Dynamic Programming

Rod Cutting

Dynamic Programming

- An algorithm design approach.
- Not a programming technique.
- Identifies smaller sub-problems.
- Computes solution of larger sub-problems using solutions of smaller sub-problems.
- After solving all sub problems, it computes the final solution. Final solution is just the answer to the biggest sub problem.

DP Vs. Divide & Conquer

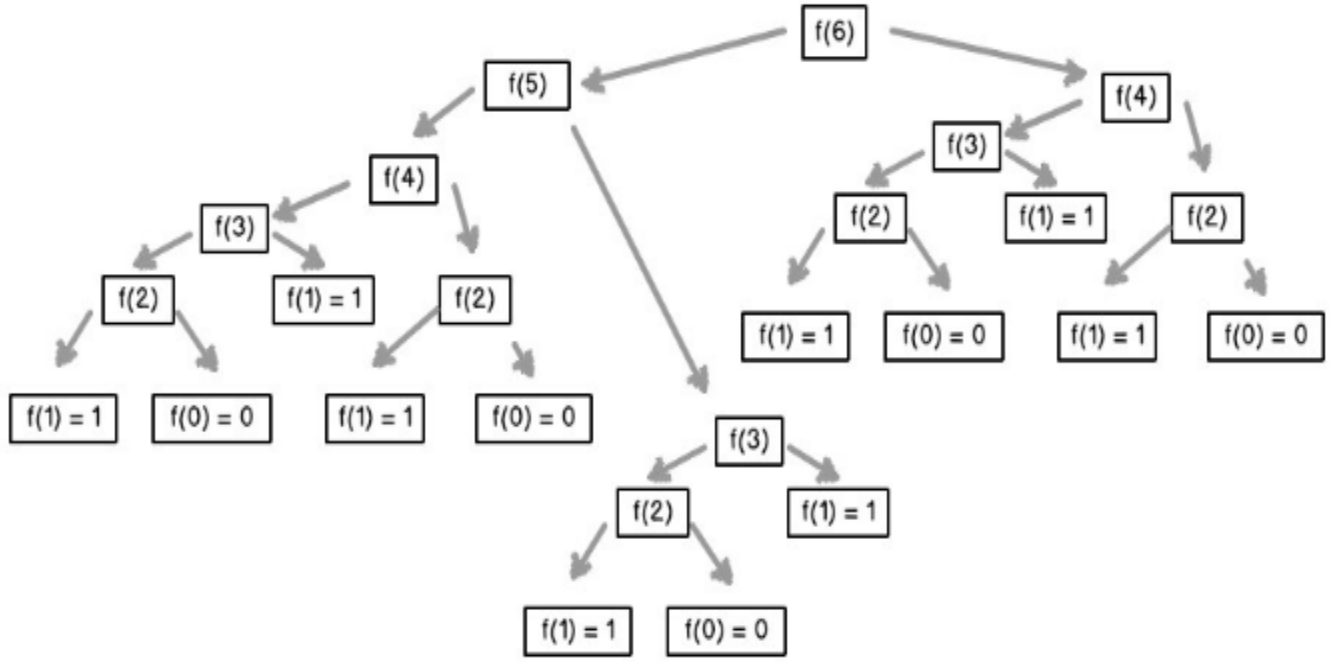
- Both partition a problem into smaller subproblems and build solution of larger problems from solutions of smaller problems.
- In D&C, work top-down. Know exact smaller problems that need to be solved to solve larger problem.
- In DP, (usually) work bottom-up. Solve all smaller size problems and build larger problem solutions from them.
  - In DP, many large subproblems reuse solution to same smaller problem.
- DP is used for optimization problems.
- Problems have many solutions; we want the best one

Overlapping Sub-problems

- D&C is used when sub-problems are disjoint. (merge sort, quick sort).
- DP is used when the sub-problems are overlapping.
- Recursion calculates the solutions of all sub-problems. It computes solutions of overlapping sub-problems again and again and hence is expensive.
- DP calculates solutions of overlapping sub-problems only once and saves them.
- DP uses pre-computed solutions of overlapping sub-problems instead of computing them again.

Overlapping Sub-problems

This figure shows the recursion tree of the recursion to solve Fibonacci numbers. This tree is for finding 6^{th} Fibonacci number. You can see that entire sub-trees are being repeated.



Memoization

- DP saves the solutions of sub-problems to use them later on. This is called memoization

Optimal Sub-structure

- DP is used to solve problems which exhibit optimal sub-structure.
- Optimal sub-structure property states that the optimal solution of a problem contains in it the optimal solutions of sub-problems.
- In other words, if the optimal solution of a problem can be constructed by combining optimal solutions of its sub-problems, then that problem has an optimal sub-structure

Steps of DP

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

- Some company buys long steel rods and cuts them into shorter rods, which it then sells. Each cut is free. The management of company wants to know the best way to cut up the rods.

- We assume that we know the price (p_i) in dollars (for i = 1,2,...n), that the company charges for a rod of length i inches. Rod lengths are always an integral number of inches.

Rod-Cutting Problem

- **Input:** We are given a rod of length n and a table of prices p_i for i = 1…… n; p_i is the price of a rod of length i.

- **Goal:** to determine the maximum revenue r_n, obtainable by cutting up the rod and selling the pieces.

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

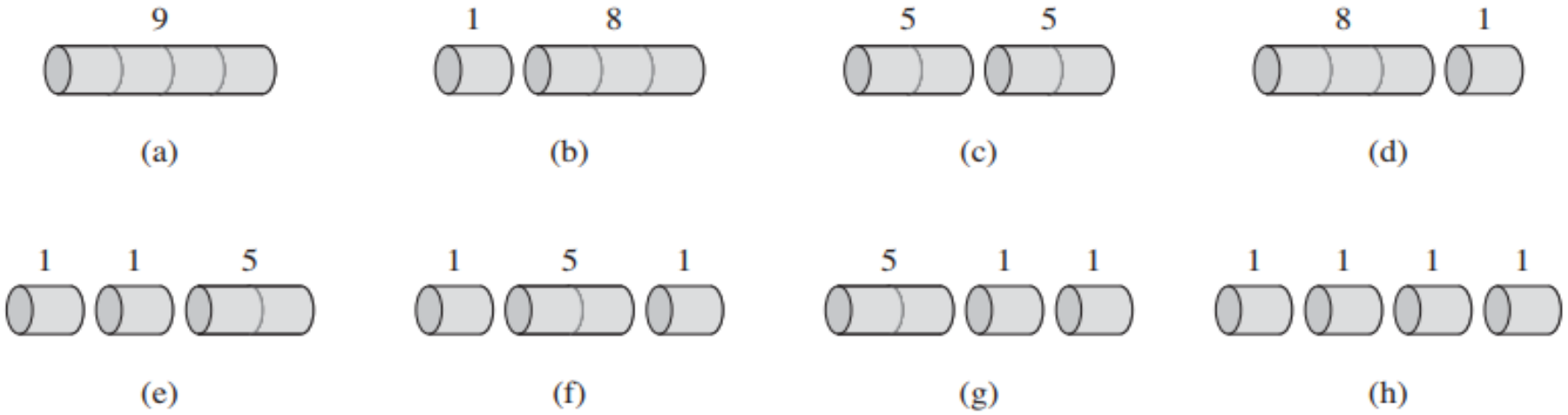


Figure 15.2 The 8 possible ways of cutting up a rod of length 4. Above each piece is the value of that piece, according to the sample price chart of Figure 15.1. The optimal strategy is part (c)—cutting the rod into two pieces of length 2—which has total value 10.

Recursive Solution

- Suppose r_n denotes optimal solution for rod of length n and p_i denotes price of rod of length i.

- We denote a decomposition into pieces using ordinary additive notation, so that 7 = 2 + 2 + 3 indicates that a rod of length 7 is cut into three pieces—two of length 2 and one of length 3.

Recursive Solution

- If an optimal solution cuts the rod into k pieces, for some 1<=k<= n, then an optimal decomposition:

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

of the rod into pieces of lengths $i_1, i_2, \ldots, i_k$ provides maximum corresponding revenue:

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k} .$$

Recursive Solution

- For a rod of length n, there are n-1 ways to make the first cut.

Recursive Solution

1. When no cut is made, revenue is r_n

2. Make the first cut and then cut the remaining rod recursively.

p_1 is the price of rod of length 1

$p_1 + r_{n-1}$

$p_2 + r_{n-2}$

$p_3 + r_{n-3}$

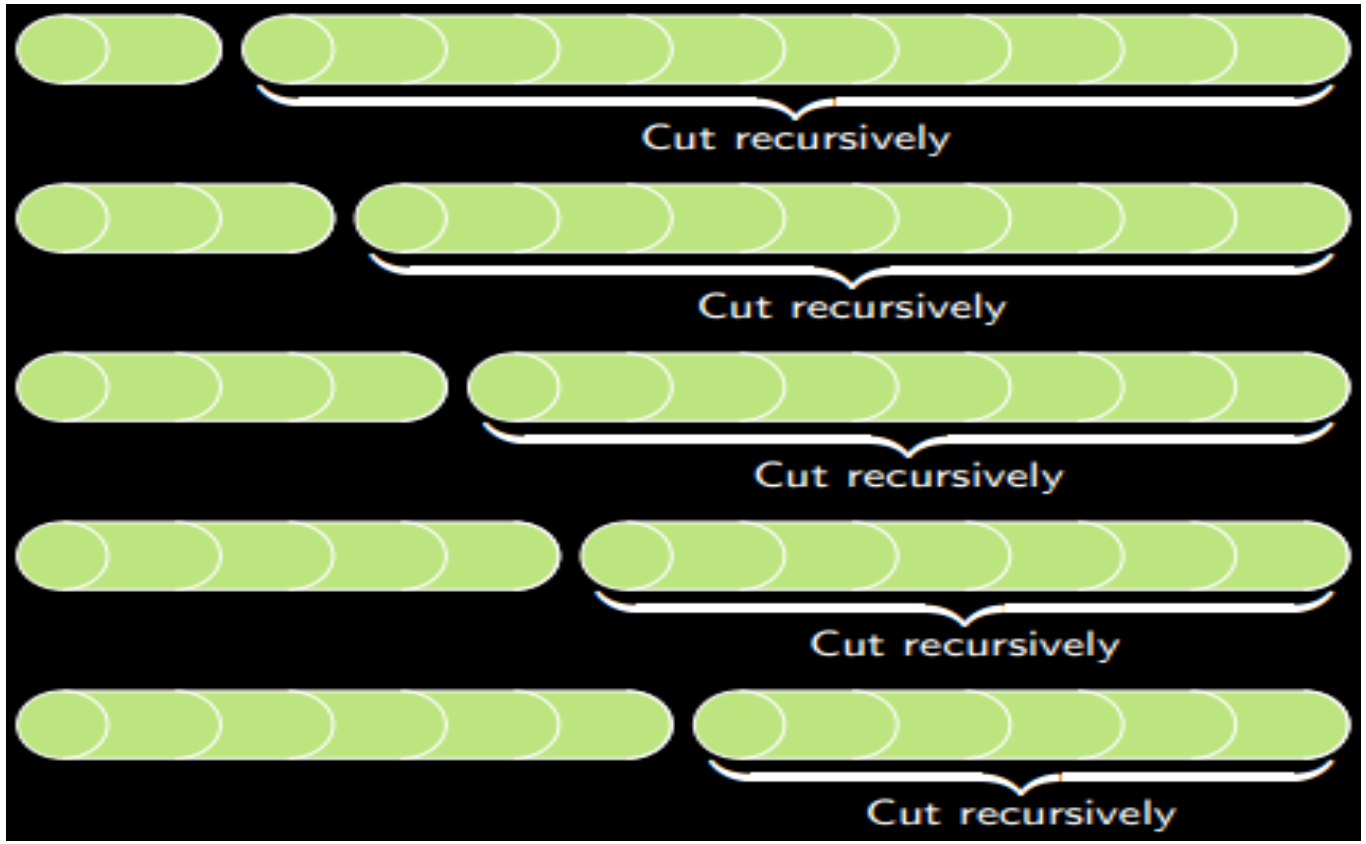
$p_4 + r_{n-4}$

.

.

.

$p_{n-1} + r_1$



Recursive Solution

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

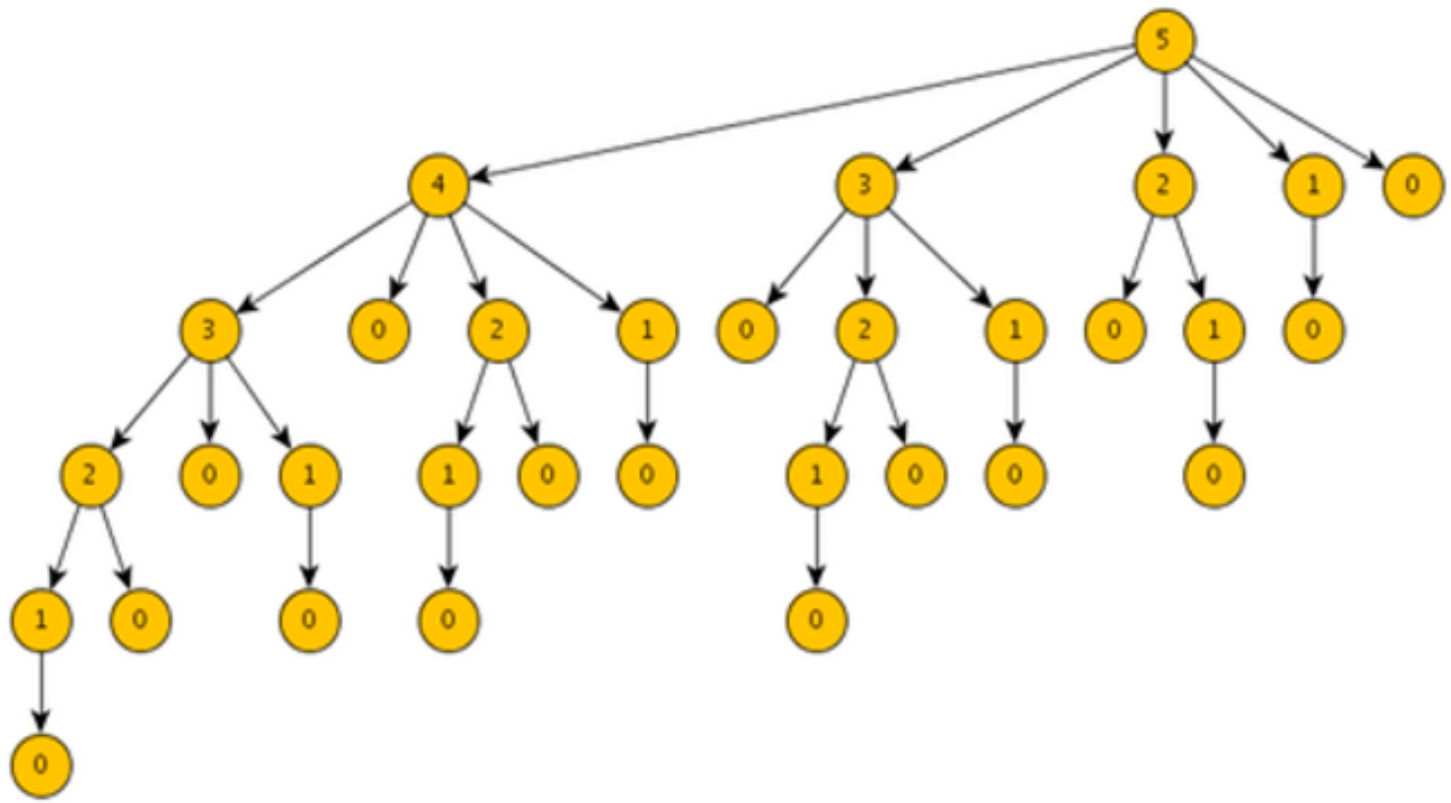
- Iterator i is deciding the position of first cut. Remaining rod is then cut recursively
- In general, rod of length n can be cut in 2^{n-1} different ways, since we can choose cutting, or not cutting, at all distances i ($1 \le i \le n - 1$) from the left end.

Recursive Solution

```
CUT-ROD(p, n)
1  if n == 0
2       return 0
3  q = -∞
4  for i = 1 to n
5       q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

To see why the procedure is inefficient, we draw the recursion tree of Cut-Rod for n= 5 in this Figure. In the tree, each vertex is a procedure calling. The number in the vertex is the parameter n.

Recursion tree for Cut-Rod(*p*, 5)



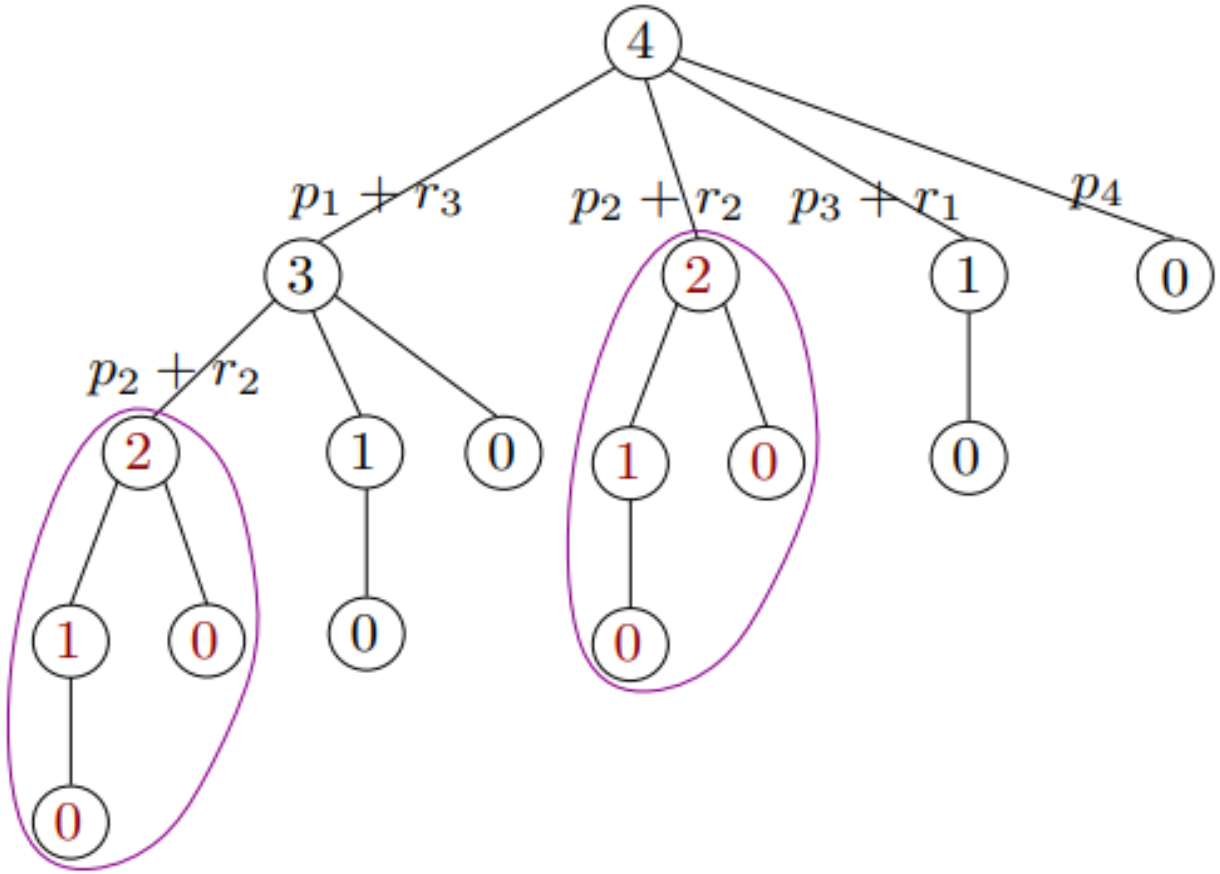
Recursion - Analysis

- Suppose T(n) is the total time. We can write following recurrence for our algorithm:

$$T(n) = \begin{cases} 1 + \sum_{0 \le j \le n-1} T(j), & \text{if } n > 0, \\ 1, & \text{if } n = 0. \end{cases}$$

- This can be expanded as follows: T(n) = 1 + {1 +2 + 4 + 8+6+32, …..+ 2^{n-1}}.
- Using geometric series (or exponential recurrence from appendix A.5 of text book), we get T(n) = 2^n

Overlapping Sub-problems



Optimal Sub-structure

The optimal solution to the original problem incorporates optimal solutions to the subproblems, which may be solved independently.

Optimal Sub-Structure

- Lets suppose we have the optimal solution for cutting the rod $C_{i\ldots j}$ where C_i is the first piece (i.e. first inch of this rod $C_{i\ldots j}$) and C_j is the last piece (i.e. last inch of this rod $C_{i\ldots j}$). (e.g. optimal revenue = 35)

- Lets suppose an optimal solution exists and cuts are specified at different inches. If we take one of the cuts from this optimal solution, lets say k, and split it so that we have two sub-problems $C_{i\ldots k}$ and $C_{k+1\ldots j}$ (assuming our optimal is not just a single piece). Suppose the revenue in our final optimal solution, for the portion $C_{i\ldots k}$ is 15 and for the portion $C_{k+1\ldots j}$ is 20 (hence total of 35).

Optimal Sub-Structure

- If optimal structure exists then this part that we have made from i to k (it is a smaller problem): the optimal solution of this sub-problem must be 15, if there had been a better solution that would create a contradiction

Optimal Sub-structure

- Lets suppose we had a more optimal way of cutting $C_{i\ldots k}$, with higher revenue 16
- We would swap the old $C_{i\ldots k}$ and replace it with the more optimal $C_{i\ldots k}$
- Overall the entire problem will now have an even more optimal solution with revenue 16+20 = 36
- But we already stated that we already had the optimal solution i.e. 35 (so, its not possible for the original rod to have revenue more than 35).
- So, this is a CONTRADICTION!
- Therefore our original optimal solution is the optimal solution and this problem exhibits optimal substructure

DP Recursive Formula

- Lets define R[i] as the price of the optimal cut of a rod up until length i. R is an array for revenues that we'll gradually fill. Each index will denote the maximum revenue for that length of rod e.g. R[5] will have maximum revenue you can get from different types of cut for rod of length 5.
- Let p_k be the price of a cut at length *k.*
- We define the smallest sub problems first, and store their solution
- Lets convert the recursive solution to DP recursive equation:
- $$r_n = \max_{1 \le i \le n} \left(p_i + r_{n-i} \right) .$$

- DP recursive formula:

$$R[i] = max_{1 \le k \le i} \{ p_k + R[i - k] \}$$

DP Solution

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

$$R[i] = max_{1 \le k \le i} \{p_k + R[i-k]\}$$

Length (i)	1	2	3	4	5	6	7	8	9
R[i]	1								

i = 2

R[2] = max { k = 1; $p_1 + R[1] = 1 + 1 = 2$

k = 2; $p_2 = 5$

DP Solution

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

$$R[i] = max_{1 \le k \le i} \{p_k + R[i-k]\}$$

Length (i)	1	2	3	4	5	6	7	8	9
R[i]	1	5							

i = 2

R[2] = max {

k = 1; $p_1 + R[1] = 1 + 1 = 2$

k = 2; $p_2 = 5$

DP Solution

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

$$R[i] = max_{1 \le k \le i} \{ p_k + R[i-k] \}$$

Length (i)	1	2	3	4	5	6	7	8	9
R[i]	1	5							

i = 3

R[3] = max { k = 1; $p_1 + R[2] = 1 + 5 = 6$
k = 2; $p_2 + R[1] = 5 + 1 = 6$
k = 3; $p_3 = 8$

DP Solution

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

$$R[i] = max_{1 \le k \le i} \{ p_k + R[i-k] \}$$

Length (i)	1	2	3	4	5	6	7	8	9
R[i]	1	5	8						

i = 3

R[3] = max {

k = 1; $p_1 + R[2] = 1 + 5 = 6$

k = 2; $p_2 + R[1] = 5 + 1 = 6$

k = 3; $p_3 = 8$

DP Solution

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

$$R[i] = max_{1 \le k \le i} \{ p_k + R[i-k] \}$$

Length (i)	1	2	3	4	5	6	7	8	9
R[i]	1	5	8						

i = 4

R[4] = max {

k = 1; $p_1 + R[3] = 1 + 8 = 9$

k = 2; $p_2 + R[2] = 5 + 5 = 10$

k = 3; $p_3 + R[1] = 8 + 1 = 9$

k = 4; $p_4 = 9$

DP Solution

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

$$R[i] = max_{1 \le k \le i} \{ p_k + R[i-k] \}$$

Length (i)	1	2	3	4	5	6	7	8	9
R[i]	1	5	8	10					

i = 4

R[3] = max {

k = 1; $p_1 + R[3] = 1 + 8 = 9$

k = 2; $p_2 + R[2] = 5 + 5 = 10$

k = 3; $p_3 + R[1] = 8 + 1 = 9$

k = 4; $p_4 = 9$

DP Algorithm

- Convert Recursive formula to DP algorithm.
- Usually you only need to put loops around the recursive formula.

DP Algorithm – Optimal Value

- Rod-Cutting-DP (P,n) // P is array of prices, n is number of items

```
{
R[0] = 0 //base case
for i = 1 to n
      max =  -infinity
      for k = 1 to i
            if (max < P[k] + R[i –k])
                  max = P[k] + R[i –k]

      R[i] = max
}
```

Cost = $O(n^2)$

HW

- Complete the dry-run of example given in previous slides.
- Update the code to find the optimal solution (Hint: when max is updated then you need to save the length which gave that max value)