text{if } r[n] \geq 0 \quad \text{// } r[n] \text{ has been solved} \\
\quad \text{return } r[n] \\
\quad \text{if } n == 0 \\
\quad \quad q = 0 \\
\quad \text{else } q = -\infty \\
\quad \quad \text{for } i = 1 \text{ to } n \\
\quad \quad \quad q = \max(q, p[i] + \text{MEMOIZED-CUT-Rod-Aux}(p, n - i, r)) \\
\quad r[n] = q \\
\quad \text{return } q
\]

Step3: Compute Opt. Solution (bottom-up)

- Sort the subproblems by size and solve the smaller ones first.

\[
\text{BOTTOM-UP-CUT-Rod}(p, n) \\
\text{let } r[0..n] \text{ be a new array} \\
\text{r[0]} = 0 \\
\text{for } j = 1 \text{ to } n \\
\quad q = -\infty \\
\quad \text{for } i = 1 \text{ to } j \\
\quad \quad q = \max(q, p[i] + r[j - i]) \\
\quad r[j] = q \quad \text{// memorization} \\
\text{return } r[n]
\]

- running time \( \Theta(n^2) \)
  - doubly nested loop
  - same as Top-down approach
  - use aggregate analysis in ch 17
Step 4: Reconstruct Optimal Solution

- Modify BOTTOM-UP-CUT-ROD
  - \( s[j] \) = the optimal size of first piece to cut for rod size \( j \)

**Extended-Bottom-Up-Cut-Rod** \((p, n)\)

let \( r[0..n] \) and \( s[0..n] \) be new arrays

\[ r[0] = 0 \]

for \( j = 1 \) to \( n \)

\[ q = -\infty \]

for \( i = 1 \) to \( j \)

if \( q < p[i] + r[j - i] \)

\[ q = p[i] + r[j - i] \]

\[ s[j] = i \]  // store size

\[ r[j] = q \]

return \( r \) and \( s \)

Exercise

- how to modify the top-down approach to save size?
Step 4: Reconstruct Optimal Solution (2)

- print size \( s[n] \), then continue with \( n = n - s[n] \)

\[
\text{PRINT-CUT-Rod-Solution}(p, n) \\
(r, s) = \text{EXTENDED-BOTTOM-UP-CUT-Rod}(p, n) \\
\text{while } n > 0 \\
\quad \text{print } s[n] \\
\quad n = n - s[n]
\]

- Example: \( n=7 \)
  - \( s[7] = 1 \)
  - \( s[6] = 6 \)

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( i )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>( s[i] )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>( r[i] )</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>10</td>
<td>13</td>
<td>17</td>
<td>18</td>
<td>22</td>
<td>25</td>
</tr>
</tbody>
</table>

- Exercise: \( n = 9 \)
- Exercise: \( n = 10 \)

Four steps of DP (revisit)

- Four steps
  - 1. Characterize the structure of an optimal solution.
    * \( \text{optimal substructure} \)
    * imagine that \( \text{DP God} \) promised to (1) give you a magic tool that can solve a smaller problem (2) to tell you the last choice in the optimal solution...
  - 2. \( \text{Recursively} \) define the value of an optimal solution
    * image that \( \text{DP God} \) is too busy to tell you the last choice .... so you have to search all possibilities
  - 3. Compute the value of an optimal solution,
    * typically in a \( \text{bottom-up} \) fashion.
    * memorize solutions to many \( \text{overlapping subproblems} \)
  - 4. Construct an optimal solution from step 3.
Outline

- Dynamic Programming, CH15
  - rod cutting
  - longest common subsequence (LCS)
  - optimal binary search tree
  - elements of DP
  - Matrix Chain (self Study)
  - Conclusion
- Greedy Algorithms, CH16
- Amortized Analysis, CH17

Longest Common Subsequence (LCS)

- Problem: Given 2 sequences, \( X = <x_1, \ldots, x_m> \) and \( Y = <y_1, \ldots, y_n> \).
  - Find a subsequence common to both whose length is longest
  - subsequence doesn’t have to be consecutive
  - but it has to be in order.
- Example 1:

- Example 2:
  
  \[
  S_1 = \text{ACCGTACGATTGCCGTCGCGAACCGCCGCGC}
  \]
  
  \[
  S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAC}
  \]
  
  \[
  \text{LCS} = \text{GTCGT CCGAA GCGC GC CG AA}
  \]
**Brute Force Approach**

- For every subsequence of $X$, check whether it’s a subsequence of $Y$
- Time: $\Theta(n \cdot 2^m)$
  - $2^m$ subsequences of $X$ to check
  - Each subsequence takes $\Theta(n)$ time to check
    - scan $Y$ for first letter, from there scan for second, and so on.

**Step 1: Optimal Substructure**

- Notation:
  - $X_i = \text{prefix } <x_1, \ldots, x_i>$
  - $Y_i = \text{prefix } <y_1, \ldots, y_i>$
- Theorem 15.1
  - Let $Z = <z_1, \ldots, z_k>$ be any LCS of $X = <x_1, \ldots, x_m>$ and $Y = <y_1, \ldots, y_n>$
    - 1. If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$
    - 2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies $Z$ is an LCS of $X_{m-1}$ and $Y$
    - 3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies $Z$ is an LCS of $X$ and $Y_{n-1}$
- **Optimal substructure**: If $Z$ is an LCS of $X$ and $Y$, and $Z$ contains a subsequence $Z'$, then $Z'$ must be LCS of prefix of $X$ and $Y$
- Proof: cut and paste
  - If $Z'$ is not LCS, replace it by LCS so $Z$ is not longest
  
  $Z \underbrace{\text{match}}_{\text{also an LCS}} Z'$
Step 2: Recursive Solution

- \( c[i, j] = \) LCS length of \( X_i \) and \( Y_j \)
  \[
  c[i, j] = \begin{cases} 
  0 & \text{if } i = 0 \text{ or } j = 0 \\
  c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \quad \text{match: } c++ \\
  \max(c[i, j-1], c[i-1, j]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \quad \text{no match: } c = \text{best solution so far}
  \end{cases}
  \]

- Example: bozo, bat
  - many overlapping subproblems

Step 3: Compute Optimal Solution (1)

LCS-LENGTH(\( X, Y, m, n \))

let \( b[1..m, 1..n] \) and \( c[0..m, 0..n] \) be new tables

\[
\text{for } i = 1 \text{ to } m \quad \text{// initialize } b \text{ and } c \\
c[i, 0] = 0
\]

\[
\text{for } j = 0 \text{ to } n \\
c[0, j] = 0
\]

\[
\text{for } i = 1 \text{ to } m \\
\text{for } j = 1 \text{ to } n \\
\quad \text{if } x_i = y_j \\
\quad \quad c[i, j] = c[i-1, j-1] + 1 \\
\quad \quad b[i, j] = "\downarrow" \\
\quad \text{else if } c[i-1, j] \geq c[i, j-1] \\
\quad \quad c[i, j] = c[i-1, j] \\
\quad \quad b[i, j] = "\uparrow" \\
\quad \text{else } c[i, j] = c[i, j-1] \\
\quad \quad b[i, j] = "\leftarrow"
\]

\[
\text{return } c \text{ and } b
\]

\( b[i, j] \) points to optimal solution of subproblem
Step 3: Compute Optimal Solution (2)

A[\text{LCS-LENGTH}(X, Y, m, n)]

let $b[1 \ldots m, 1 \ldots n]$ and $c[0 \ldots m, 0 \ldots n]$ be new tables

\begin{align*}
\text{for } i &= 1 \text{ to } m \\
&\quad c[i, 0] = 0 \\
\text{for } j &= 0 \text{ to } n \\
&\quad c[0, j] = 0 \\
\text{for } i &= 1 \text{ to } m \\
&\quad \text{for } j = 1 \text{ to } n \\
&\quad \text{if } x_i = y_j \\
&\quad \quad c[i, j] = c[i-1, j-1] + 1 \\
&\quad \quad b[i, j] = "\downarrow" \\
&\quad \text{else if } c[i-1, j] \geq c[i, j-1] \\
&\quad \quad c[i, j] = c[i-1, j] \\
&\quad \quad b[i, j] = "\rightarrow" \\
&\quad \text{else if } c[i, j-1] \geq c[i-1, j] \\
&\quad \quad c[i, j] = c[i, j-1] \\
&\quad \quad b[i, j] = "\leftarrow" \\
&\quad \text{return } c \text{ and } b
\end{align*}

Time Complexity

\begin{align*}
\text{LCS-LENGTH}(X, Y, m, n) \\
\text{let } b[1 \ldots m, 1 \ldots n] \text{ and } c[0 \ldots m, 0 \ldots n] \text{ be new tables} \\
\text{for } i &= 1 \text{ to } m \\
&\quad c[i, 0] = 0 \\
\text{for } j &= 0 \text{ to } n \\
&\quad c[0, j] = 0 \\
\text{for } i &= 1 \text{ to } m \\
&\quad \text{for } j = 1 \text{ to } n \\
&\quad \text{if } x_i = y_j \\
&\quad \quad c[i, j] = c[i-1, j-1] + 1 \\
&\quad \quad b[i, j] = "\downarrow" \\
&\quad \text{else if } c[i-1, j] \geq c[i, j-1] \\
&\quad \quad c[i, j] = c[i-1, j] \\
&\quad \quad b[i, j] = "\rightarrow" \\
&\quad \text{else if } c[i, j-1] \geq c[i-1, j] \\
&\quad \quad c[i, j] = c[i, j-1] \\
&\quad \quad b[i, j] = "\leftarrow" \\
&\quad \text{return } c \text{ and } b
\end{align*}

- doubly nested loop
  - $\Theta(mn)$
Step 4: Constructing Optimal Solution

- X=ABCBDAB
- Y=BDCABA
- LCS = BCBA
- O(m+n)

```
PRINT-LCS(b, X, i, j)
if i == 0 or j == 0
    return
if b[i, j] == “\”
    PRINT-LCS(b, X, i - 1, j - 1)
    print x_i
else if b[i, j] == “↑”
    PRINT-LCS(b, X, i - 1, j)
else PRINT-LCS(b, X, i, j - 1)
```

Outline

- Dynamic Programming, CH15
  - rod cutting
  - longest common subsequence
  - optimal binary search tree
  - elements of DP
  - Matrix Chain (self Study)
  - Conclusion
- Greedy Algorithms, CH16
- Amortized Analysis, CH17
Optimal Binary Search Tree (BST)

- $K = \langle k_1, k_2, ..., k_n \rangle$ are $n$ distinct keys in sorted order
  - $k_1 < k_2 < ... < k_n$
- $P = \langle p_1, p_2, ..., p_n \rangle$ are probability that $k_i$ is searched
- $D = \langle d_0, d_1, ..., d_n \rangle$ are $n+1$ dummy keys for unsuccessful searches
- $Q = \langle q_0, q_1, q_2, ..., q_n \rangle$ are probability that $d_i$ is searched

\[ \sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1 \]

- **expected cost of search**

\[ E[\text{search cost in } T] = \sum_{i=0}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i \]

\[ = 1 + \sum_{i=1}^{n} \text{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \text{depth}_T(d_i) \cdot q_i \]

- **Optimal BST:** the BST of lowest cost of search

**Example**

- cost of left tree = 2.80; cost of right tree = 2.75 $\rightarrow$ optimal BST

![Binary Trees](image)

**Figure 15.7** Two binary search trees for a set of $n = 5$ keys with the following probabilities:

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_i$</td>
<td>0.15</td>
<td>0.10</td>
<td>0.05</td>
<td>0.10</td>
<td>0.20</td>
<td></td>
</tr>
<tr>
<td>$q_i$</td>
<td>0.05</td>
<td>0.10</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.10</td>
</tr>
</tbody>
</table>

(a) A binary search tree with expected search cost 2.80. (b) A binary search tree with expected search cost 2.75. This tree is optimal.
Step 1: Optimal Substructure

- If an optimal BST $T$ contains subtree $T'$ with keys $k_i,..., k_j$,
  - $T'$ must be optimal BST for keys $k_i,..., k_j$ and dummy keys $d_{i-1},...,d_j$
- Proof: cut and paste
  - If $T'$ is NOT optimal, then we just replace $T'$ by an optimal one
    * Violate our assumption that $T$ is optimal

Cost of $T'$

- Given a subtree $T'$, $k_r$ is the root, where $i \leq r \leq j$
  - Left subtree of $k_r$ contains $k_i,..., k_{r-1}$
  - Right subtree of $k_r$ contains $k_{r+1},..., k_j$
- we’re guaranteed to find an optimal subtree $T'$ for $k_i,..., k_j$ as long as
  - 1. we examine all candidate roots $k_r$, $i \leq r \leq j$, and
  - 2. we determine all optimal BSTs containing $k_i,..., k_{r-1}$ and containing $k_{r+1},..., k_j$,
Step 2: Recursive Solution

- $e[i, j] = \text{expected search cost of optimal subtree } T', \ k_p, ..., k_j$
- $w[i, j] = \text{sum of probabilities of subtree } T'$

\[
w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l
\]

- If $k_r$ is the root of optimal subtree $T'$

\[
e[i, j] = p_r + (e[i, r - 1] + w(i, r - 1)) + (e[r + 1, j] + w(r + 1, j))
\]

\[
\therefore w(i, j) = w(i, r - 1) + p_r + w(r + 1, j)
\]

\[
\therefore e[i, j] = e[i, r - 1] + e[r + 1, j] + w(i, j)
\]

Step 2: Recursive Solution (cont’d)

- But we do not know which $r$ is optimal, so try all $r$ (eq 15.14)

\[
e[i, j] = \begin{cases} 
q_{i-1} & \text{if } j = i - 1 \\
\min_{i \leq r \leq j} \{e[i, r - 1] + e[r + 1, j] + w(i, j)\} & \text{if } i \leq j
\end{cases}
\]

NOTE: $j = i - 1$ is an empty subtree containing only $d_{i-1}$ so cost = $q_i$
Step 3: Compute Optimal Solution (1)

**OPTIMAL-BST** \((p, q, n)\)

let \(e[1..n + 1, 0..n]\), \(w[1..n + 1, 0..n]\), and \(\text{root}[1..n, 1..n]\) be new tables

for \(i = 1 \text{ to } n + 1\)

\[
e[i, i-1] = \begin{cases} \quad q_{i-1} \quad & \text{empty subtree} \\ \quad w[i, i-1] = q_{i-1} \end{cases}
\]

for \(l = 1 \text{ to } n\)

for \(i = 1 \text{ to } n - l + 1\)

\(j = i + l - 1\)

\(e[i, j] = \infty\)

\[w[i, j] = w[i, j-1] + p_j + q_j\]

for \(r = i \text{ to } j\)

\[t = e[i, r-1] + e[r+1, j] + w[i, j]\]

if \(t < e[i, j]\)

\[e[i, j] = t\]

\[\text{root}[i, j] = r\]

return \(e\) and \(\text{root}\)

---

Step 3: Compute Optimal Solution (2)

**OPTIMAL-BST** \((p, q, n)\)

let \(e[1..n + 1, 0..n]\), \(w[1..n + 1, 0..n]\), and \(\text{root}[1..n, 1..n]\) be new tables

for \(i = 1 \text{ to } n + 1\)

\[
e[i, i-1] = q_{i-1} \quad \text{empty subtrees}
\]

for \(l = 1 \text{ to } n\)

for \(i = 1 \text{ to } n - l + 1\)

\(j = i + l - 1\)

\(e[i, j] = \infty\)

\[w[i, j] = w[i, j-1] + p_j + q_j\]

for \(r = i \text{ to } j\)

\[t = e[i, r-1] + e[r+1, j] + w[i, j]\]

if \(t < e[i, j]\)

\[e[i, j] = t\]

\[\text{root}[i, j] = r\]

return \(e\) and \(\text{root}\)
Step 3: Compute Optimal Solution (3)

OPTIMAL-BST \((p, q, n)\)

let \(e[1..n + 1, 0..n]\), \(w[1..n + 1, 0..n]\), and \(root[1..n, 1..n]\) be new tables

for \(i = 1\) to \(n + 1\)

\[
e[i, i - 1] = 0
\]

\[
w[i, i - 1] = 0
\]

for \(l = 1\) to \(n\)

for \(i = 1\) to \(n - l + 1\)

\[
j = i + l - 1
\]

\[
e[i, j] = \infty
\]

\[
w[i, j] = w[i, j - 1] + p_j
\]

for \(r = i\) to \(j\)

\[
t = e[i, r - 1] + e[r + 1, j] + w[i, j]
\]

if \(t < e[i, j]\)

\[
e[i, j] = t
\]

\[
root[i, j] = r
\]

return \(e\) and \(root\)

Running Time

- three-level nested loops
  - \(\Theta(n^3)\)
  - Exercise 15.2-5

OPTIMAL-BST \((p, q, n)\)

let \(e[1..n + 1, 0..n]\), \(w[1..n + 1, 0..n]\), and \(root[1..n, 1..n]\) be new tables

for \(i = 1\) to \(n + 1\)

\[
e[i, i - 1] = 0
\]

\[
w[i, i - 1] = 0
\]

for \(l = 1\) to \(n\)

for \(i = 1\) to \(n - l + 1\)

\[
j = i + l - 1
\]

\[
e[i, j] = \infty
\]

\[
w[i, j] = w[i, j - 1] + p_j
\]

for \(r = i\) to \(j\)

\[
t = e[i, r - 1] + e[r + 1, j] + w[i, j]
\]

if \(t < e[i, j]\)

\[
e[i, j] = t
\]

\[
root[i, j] = r
\]

return \(e\) and \(root\)
Step 4: Construct Optimal BST

- Your exercise!

Outline

- **Dynamic Programming, CH15**
  - rod cutting
  - longest common subsequence
  - optimal binary search tree
  - elements of DP
  - Matrix Chain (self Study)
  - Conclusion
- **Greedy Algorithms, CH16**
- **Amortized Analysis, CH17**
Elements of DP

- Two key elements
  - **optimal substructure**
    - the solutions to the subproblems used within the optimal solution must themselves be optimal.
  - **overlapping subproblems**
    - recursively solve the same subproblems over and over again
      - not brand new subproblems

Optimal Substructures

- Show that the solutions to the subproblems used within the optimal solution must themselves be optimal.
  - How to prove? Usually use “cut-and-paste”
- Two questions to ask:
  - 1. How many subproblems are used in an optimal solution.
  - 2. How many choices in determining which subproblem(s) to use.
- Example:
  - rod cutting:
    - 1 subproblem of size \( n-i \), for \( 1 \leq i \leq n \).
    - less than \( n \) choices for each subproblem
  - LCS
    - 1 subproblem
    - 1 choice (if \( x_i = y_j \), LCS of \( X_{i-1} \) and \( Y_{j-1} \)), or
    - 2 choices (if \( x_i \neq y_j \), LCS of \( X_{i-1} \) and \( Y \), and LCS of \( X \) and \( Y_{j-1} \))
  - optimal BST
    - 2 subproblems \( (k_1, \ldots, k_{r-1}) \) and \( (k_{r+1}, \ldots, k_j) \)
    - \( j-i+1 \) choices for \( k_r \) in \( k_i, \ldots, k_j \)
Running Time

- Running time depends on
  - (# of subproblems overall) times (# of choices)

- Examples:
  - rod cutting
    - $\Theta(n)$ subproblems, $\leq n$ choices for each
    - $O(n^2)$ running time
  - LCS
    - $\Theta(mn)$ subproblems, 2 choices for each
    - $\Theta(mn)$ running time
  - OBST
    - $\Theta(n^2)$ subproblems, $\Theta(n)$ choices for each
    - $\Theta(n^3)$ running time

Subtleties

- Do not assume the optimal substructure applies when it does not
- Shortest path: find simple path (no cycle) $u \rightarrow v$ with fewest edges
  - it has optimal substructure
    - any subpath is optimal
      - $p_1$ is shortest
      - $p_2$ is shortest

- Longest simple path: find simple path $u \rightarrow v$ with most edges.
  - it does NOT has optimal substructure
    - Q1: what is longest path
      - $q \rightarrow t$
    - Q2: what are longest paths
      - $q \rightarrow r$
      - $r \rightarrow t$
    - Q1 does NOT contain Q2
  - Why? because simple longest paths are NOT independent
Overlapping Subproblems

- DP recursively solve the same subproblems over and over again
  - Example: rod cutting

- NOTE: divide-and-conquer generates brand new subproblems
  - Example: merge sort is not DP

Outline

- Dynamic Programming, CH15
  - rod cutting
  - longest common subsequence
  - optimal binary search tree
  - elements of DP
  - Matrix Chain (self Study)
  - Conclusion
- Greedy Algorithms, CH16
- Amortized Analysis, CH17
Matrix Chain Multiplication

- Given \( A = A_1A_2 \ldots A_n \)
  - Find \( A \) using the minimum number of multiplications
- A product of Matrices is **fully parenthesized** if it is a single matrix or the produce of 2 fully parenthesized matrix products
- Example:
  - \((A_1(A_2(A_3A_4)))\)
  - \((A_1((A_2A_3)A_4))\)
  - ...
- \( C=AB \)
  - if size \( A \) is \( pxq \), size \( B \) is \( qxr \) then \( pqr \) multiplications is needed
- The order of multiplication makes a big difference
  - Example: \( A1: 10\times100, A2: 100\times5, A3: 5\times50 \)
    - \((A1A2)A3\) needs 7,500 multiplications
    - \(A1(A2A3)\) needs 75,000 multiplications!

---

Brute Force

- Check all possible orders
- \( P(n) \): number of ways to multiply \( n \) matrices.

\[
P(n) = \begin{cases} 
1 & \text{if } n = 1 \\
\sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2 
\end{cases}
\]

- \( P(n) \) is a sequence of **Catalan** numbers, which grows exponentially!
  - \( \Omega(4^n /n^{3/2}) \)
- How to solve it smartly? using DP
Outline

- Dynamic Programming, CH15
  - rod cutting
  - longest common subsequence
  - optimal binary search tree
  - elements of DP
  - Matrix Chain (self Study)
  - Conclusion
- Greedy Algorithms, CH16
- Amortized Analysis, CH17

Compare Two Approaches

- 1. Bottom-up iterative approach
  - Start with recursive divide-and-conquer algorithm.
  - Find the dependencies between the subproblems (which solutions are needed for computing a subproblem).
  - Solve the subproblems in the correct order.
- 2. Top-down recursive approach (memorization)
  - Start with recursive divide-and-conquer algorithm.
  - Keep top-down approach of original algorithms.
  - Save solutions to subproblems in a table
  - Recurse only on a subproblem if the solution is not already available in the table.
- If all subproblems must be solved at least once, bottom-up DP is better due to less overhead for recursion and for maintaining tables.
- If many (but not all) subproblems need not be solved, top-down DP is better since it computes only those required.
When to Use DP

- DP computes recurrence efficiently by storing partial results
  - It is efficient only when small number of partial results.
- DP is NOT suitable for:
  - $n!$ permutations of an $n$-element set,
  - $2^n$ subsets of an $n$-element set, etc.
- DP is suitable for:
  - contiguous substrings of an $n$-character string,
  - $n(n+1)/2$ possible subtrees of a binary search tree, etc.
- DP works best: objects that are linearly ordered and cannot be arranged.
  - Matrices in a chain,
  - characters in a string

Outline

- Dynamic Programming, CH15
- Greedy Algorithms, CH16
  - Activities Selection Problem
  - Channel Routing Problem* (not in exam)
  - Elements of Greedy Strategy
  - Huffman Codes
- Amortized Analysis, CH17
**Activity Selection Problem**

- $n$ activities require exclusive use of a common resource.
  - For example, scheduling the use of a classroom.
- Set of activities $S = \{a_1, \ldots, a_n\}$.
  - $a_i$ needs resource during period $[s_i, f_i]$, where $s_i$ = start time and $f_i$ = finish time.
- Goal: Select the largest possible set of non-overlapping activities
  - $a_i$ and $a_j$ are *compatible* if their intervals do not overlap

- Assume that activities are sorted by finish time:
  - $f_1 \leq f_2 \leq f_3 \leq \ldots \leq f_{n-1} \leq f_n$

**Example**

- $S$ sorted by finish time

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>11</td>
<td>13</td>
</tr>
<tr>
<td>$f_i$</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

- Possible solution $\{a_1, a_3, a_9\}$: not maximum
- Maximum-size mutually compatible set: $\{a_1, a_3, a_6, a_8\}$.
  - Not unique: also $\{a_2, a_5, a_7, a_9\}$. 
Step 1: Optimal Substructure

- \( S_{ij} \) = set of activities that
  - starts after \( a_i \) finished and finishes before \( a_j \) starts
  - \( \{a_k \in S ; f_i \leq s_k < f_k \leq s_j \} \)
- Let \( A_{ij} \) be a maximum-size set of mutually compatible activities in \( S_{ij} \)
- Suppose we already know \( a_k \in A_{ij} \). Then we have two subproblems:
  - Find \( A_{ik} = A_{ij} \cap S_{ik} \) = set of activities in \( A_{ij} \) that finish before \( a_k \)
  - Find \( A_{kj} = A_{ij} \cap S_{kj} \) = set of activities in \( A_{ij} \) that start after \( a_k \)
  - Then \( A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj} \)
- Optimal solution \( A_{ij} \) must include optimal solutions for the two subproblems for \( S_{ik} \) and \( S_{kj} \).
- Proof: cut and paste
  - Suppose we could find a better set \( A_{kj}' \) in \( S_{kj} \), where \( |A_{kj}'| > |A_{kj}| \).
  - Then use \( A_{kj}' \) instead of \( A_{kj} \)

\[
\begin{array}{c|c|c|c}
& f_i & s_k & f_k \\
\hline
a_i & \cdot & \cdot & \cdot \\
\hline
a_k & \cdot & \cdot & \cdot \\
\hline
a_j & \cdot & \cdot & \cdot \\
\hline
\end{array}
\]

Step 2: Recursive Solution

- Let \( c[i, j] = |A_{ij}| \) = size of optimal solution in \( S_{ij} \)
  - \( c[i, j] = c[i, k] + c[k, j] + 1 \)
- But actually we do not know \( a_k \)
  - exhaustively try all possible \( a_k \)

\[
c[i, j] = \begin{cases} 
0 & \text{if } s_{ij} = \phi \\
\max \{c[i, k] + c[k, j] + 1\} & \text{if } s_{ij} \neq \phi 
\end{cases} 
\]

- We could solve the problem like DP
  - but we ignore some important characteristic of this problem
  - can we solve the problem before solving the subproblems?
Early Bird Always Wins

- (Theorem 16.1) If $S_k$ is nonempty and $a_m$ has the earliest finish time in $S_k$, then $a_m$ is included in some optimal solution
  - Proof: cut and paste
  - Let $A_k$ be an optimal solution to $S_k$
  - Let $a_j$ have the earliest finish time of any activity in $A_k$
    - If $a_j = a_m$, done
    - Otherwise, let $A_k' = A_k \setminus \{a_j\} \cup \{a_m\}$
      - substitute $a_m$ for $a_j$.
  - activities in $A_k'$ are disjoint because activities in $A_k$ are disjoint
    - $a_j$ is first activity in $A_k$ to finish so $f_m \leq f_j$
  - Since $|A_k'| = |A_k|$, conclude that $A_k'$ is an optimal solution to $S_k$, and it includes $a_m$

- So, don’t need dynamic programming. Don’t need to work bottom up
- Instead, can just repeatedly choose the activity that finishes first,
  - keep only the activities that are compatible with that one, and repeat until no activities remain

Recursive Greedy Algorithm

- $s =$ start time array
- $f =$ finish time array, sorted in monotonically increasing order.
- $k =$ index of currently selected activity
- $n =$ numbers of total activities
- initial call REC-ACTIVITY-SELECTOR ($s$, $f$, 0, $n$)

```
REC-ACTIVITY-SELECTOR($s$, $f$, $k$, $n$)

$m = k + 1$

while $m \leq n$ and $s[m] < f[k]$ // find the first activity in $S_k$ to finish
  $m = m + 1$

if $m \leq n$
  return $\{a_m\} \cup$ REC-ACTIVITY-SELECTOR($s$, $f$, $m$, $n$)
else return $\emptyset$
```

- time complexity = ?
Iterative Greedy Algorithm

- \( s = \text{start time array} \)
- \( f = \text{finish time array, sorted in monotonically increasing order.} \)
- same time complexity as before
- but less overhead of recursive call of functions

\[
\text{GREEDY-ACTIVITY-SELECTOR}(s, f)
\]
\[
\begin{align*}
&n = s.\text{length} \\
&A = \{a_1\} \\
&k = 1 \\
&\text{for } m = 2 \text{ to } n \\
&\quad \text{if } s[m] \ge f[k] \\
&\quad \quad A = A \cup \{a_m\} \\
&\quad \quad k = m \\
&\text{return } A
\end{align*}
\]

Outline

- Dynamic Programming, CH15
- Greedy Algorithms, CH16
  - Activities Selection Problem
  - Channel Routing Problem* (not in exam)
  - Elements of Greedy Strategy
  - Huffman Codes
- Amortized Analysis, CH17
Standard Cell Design

- Logic gates are pre-designed cells of the same height
- Interconnects are routed in routing channels
- Very flexible and scalable design style

Terminology for Channel Routing

- Terminals of the same number should be connected together
- Every track has one
Channel Routing Problem

- Very important EDA problem
- Assignments of horizontal segments to *tracks*
- All terminals must be connected
- *Horizontal constraints* must not be violated
  - horizontal span of two nets CANNOT overlaps each other
- Objective: Channel height is minimized
  - i.e., minimized the number of tracks

Left-Edge Algorithm

- HV-layer model is used.
- Treat each horizontal segment as an *interval*.
- Intervals are sorted according to their left-end *x*-coordinates.
- Intervals are routed one-by-one according to the order.
- For a net, tracks are scanned from top to bottom,
  - the first track that can accommodate the net is assigned to the net.
- Always produces an optimal routing solution with the minimum # of tracks
  - if no vertical constraint
Left-Edge Algorithm

Algorithm: Basic_Left-Edge(U, track[j])

U: set of unassigned intervals (nets) I₁, ..., Iₙ;
Iⱼ=[sⱼ, eⱼ]: interval j with left-end x-coordinate sⱼ and right-end eⱼ;
track[j]: track to which net j is assigned.

1 begin
2 U ← {I₁, I₂, ..., Iₙ};
3 t ← 0;
4 while (U ≠ ∅) do
5      t ← t + 1;
6      watermark ← 0;
7      while (there is an Iⱼ ∈ U s.t. sⱼ > watermark) do
8          Pick the interval Iⱼ ∈ U with sⱼ > watermark, nearest watermark;
9          track[j] ← t;
10         watermark ← eⱼ;
11        U ← U - {Iⱼ};
12     end
13 end

Basic Left-Edge Example

- U = {l₁, l₂, ..., l₆}
  - l₁ = [1, 3], l₂ = [2, 6], l₃ = [4, 8], l₄ = [5, 10], l₅ = [7, 11], l₆ = [9, 12].
- t = 1:
  - Route l₁: watermark = 3;
  - Route l₃: watermark = 8;
  - Route l₆: watermark = 12;
- t = 2:
  - Route l₂: watermark = 6;
  - Route l₅: watermark = 11;
- t = 3: Route l₄
Outline

- Dynamic Programming, CH15
- **Greedy Algorithms, CH16**
  - Activities Selection Problem
  - Channel Routing Problem* (not in exam)
  - Elements of Greedy Strategy
  - Huffman Codes
- Amortized Analysis, CH17

---

Elements of Greedy Algorithms

- Six steps to develop a greedy algorithm
  1. Determine the *optimal substructure*.
  2. Develop a recursive solution.
  3. Show that if we make the greedy choice, only one subproblem remains.
  4. Prove that it’s always safe to make the greedy choice.
     * greedy choice property
  5. Develop a recursive greedy algorithm.
  6. Convert it to an iterative algorithm.
Key Ingredients of Greedy

1. Greedy-choice Property
   • Can assemble a globally optimal solution by making locally optimal (greedy) choices.
   • Example: activity selection:
     ✤ Look at an optimal solution.
     ✤ If it includes the greedy choice, done.
     ✤ Otherwise, modify the optimal solution to include the greedy choice, yielding another solution that’s just as good.
   • different from DP

2. Optimal Substructure
   • Optimal solution to subproblem is included in optimal solution to the whole problem
   • same as DP

DP vs. Greedy

• Dynamic programming
  ✤ Make a choice at each step
  ✤ Choice depends on knowing optimal solutions to subproblems
  ✤ Solve subproblems first — bottom-up

• Greedy
  ✤ Make a choice at each step
  ✤ Make the choice before solving the subproblems
  ✤ Solve problems top-down

• Both greedy and DP exploits optimal substructure
  ✤ sometimes we can get confused
• Two examples to demonstrate the difference
  ✤ Fractional Knapsack Problem
  ✤ 0-1 Knapsack Problem
Fractional Knapsack Problem

- A thief has $n$ items to take
- Item $i$ is worth $v_i$, weighs $w_i$ pounds.
- Find a most valuable subset of items with total weight $\leq W$
- *Can take a fraction of an item*
- Example: Fig. 16.2
  - optimal solution: item 1 + item2 + 2/3 item 3
- Greedy algorithm applies

![Diagram of Fractional Knapsack Problem with item weights and values]

Greedy Algorithm

- Rank items by value/weight: $v_i/w_i$.
- Take items in decreasing order of value/weight.
- Possibly a fraction of the item
- Time complexity:
  - $O(n \lg n)$ to sort, $O(n)$ thereafter.

```
FRACTIONAL-KNAPSACK(v, w, W)

load = 0
i = 1

while load < W and i ≤ n
    if $w_i \leq W - load$
        take all of item $i$
    else take $(W - load)/w_i$ of item $i$
    add what was taken to load
    i = i + 1
```

<table>
<thead>
<tr>
<th>i</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_i$</td>
<td>60</td>
<td>100</td>
<td>120</td>
</tr>
<tr>
<td>$w_i$</td>
<td>10</td>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>$v_i/w_i$</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

$W = 50$. 

---

74
0-1 Knapsack Problem

- Have to either take an item or not take it — can’t take part of it
- Example: Fig. 16.2
  - optimal solution: item3 + item2
- Greedy strategy does NOT apply
- Need to use DP
  - \( O(nW) \) time, exercise 16.2-2

### WHY?

- Both have optimal substructure.
- But fractional knapsack problem has the greedy-choice property
  - the 0-1 knapsack problem does not
- FFT
  - In 0-1 Knapsack problem, what if we take items in the order of their values? Does greedy algorithm work in this way?
    - Yes, prove the greedy-choice property
    - No, give an counter example
Outline

- Dynamic Programming, CH15
- Greedy Algorithms, CH16
  - Activities Selection Problem
  - Channel Routing Problem* (not in exam)
  - Elements of Greedy Strategy
  - Huffman Codes
- Amortized Analysis, CH17

David A. Huffman [1925-1999 USA]

- In 1951 Huffman (aged 25) and his classmates in an electrical engineering graduate course on information theory were given the choice of a term paper or a final exam. Huffman’s professor, Robert M. Fano, had assigned what at first appeared to be a simple problem. Students were asked to find the most efficient method of representing numbers, letters or other symbols using a binary code.

- Huffman worked on the problem for months, developing a number of approaches, but failed. Just as he was throwing his notes in the garbage, the solution came to him.

- Huffman says he might never have tried his hand at the problem if he had known that Fano, his professor, and Claude E. Shannon, the creator of information theory, had struggled with it.
Coding

- Application: data compression, error correction, encryption etc
- Binary Character Code: each character represented by a unique binary string (codeword)
  - Fixed-length code: codeword are the same length
    * e.g. a=000, b=001, ... f=101
    - afb=000 101 001
  - Variable-length code: codeword are different in length
    * e.g. Huffman code
      - frequent characters are represented by shorter codeword

- Example: Fig 16.3, 100K characters (a ~ f)
  * fixed length coding: 300K bits
  * variable length coding: 224 K bits

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (in thousands)</td>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>Fixed-length codeword</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>Variable-length codeword</td>
<td>0</td>
<td>101</td>
<td>100</td>
<td>111</td>
<td>1101</td>
<td>1100</td>
</tr>
</tbody>
</table>

Prefix Code

- Prefix Code: No code is a prefix of some other code
- Example
  - C={a, b, c, d, e, f}
  - C is called alphabet, the set of characters in use

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency (in thousands)</td>
<td>45</td>
<td>13</td>
<td>12</td>
<td>16</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>Fixed-length codeword</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
</tr>
<tr>
<td>Variable-length codeword</td>
<td>0</td>
<td>101</td>
<td>100</td>
<td>111</td>
<td>1101</td>
<td>1100</td>
</tr>
</tbody>
</table>

- Example of none-prefix code
  - a=000, b = 0001
- Encoding is always simple
  - just concatenation
    * e.g. afb=0 1100 101
- Decoding is easy when prefix code is used
  - decode each character is unambiguous
    * e.g. 001011101 → 0 0 101 1101
Tree Representation

- decoding process of prefix code
  - codeword is represented by a simple path from root to leave

Optimal Prefix Code Design Problem

- Input: given a set of alphabet and the frequency of each character
  - Given a tree which represents a prefix code, cost of the tree \( T \)
  - \( c \_\text{freq} \) is the frequency of appearance of character \( c \)
  - \( d_T(c) \) is depth of \( c \)’s leaf in the tree = length of \( c \)’s codeword

- Output: find a binary tree such that cost is minimized
  \[ B(T) = \sum_{c \in C} c \_\text{freq} \cdot d_T(c) \]

- Optimal prefix code is always represented by a full binary tree
  - every non-leaf node has two children
  - optimal tree has \(|C|\) leaves and \(|C|\)-1 nodes

- Example (a) cost = 300, not full (b) =224, optimal cost, full tree
Huffman’s Algorithm

- Idea: more frequent characters use shorter depth
  - combine two nodes with the least cost at each step

Huffman’s Pseudo Code

- Time complexity: \( Q \) is implemented in min-heap
  - loop \( n-1 \) times, each EXTRACT-MIN is \( O(\lg n) \)
  - totally \( O(n \lg n) \)
Greedy-choice Property

- Lemma 16.2: Two characters $x$ and $y$ with the lowest frequencies must have the same length and differ only in the last bit.
  - Proof: suppose tree $T$ is optimal
    * swapping $x$ with $a$, $y$ with $b$ (does not increase cost)
    * resulting in $T''$ with lower cost than $T$
  - Therefore, building an optimal tree can start with greedy choice of merging together two characters of lowest frequency

![Diagram of trees](image)

Figure 16.6: An illustration of the key step in the proof of Lemma 16.2. In the optimal tree $T$, if we swap $x$ with $a$ and $y$ with $b$, we get a tree $T''$ with lower cost than $T$.

Optimal Substructure

- Tree $T$ is the tree representing code over $C$
  - $z$ is parent of two leaf characters $x$ and $y$ of minimum frequency in $C$, $z.freq = x.freq + y.freq$
- Tree $T'$ is a tree representing code over $C'$.
  - $C' = C - \{x, y\} \cup \{z\}$
- If tree $T'$ represents optimal prefix code for $C'$,
  - then tree $T$ also represents optimal prefix code $C$
- Proof: cut and paste
  - if there is a better tree $T''$ than $T$ i.e. $B(T'') < B(T)$
  - both $T''$ and $T$ has same leaves $x$ and $y$ as siblings (lemma 16.2)
  - we can replace $x$ and $y$ by $z$ in $T''$
    - get a better tree $T'''$ than $T'$ → contradiction
Optimal Substructure Lemma 16.3 V2

- Tree $T$ is the tree representing code over $C$
  - $z$ is parent of two leaf characters $x$ and $y$ of minimum frequency in $C$, $z.freq = x.freq + y.freq$
- Tree $T'$ is a tree representing code over $C'$.
  - $C' = C - \{x, y\} \cup \{z\}$
- If tree $T$ represents optimal prefix code for $C$,
  - then tree $T'$ also represents optimal prefix code $C'$
- Proof: cut and paste
  - if there is a better tree $T''$ than $T'$ i.e. $B(T'') < B(T')$
  - then we add leaves $x$ and $y$ under $z$ in $T''$
    - get a better tree $T'''$ than $T$ \(\rightarrow\) contradiction

Finally

- Theorem 16.4: HUFFMAN produces an optimal prefix code
  - results from lemma 16.2 and 16.3

- Huffman coding today is often used as back-end to other compression methods
  - PKZIP, JPEG and MP3 uses Huffman coding
- Huffman coding beat his teacher’s method, Shannon-Fano coding
  - you can beat your teachers too!
Reading

- CH15
- CH16