Dynamic Programming

Hyrum D. Carroll
(Based on slides prepared by Suk Jin Lee)

Dynamic programming

- It is used, when the solution can be ***recursively*** described in terms of solutions to subproblems (*optimal substructure*)
- Algorithm solves each subproblem just once and ***stores its answer in memory*** (a table) for later use
- More efficient than "*brute-force methods*", which solve the same subproblems over and over again
- Call such a solution ***an* optimal solution** to the problem, as opposed to *the* optimal solution

Dynamic programming

- Follow a sequence of four steps:
  1. Characterize the structure of an optimal solution.
  2. Recursively define the value of an optimal solution.
  3. Compute the value of an optimal solution, typically in a bottom-up fashion.
  4. Construct an optimal solution from computed information.

Rod-cutting Problem

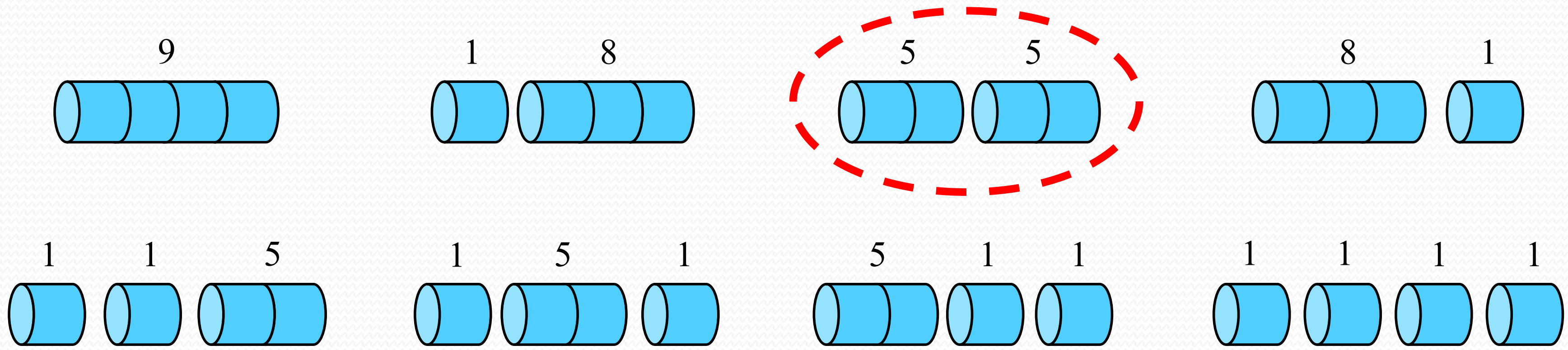
- How to cut steel rods into pieces in order to maximize the revenue you can get?
  - Each cut is free
  - Rod lengths are always an integral number of inches
- Definition
  - **Input**: A rod of length n inches and a table of prices p_i, for $i = 1, 2, \ldots, n$
  - **Output**: determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces

Rod-cutting Problem

- Example
  - A table of pieces p_i

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

  - When n = 4, cut the length n in 2^{n-1} different ways



Rod-cutting Problem

- When $n = 7$, a rod of length 7 is cut into three pieces
  7 = 2 + 2 + 3 – two of length 2 and one of length 3
- If an optimal solution cuts the rod into k pieces for $1 \leq k \leq n$, then an optimal decomposition

$$n = i_1 + i_2 + \dots + i_k$$

- Provides maximum corresponding revenue

$$r_n = p_{i_1} + p_{i_2} + \ldots + p_{i_k}$$

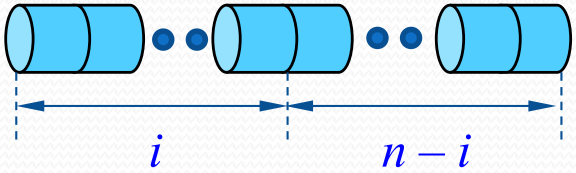
Rod-cutting Problem

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

Maximum revenue:
$r_1 = 1$ from solution 1 = 1 (no cuts),
$r_2 = 5$ from solution 2 = 2 (no cuts),
$r_3 = 8$ from solution 3 = 3 (no cuts),
$r_4 = 10$ from solution 4 = 2 + 2,
$r_5 = 13$ from solution 5 = 2 + 3,
$r_6 = 17$ from solution 6 = 6 (no cuts),
$r_7 = 18$ from solution 7 = 1 + 6 or 7 = 2 + 2 + 3,
$r_8 = 22$ from solution 8 = 2 + 6,
$r_9 = 25$ from solution 9 = 3 + 6,
$r_{10} = 30$ from solution 10 = 10 (no cuts),

- First piece of length i and then a remainder of length $n - i$

$$r_n = \max_{1 \le i \le n}\left(p_i + r_{n-i}\right)$$



Recursive top-down implementation

Input: an array $p[1\,..\,n]$ and an integer n

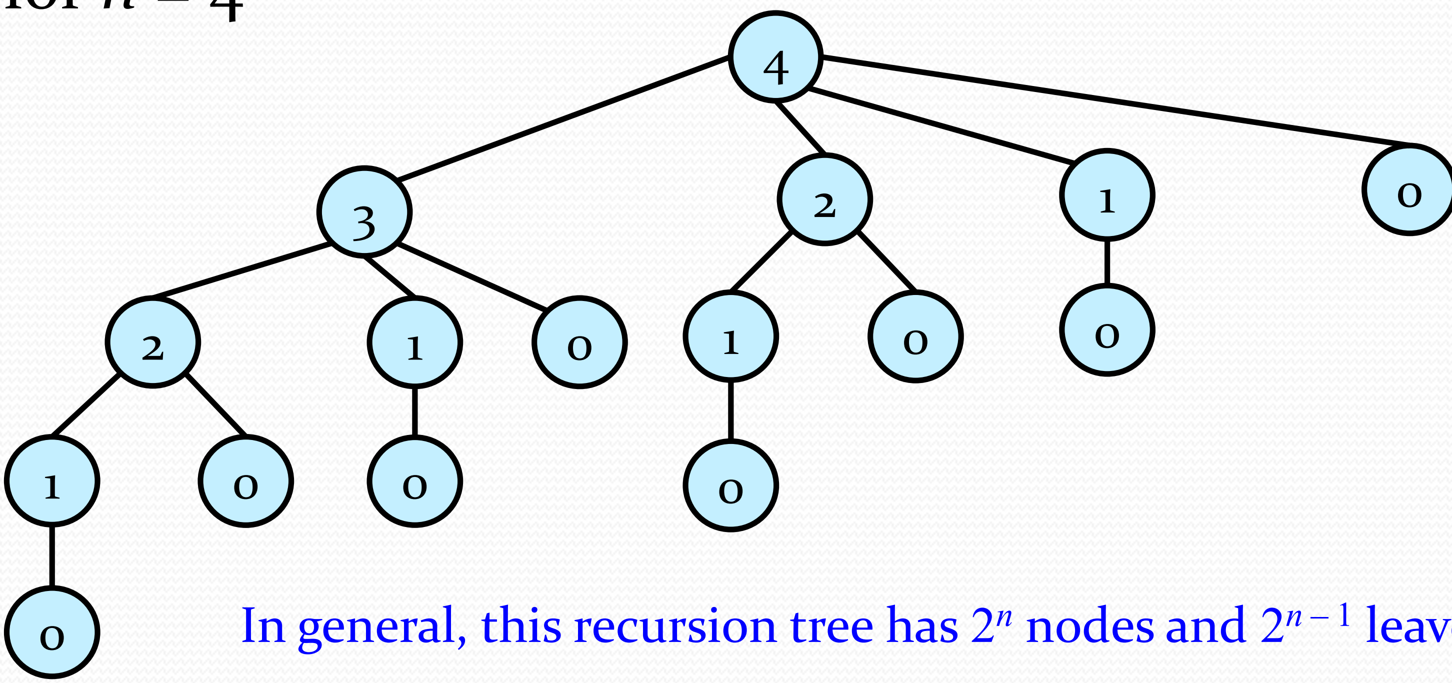
CUT-ROD(p, n) // p: prices

1. **if** $n == 0$
2. **return** 0
3. $q = -\infty$ // Initialize the maximum revenue q to $-\infty$
4. **for** $i = 1$ **to** n
5. $q = \max(q, p[i] +$ **CUT-ROD**$(p, n - i))$ // $r_n = \max_{1 \le i \le n}(p_i + r_{n-i})$
6. **return** q

Each time you increase n by 1, your program's running time would approximately double.

Recursive top-down implementation

- Recursion tree
  - Recursive calls resulting from a call CUT-ROD(p, r) for $n = 4$



In general, this recursion tree has 2^n nodes and 2^{n-1} leaves.

Recursive top-down implementation

- Recursion tree
  - Lots of repeated subproblems
  - Solve the subproblem for size 2 twice, for size 1 four times, and for size 0 eight times
  - Exponential growth

$$T(n)=\begin{cases} 1 & if \quad n=0 \\ 1+\sum_{j=0}^{n-1} T(j) & if \quad n\geq 1 \end{cases}$$

- Solution to recurrence: $T(n) = 2^n$

Dynamic-programming solution

Dynamic-programming solution

- Instead of solving the same subproblems repeatedly, arrange to solve each subproblem just once.
- Save the solution to a subproblem in a table, and refer back to the table whenever we revisit the subproblem.
- "Store, don't recompute" $\Rightarrow$ time-memory trade-off
- Can turn an exponential-time solution into a polynomial-time solution.
- Two basic approaches: top-down with memoization[1], and bottom-up

[1] This is not a misspelling. *Memoization* comes from *memo*, since the technique consists of recording a value so that we can look it up later

Using dynamic programming for optimal rod cutting

- Top-down approach with memoization
  - To find the solution to a subproblem, first look in the table.
  - If the answer is there, use it.
  - Otherwise, compute the solution to the subproblem and then store the solution in the table for future use
  - **Memoized** $\Rightarrow$ it "remembers" what results it has computed previously

Using dynamic programming for optimal rod cutting

- Top-down approach with memoization

MEMOIZED-CUT-ROD(p, n)

Let $r[0 \,..\, n]$ be a new array
for $i = 0$ **to** n
    $r[i] = -\infty$ // Initializes a new array $r[0 \,..\, n]$ with $-\infty$ (unknown)
return MEMOIZED-CUT-ROD-AUX(p, n, r)

Using dynamic programming for optimal rod cutting

- Top-down approach with memoization

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0          // check to see whether the desired value is already known
        return r[n]       // if the desired value is known
if n == 0            // compute the desired value q in the usual manner if it is unknown
        q = 0
else q = −∞          // the solution is unknown
        for i = 1 to n
                q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n−i, r))
r[n] = q             // Save the computed value q in r[n]
return q
```

The procedure MEMOIZED-CUT-ROD-AUX is just the memoized version of the previous procedure, CUT-ROD.

Using dynamic programming for optimal rod cutting

- Bottom-up approach

```
BOTTOM-UP-CUT-ROD(p, n)
Let r[0 . . n] be a new array        // create a new array to save the results
r[0] = 0          // a rod of length 0 earns no revenue
for j = 1 to n
        q = −∞
        for i = 1 to j        // i < j
                q = max(q, p[i] + r[j – i])
        r[j]  = q  // Save the solution to the subproblem of size j in r[n]
return r[n]
```

Solve each subproblem of size j, for $j = 1, 2, \ldots, n$, in order of increasing size

Directly references array entry $r[j - i]$ instead of making a recursive call to solve the subproblem $j - i$

Using dynamic programming for optimal rod cutting

- Running time
  - BOTTOM-UP-CUT-ROD
    - Doubly-nested loop structure
    - Number of iterations of inner **for** loop forms an arithmetic series
  - MEMOIZED-CUT-ROD
    - **for** loop of lines 6 – 7 iterates n times
    - Total number of iterations forms an arithmetic series

Using dynamic programming for optimal rod cutting

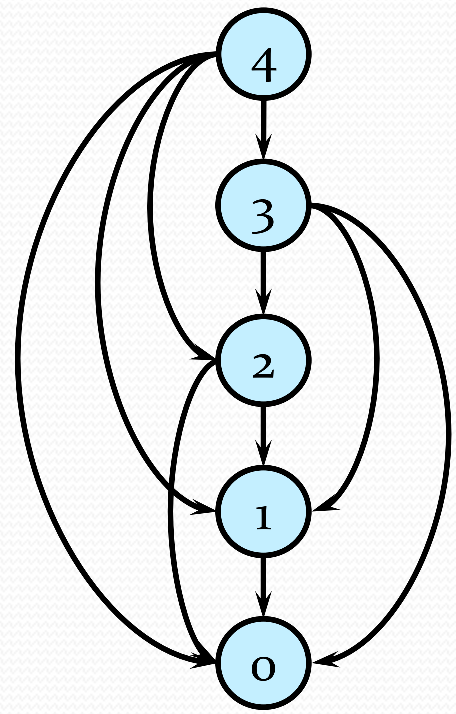
- Running time
  - BOTTOM-UP-CUT-ROD
    - Doubly-nested loop structure
    - Number of iterations of inner **for** loop forms an arithmetic series

    $\Rightarrow \Theta(n^2)$
  - MEMOIZED-CUT-ROD
    - **for** loop of lines 6 – 7 iterates n times
    - Total number of iterations forms an arithmetic series

    $\Rightarrow \Theta(n^2)$

Subproblem graphs

- How to understand the subproblems involved and how they depend on each other.
- Directed graph:
  - Vertex labels: sizes of the corresponding subproblems
  - Directed edge (x, y): need a solution to subproblem y when solving subproblem x
- ***Example***: For the rod-cutting problem with $n = 4$



Reconstructing a solution

- So far, have focused on computing the *value* of an optimal solution, rather than the *choices* that produced an optimal solution
- Extend the bottom-up approach to record not just optimal values, but *optimal choices*
  - Save the optimal choices in a separate table (s[ ])
  - Then use a separate procedure to print the optimal choices

Reconstructing a solution

- Extended version of BOTTOM-UP-CUT-ROD

EXTENDED-BOTTOM-UP-CUT-ROD(p, n)

1. Let $r[0..n]$ and $s[1..n]$ be new arrays // s_j: optimal size of the first piece to cut
2. $r[0] = 0$ // a rod of length 0 earns no revenue
3. **for** $j = 1$ **to** n
4. $\quad q = -\infty$
5. $\quad$ **for** $i = 1$ to j // $i < j$
6. $\quad\quad$ **if** $q < p[i] + r[j - i]$
7. $\quad\quad\quad q = p[i] + r[j - i]$
8. $\quad\quad\quad s[j] = i$ // hold the optimal size i of the first piece to cut off
9. $\quad r[j] = q$ // Save the solution to the subproblem of size j in $r[n]$
10. **return** r and s

Reconstructing a solution

- To print out the cuts made in an optimal solution:

PRINT-CUT-ROD-SOLUTION(p, n)
1. (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2. **while** $n > 0$
3. print $s[n]$
4. $n = n - s[n]$

Reconstructing a solution

- Example: EXTENDED-BOTTOM-UP-CUT-ROD(p, 10) returns

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

Length i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0										
$s[i]$											

$r[0] = 0$

Reconstructing a solution

- Example: EXTENDED-BOTTOM-UP-CUT-ROD(p, 10) returns

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

Length i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	**1**									
$s[i]$		**1**									

if $q < p[i] + r[j-i] = p[1] + r[0] = 1 + 0 = 1$ // $j = 1$, $i = 1$

$q = p[1] + r[0] = 1$

$s[1] = 1$ // $s[j] = i$

Reconstructing a solution

- Example: EXTENDED-BOTTOM-UP-CUT-ROD(p, 10) returns

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

Length i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	**5**								
$s[i]$		1	**2**								

if $q < p[i] + r[j-i] = p[2] + r[0] = 5 + 0 = 5$ // $j = 2$, $i = 2$

$q = p[2] + r[0] = 5$

$s[2] = 2$ // $s[j] = i$

Reconstructing a solution

- Example: EXTENDED-BOTTOM-UP-CUT-ROD(p, 10) returns

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

Length i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	**8**							
$s[i]$		1	2	**3**							

if $q < p[i] + r[j - i] = p[3] + r[0] = 8 + 0 = 8$ // $j = 3$, $i = 3$

$q = p[3] + r[0] = 8$

$s[3] = 3$ // $s[j] = i$

Reconstructing a solution

- Example: EXTENDED-BOTTOM-UP-CUT-ROD(p, 10) returns

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

Length i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	**10**						
$s[i]$		1	2	3	**2**						

if $q < p[i] + r[j-i] = p[2] + r[2] = 5 + 5 = 10$ // $j = 4$, $i = 2$

$q = p[2] + r[2] = 10$

$s[4] = 2$ // $s[j] = i$

Reconstructing a solution

- Example: EXTENDED-BOTTOM-UP-CUT-ROD(p, 10) returns

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

Length i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	**13**					
$s[i]$		1	2	3	2	**2**					

if $q < p[i] + r[j - i] = p[2] + r[3] = 5 + 8 = 13$ // $j = 5, i = 2$

$q = p[2] + r[3] = 13$

$s[5] = 2$ // $s[j] = i$

Reconstructing a solution

- Example: EXTENDED-BOTTOM-UP-CUT-ROD(p, 10) returns

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

Length i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	**17**				
$s[i]$		1	2	3	2	2	**6**				

if $q < p[i] + r[j-i] = p[6] + r[0] = 17 + 0 = 17$ // $j = 6, i = 6$

$q = p[6] + r[0] = 17$

$s[6] = 6$ // $s[j] = i$

Reconstructing a solution

- Example: EXTENDED-BOTTOM-UP-CUT-ROD(p, 10) returns

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

Length i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	**18**			
$s[i]$		1	2	3	2	2	6	**1**			

if $q < p[i] + r[j - i] = p[1] + r[6] = 1 + 17 = 18$ // $j = 7$, $i = 1$

$q = p[1] + r[6] = 18$

$s[7] = 1$ // $s[j] = i$

Reconstructing a solution

- Example: EXTENDED-BOTTOM-UP-CUT-ROD(p, 10) returns

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

Length i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	**22**		
$s[i]$		1	2	3	2	2	6	1	**2**		

if $q < p[i] + r[j - i] = p[2] + r[6] = 5 + 17 = 22$ // $j = 8$, $i = 2$

$q = p[2] + r[6] = 22$

$s[8] = 2$ // $s[j] = i$

Reconstructing a solution

- Example: EXTENDED-BOTTOM-UP-CUT-ROD(p, 10) returns

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

Length i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	**25**	
$s[i]$		1	2	3	2	2	6	1	2	**3**	

if $q < p[i] + r[j-i] = p[3] + r[6] = 8 + 17 = 25$ // $j = 9$, $i = 3$

$q = p[6] + r[3] = 25$

$s[9] = 3$ // $s[j] = i$

Reconstructing a solution

- Example: EXTENDED-BOTTOM-UP-CUT-ROD(p, 10) returns

Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

Length i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	**30**
$s[i]$		1	2	3	2	2	6	1	2	3	**10**

if $q < p[i] + r[j - i] = p[10] + r[0] = 30 + 0 = 30$ // $j = 10$, $i = 10$

$q = p[10] + r[0] = 30$

$s[10] = 10$ // $s[j] = i$