Optimal Rod Cutting

If we cut a rod of length $n$ into $k$ pieces each of length $i_1, i_2, \ldots, i_k$ then

$$\text{Revenue} = p_{i_1} + p_{i_2} + \cdots + p_{i_k} = n \cdot \text{price}$$

Different ways to split a pipe of length 4

<table>
<thead>
<tr>
<th>Piece price</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Cut rod into smaller rods to give best possible price

Overview

Optimal Rod Cutting (Chapter 15.1)
Optimal Substructure

Alternatively, set up a recursive definition for max revenue:

\[
r(n) = \max (p_n, r_1 + r(n-2), r_2 + r(n-3), \ldots, r_{n-1} + r_1)
\]

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.

Alternative: set up a recursive definition for max revenue.

---

How to Find Maximum Revenue

- Optimal Substructure: optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.
- Overall solution incorporates optimal solutions to the two related subproblems.

Alternatively, set up a recursive definition for max revenue:

\[
r(n) = \max (p_n, r_1 + r(n-2), r_2 + r(n-3), \ldots, r_{n-1} + r_1)
\]

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.
- Optimal Substructure: optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.
\begin{itemize}
\item \[ u, = (u), L \]
\item Initial call \[ (f), L \{ \frac{0}{-u} \} + 1 = (u), L \]
\item Include the initial call \texttt{Cut-Rod}(p,0), which just returns 0 \[ I = (0), L \]
\end{itemize}

Let \( T(n) \) be the total number of calls to \texttt{Cut-Rod}(p,n) -
\[ T(0) = 1 \]
Include the initial call \[ + \sum_{i=0}^{n-1} T(j) \]
Initial call + calling \texttt{Cut-Rod} on 0 to \( n-1 \)
\[ T(n) = 2^n \]

Another Version

\begin{itemize}
\item Another version:
\[ r_n = \max(\rho_{n}, r_{1+n} + [1] d, b) \]
\[ \max(\rho_{n}, \rho_{1+n} + [1] d, b) = b \]
\item For \[ i = 1 \]
\[ \infty = b \]
\item Return 0 \[ i \]
\[ 0 = u \]
\[ \texttt{Cut-Rod}(u, d) \]
\end{itemize}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Code and Running Time &  \\
\hline
\end{tabular}
\end{table}
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

```c
((t - u \cdot d) \cdot b) = b
\ni = 1
\infty = b
\ni = 0
\iif (u \cdot d) = 0
\ielse (u \cdot d)(cut-rod(p; n))
```

Dynamic Programming

- Naive solution keeps recomputing subproblems it has already seen
- This dynamic programming might use more memory
- Time-memory trade-off
- But dynamic programming exponential algorithm to polynomial
- Dynamic programming runs in polynomial time if
  - at most polynomial number of distinct subproblems
  - Each subproblem at most polynomial time
  - Each subproblem at most polynomial time

Can do dynamic programming top-down or bottom-up

- Dynamic programming runs in polynomial time if
  - at most polynomial number of distinct subproblems
  - Each subproblem at most polynomial time

Dynamic Programming
Reconstructing a Solution

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

- Use saved solutions when done

Running time?

```plaintext
return [u].[t]

b = [f].t

([t - f] + [f]d) max = [b]

for i = 1 to n

q = /NUL1

for i = 0 to j

q = max [q; p[i]]

/NUl1 /NUL1

r[j] = q

return r[n]
```

Sort problems by size and solve them smallest first.

- Use saved solutions for its subproblem

- Use saved solutions when done

- Save solution when done

- Depends on some natural notion of 'size' of a subproblem

- Running time?