http://www.cs.unh.edu/~ruml/cs758

1 handout: slides
Rod Cutting
Given table of profits $p_i$ for each possible integer length $i$, find the best way to cut a rod of length $n$. Cuts are free, but must be of integer length.

<table>
<thead>
<tr>
<th>length $i$</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>profit $p_i$</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>

$\approx 2^{n-1}$ possible solutions! How to solve in $O(n^2)$ time?
Step 1: write down value of optimal solution

\[
\begin{align*}
    \text{best}(n) & = \text{best profit achievable for length } n \\
    \text{best}(n) & = \max_{\text{first} = 1}^{n} (p_{\text{first}} + \text{best}(n - \text{first})) \\
    \text{best}(0) & = 0
\end{align*}
\]

What is the complexity of the naive recursive algorithm? How to make this efficient?
Step 2: compute optimal value (top-down or bottom-up)

1. best[0] ← 0
2. for len from 1 to \(n\)
   3. \(\text{best}[\text{len}] \leftarrow \max_{\text{first}=1}^{\text{len}} (p_{\text{first}} + \text{best}[\text{len} - \text{first}])\)
4. best[\(n\)]

Will this access uninitialized data?
What is the complexity?
1. best[0] ← 0
2. cut[0] ← 0
3. for len from 1 to \( n \)
4. \( \text{best}[\text{len}] \leftarrow -\infty \)
5. for first from 1 to len
6. \( \text{this} \leftarrow p_{\text{first}} + \text{best}[\text{len} - \text{first}] \)
7. if this > best[len]
8. best[len] ← this
9. cut[len] ← first
10. print best[n]
11. while \( n > 0 \)
12. print cut[n]
13. \( n \leftarrow n - \text{cut}[n] \)
- optimal substructure: global optimum uses optimal solutions of subproblems
- ordering over subproblems: solve ‘smallest’ first, build ‘larger’ from them
- ‘overlapping’ subproblems: polynomial number of subproblems, each possibly used multiple times
- independent subproblems: optimal solution of one subproblem doesn’t affect optimality of another
shortest path

- path to any intermediate vertex along optimal path must be optimal path to that vertex. otherwise, could be shorter.

longest simple path

- path to an intermediate vertex along optimal path may not use vertices used elsewhere: subproblems are not independent.
asst 5
Two-Dimensional Dynamic Programming
Given two strings, \( x \) of length \( m \) and \( y \) of length \( n \), find a common (non-contiguous) subsequence that is as long as possible.

\[
x = \text{ABCBDAB} \\
y = \text{BDCABA}
\]
Given two strings, \(x\) of length \(m\) and \(y\) of length \(n\), find a common (non-contiguous) subsequence that is as long as possible.

\[
x = \text{ABCBDAB} \\
y = \text{BDCABA}
\]

LCS = BCBA or BCAB

\[
x' = \text{AB-C-BDAB} \\
y' = \text{-BDCAB-A-}
\]

What is the complexity of the naive algorithm? How to make this efficient?
$LCS(i, j)$ means length of LCS considering only up to $x_i$ and $y_j$
Recursive Approach

$LCS(i, j)$ means length of LCS considering only up to $x_i$ and $y_j$

$$LCS(i, j) = \begin{cases} 
0 & \text{if } i \text{ or } j = 0 \\
LCS(i - 1, j - 1) + 1 & \text{if } x_i = y_j \\
\max(LCS(i - 1, j), LCS(i, j - 1)) & \text{otherwise}
\end{cases}$$
global optimum uses optimal solutions of subproblems

Proof by contradiction: What if subsolution were not optimal?

Let \( z \) be an \( LCS(i, j) \) of length \( k \).

1. If \( x_i = y_j \), then \( z_k = x_i = y_j \) and \n\[
LCS(i - 1, j - 1) = z_0..z_{k-1}.
\]
Not including \( z_k \) makes LCS suboptimal: contradiction!

If \( z_0..z_{k-1} \) were not LCS, \( z \) could be longer, hence not optimal: contradiction!

2. If \( x_i \neq y_j \) and \( z_k \neq x_i \), then \( z \) is \( LCS(i - 1, j) \).
If longer exists, \( z \) would not be an LCS: contradiction!

3. If \( x_i \neq y_j \) and \( z_k \neq y_j \), then \( z \) is \( LCS(i, j - 1) \)
Similar to 2.
Summary of Dynamic Programming

1. **optimal substructure**: global optimum uses optimal solutions of subproblems
2. **ordering over subproblems**: solve ‘smallest’ first, build ‘larger’ from them
3. ‘overlapping’ subproblems: polynomial number of subproblems, each possibly used multiple times
4. independent subproblems: optimal solution of one subproblem doesn’t affect optimality of another

- top-down: memoization
- bottom-up: compute table, then recover solution
For example:

- What’s still confusing?
- What question didn’t you get to ask today?
- What would you like to hear more about?

Please write down your most pressing question about algorithms and put it in the box on your way out.

*Thanks!*