Overview

⇒ Rod Cutting (Chapter 15.1)

Optimal Rod Cutting

• Cut rod into smaller rods to give best possible price

<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>price $p_i$</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>

• Different ways to split a pipe of length 4

• If we cut a rod of length $n$ into $k$ pieces each of $i_1, i_2, \ldots, i_k$
  - $n = i_1 + i_2 + \ldots + i_k$
  - Revenue = $p_{i_1} + p_{i_2} + \ldots + p_{i_k}$
How to Find Maximum Revenue

- Want to find the cuts that result in the most revenue
  - Let $r_n$ be the maximum revenue of a pipe of length $n$
- Cannot do this with divide and conquer
  - Do not know what an optimal first cut is
- Brute force
  - Pipe of length $n$ has $n - 1$ possible points where it can be cut
    + Price out each of the $2^{n-1}$ different possible cuts

Optimal Substructure

- Alternatively, set up a recursive definition for max revenue
  - $r_n = \max(p_n, r_{n-1} + r_{n-2}, r_{n-3}, \ldots, r_{n-1} + r_1)$
  - To solve a bigger problem, solve smaller problems of same type, but of smaller sizes
  - Overall solution incorporates optimal solutions to the two related subproblems
  - Has optimal substructure: optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently
Another Version

- Another version:
  \[ r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1) \]
  \[ = \max(p_n, p_1 + r_{n-1}, p_2 + r_{n-2}, \ldots, p_{n-1} + r_1) \]
  \[ = \max_{i \leq j \leq n}(p_i + r_{n-i}) \]
  - There will be a first cut to the rod
    + So do that first (rather than cutting in the middle of the rod)
    + Now just has one related subproblem
    + Just one recursion (can do through iteration)
  - Still see the optimal substructure

Code and Running Time

**Cut-Rod(p, n)**

1. \textbf{if} \( n == 0 \)
2. \textbf{return} 0
3. \( q = -\infty \)
4. \textbf{for} \( i = 1 \) \textbf{to} \( n \)
5. \( q = \max(q, p[i] + \text{Cut-Rod}(p, n-i)) \)
6. \textbf{return} \( q \)

- \( T(n) \) be the total number of calls to \text{Cut-Rod}(p, n)\)
  - \( T(0) = 1 \)
    + Include the initial call \text{Cut-Rod}(p, 0), which just returns 0
  - \( T(n) = 1 + \sum_{i=0}^{n-1} T(j) \)
    + Initial call + calling \text{Cut-Rod} on 0 to \( n-1 \)
  - \( T(n) = 2^n \)
Dynamic Programming

• Naive solution keeps recomputing subproblems it has already seen
• Instead, remember results for subproblems
  - Thus dynamic programming might use more memory
    + Time-memory trade-off
  - But might transform exponential algorithm to polynomial
  - Dynamic programming runs in polynomial time if
    + at most polynomial number of distinct subproblems
    + Each takes at most polynomial time
• Can do dynamic programming top-down or bottom-up

Top-down with memoization

• Write procedure recursively
  - but modified to save the result of each subproblem
    + Usually in an array or hash-table
  - First check if already solved the subproblem

\textbf{\texttt{CUT-ROD}(p, n)}

\begin{verbatim}
1  if n == 0
2     return 0
3  q = -\infty
4  for i = 1 to n
5     q = \max(q, p[i] + \texttt{CUT-ROD}(p, n - i))
6  return q
\end{verbatim}
Bottom-up method

- Depends on some natural notion of ‘size’ of a subproblem
  - such that subproblems depend only on ‘smaller’ subproblems
- Sort problems by size and solve them smallest first
  - Use saved solutions for its subproblem
  - Save solution when done

- Running time?

```plaintext
BOTTOM-UP-CUT-ROD(p, n)
1 let r[0...n] be a new array
2 r[0] = 0
3 for j = 1 to n
4 q = -∞
5 for i = 1 to j
6 q = max(q, p[i] + r[j - i])
7 r[j] = q
8 return r[n]
```