Dynamic Programming Recipe

- **Step 1:** Devise simple recursive algorithm
  - Flavor: make “first choice”, then recursively solve remaining part of the problem
- **Step 2:** Write recurrence for optimal value
- **Step 3:** Design bottom-up iterative algorithm
  - Weighted interval scheduling: first-choice is binary
  - Rod-cutting: first choice has \( n \) options
  - Subset Sum: need to “add a variable”

### Rod Cutting

- **Formulate problem on board**

- **Problem Input:**
  - Steel rod of length \( n \), can be cut into integer lengths
  - Price \( p(i) \) for a rod of length \( i \)

- **Goal**
  - Cut rods into lengths \( i_1, \ldots, i_k \) such that \( i_1 + i_2 + \ldots i_k = n \).
  - Maximize value \( p(i_1) + p(i_2) + \ldots + p(i_n) \)

### First choice?

- **Choose length \( i \) of first piece, then recurse on smaller rod**

### Steps 1 and 2

**Step 1:** Recursive Algorithm.

\[
\text{CutRod}(j) \\
\quad \text{if } j = 0 \text{ then return } 0 \\
\quad v = 0 \\
\quad \text{for } i = 1 \text{ to } j \text{ do} \\
\quad \quad v = \max (v, p[i] + \text{CutRod}(j - i)) \\
\quad \text{end for} \\
\quad \text{return } v \\
\]

- **Running time for CutRod(\( n \))? \( \Theta(2^n) \)**

**Step 2:** Recurrence

\[
\text{OPT}(j) = \max_{1 \leq i \leq j} \{p_i + \text{OPT}(j - i)\} \\
\text{OPT}(0) = 0
\]

### Step 3: Iterative Algorithm

- **Array \( M[0..n] \) where \( M[i] \) holds value of OPT(\( i \)). Order to fill \( M \)? From 0 to \( n \).**

\[
\text{CutRod-Iterative} \\
\text{Initialize array } M[0..n] \\
\text{Set } M[0] = 0 \\
\text{for } j = 1 \text{ to } n \text{ do} \\
\quad \text{for } i = 1 \text{ to } j \text{ do} \\
\quad \quad v = 0 \\
\quad \quad \text{for } i = 1 \text{ to } j \text{ do} \\
\quad \quad \quad v = \max (v, p[i] + M[j - i]) \\
\quad \quad \text{end for} \\
\quad \text{Set } M[j] = v \\
\text{end for}
\]

- **Running time? \( \Theta(n^2) \)**
  - Note: body of for loop identical to recursive algorithm, directly implements recurrence
Epilogue: Recover Optimal Solution

Run previous algorithm to fill in $M$ array

cuts = {}

while $j > 0$
do
  $i^* = \text{null}$, $v = 0$. $i^*$ is the selected cut, $v$ is its value
  for $i = 1$ to $j$
do
    if $p[i] + M[j - i] > v$ then
      $i^* = i$
      $v = p[i] + M[i]$
    end if
  end for
  $j = j - i^*$
end while

Problem Formulation

▶ Example on board

▶ Input
  ▶ Items 1, 2, . . . , n
  ▶ Weights $w_i$ for all items (integers)
  ▶ Capacity $W$

▶ Goal: select a subset $S$ whose total weight is as large as possible without exceeding $W$.

▶ Subset Sum: need to “add a variable” to recurrence

Step 1: Recursive Algorithm, First Try

▶ Let $O$ be optimal solution on items $\{1, 2, \ldots, j\}$. Is $j \in O$ or not?

▶ SubsetSum($j$)
  if $j = 0$ then return 0
  ▶ Case 1: $j \notin O$
    val1 = ???
  ▶ Case 2: $j \in O$
    val2 = 0
    if $w_j \leq W$ then
      val2 = ???
    end if
  end if
  return max(val1, val2)


Step 1: Recursive Algorithm, Add a Variable

▶ Find value of optimal solution $O$ on items $\{1, 2, \ldots, j\}$ when the remaining capacity is $w$

▶ SubsetSum($j, w$)
  if $j = 0$ then return 0
  ▶ Case 1: $j \notin O$
    val1 = SubsetSum($j - 1, w$)
  ▶ Case 2: $j \in O$
    val2 = 0
    if $w_j \leq w$ then
      val2 = $w_j + \text{SubsetSum}(j - 1, w - w_j)$
    end if
  end if
  return max(val1, val2)

Recurrence

▶ Let $OPT(j, w)$ be the maximum weight of any subset of items $\{1, \ldots, j\}$ that does not exceed $w$

\[
OPT(j, w) = \max \left\{ \begin{array}{c} OPT(j - 1, w), \\
  w_j + OPT(j - 1, w - w_j) \end{array} \right\}
\]

▶ Unless $w_j > w$, then $OPT(j, w) = OPT(j - 1, w)$

▶ Base case: $OPT(0, w) = 0$ for all $w = 0, 1, \ldots, W$.

Questions

▶ Do we need a base case for $OPT(j, 0)$?
▶ What is overall optimum to original problem? $OPT(n, W)$
Step 3: Iterative Algorithm

- SubsetSum($n, W$)
  - Initialize array $M[0..n, 0..W]$
  - Set $M[0, w] = 0$ for $w = 0, \ldots, W$
  - for $j = 1$ to $n$
    - for $w = 1$ to $W$
      - Use recurrence from previous slide to compute $M[j, w]$
    - end for
  - end for
  - return $M[n, W]$

- Example on board.

- Running Time? $\Theta(nW)$. Note: this is “pseudopolynomial”. Not strictly polynomial, because it can be exponential in the number of bits used to represent the values.