

CSCI203

Algorithms and Data Structures



Dynamic Programming (I)

Lecturer: Dr. Fenghui Ren

Room 3.203

Email: fren@uow.edu.au

Dynamic Programming (DP)



- ▶ Typically applied to optimization problems
- ▶ Main idea:
 - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
 - solve smaller instances once
 - record solutions in a table
 - extract solution to the initial instance from that table

Coin-Collection Problem

- ▶ Several coins are placed in cells of an $n \times m$ board, no more than one coin per cell. A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell.
- ▶ On each step, the robot can move either one cell to the right or one cell down from its current location. When the robot visits a cell with a coin, it always picks up that coin.
- ▶ Find the maximum number of coins the robot can collect and a path it needs to follow to do this.

	1	2	3	4	5	6
1					●	
2		●		●		
3				●		●
4			●			●
5	●				●	

Coin-Collection Problem

- ▶ $F(i, j)$ - the largest number of coins the robot can collect and bring to the cell (i, j) .
- ▶ It can reach this cell either from the adjacent cell $(i - 1, j)$ above it or from the adjacent cell $(i, j - 1)$ to the left of it. The largest numbers of coins that can be brought to these cells are $F(i - 1, j)$ and $F(i, j - 1)$, respectively.
- ▶ $F(i - 1, j)$ and $F(i, j - 1)$ are equal to 0 for their nonexistent neighbors

	1	2	3	4	5	6
1					●	
2		●		●		
3				●		●
4			●			●
5	●				●	

Coin-Collection Problem



$$F(i, j) = \max\{F(i - 1, j), F(i, j - 1)\} + c_{ij} \text{ for } 1 \leq i \leq n, 1 \leq j \leq m$$

$$F(0, j) = 0 \text{ for } 1 \leq j \leq m \text{ and } F(i, 0) = 0 \text{ for } 1 \leq i \leq n$$

- ▶ $c_{ij} = 1$ if there is a coin in cell (i, j) and $c_{ij} = 0$ otherwise

Algorithm

```
RobotCoinCollection(C[1..n, 1..m])  
//Applies dynamic programming to compute the largest number of  
//coins a robot can collect on an  $n \times m$  board by starting at (1, 1)  
//and moving right and down from upper left to down right corner  
//Input: Matrix C[1..n, 1..m] whose elements are equal to 1 and 0  
//for cells with and without a coin, respectively  
//Output: Largest number of coins the robot can bring to cell (n, m)
```

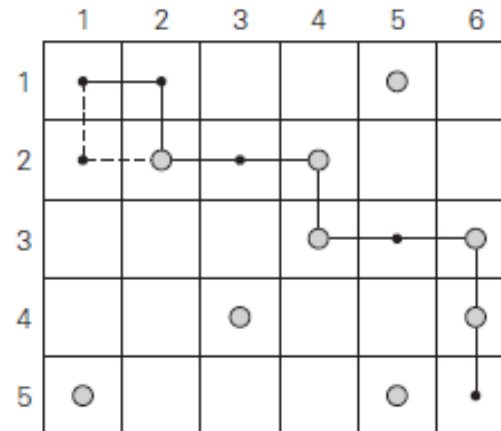
```
F[1, 1] ← C[1, 1];  
for j ← 2 to m do  
    F[1, j] ← F[1, j - 1] + C[1, j]  
for i ← 2 to n do  
    F[i, 1] ← F[i - 1, 1] + C[i, 1]  
    for j ← 2 to m do  
        F[i, j] ← max(F[i - 1, j], F[i, j - 1]) + C[i, j]  
return F[n, m]
```

	1	2	3	4	5	6
1					●	
2		●		●		
3				●		●
4			●			●
5	●				●	

(a)

	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	5

(b)



(c)

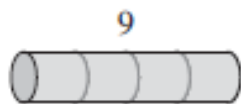
- (a) Coins to collect. (b) Dynamic programming algorithm results. (c) Two paths to collect 5 coins, the maximum number of coins possible

Rod-Cutting Problem

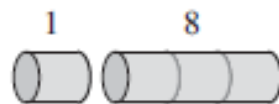
- ▶ Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \dots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.
 - Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.
- ▶ We can cut up a rod of length n in 2^{n-1} different ways, since we have an independent option of cutting, or not cutting, at distance i inches from the left end.

► Consider a case $n = 4$

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



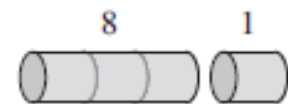
(a)



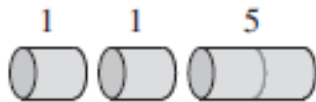
(b)



(c)



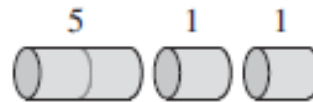
(d)



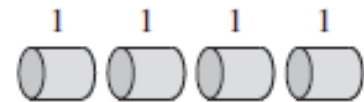
(e)



(f)



(g)



(h)

Optimal revenue $p_2 + p_2 = 10$

- ▶ A rod of length n in 2^{n-1} different ways
- ▶ For $i = 1, 2, \dots, n - 1$, we denote a decomposition using ordinary additive notation
 - $7 = 2 + 2 + 3$ means a rod of 7 length is cut into three pieces, two of length 2 and one of length 3
- ▶ If an optimal solution cuts the rod into k pieces, $1 \leq k \leq n$, of length i_1, i_2, \dots, i_k
$$n = i_1 + i_2 + \dots + i_k$$
$$r_n = p_{i_1} + p_{i_2} + \dots + p_{i_k}$$

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

- For the sample, we can determine the optimal revenue figures r_i for $i = 1, 2, \dots, 10$ by inspection

$r_1 = 1$ 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) :

- ▶ Generally, r_n for $n \geq 1$ in terms of optimal revenues from shorter rods
 - $r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$
 - p_n corresponds to no cut at all
 - $r_i + r_{n-i}$ corresponds to an initial cut into two pieces of size i and $n - i$ for each $i = 1, 2, \dots, n - 1$, then optimally cut those pieces further to obtain revenues r_i and r_{n-i}
- ▶ We don't know which i optimizes revenue, so we have to consider all possible i and pick the one that optimizes the revenue
- ▶ We also have the option of picking no i , i.e. selling the rod without cut

- ▶ Assumption to make the problem simpler
 - Piece of length i is left-hand end
 - Piece of length $n - i$ is the right hand remainder
 - Only the remainder may be further divided
- ▶ We view the cut of a length n rod as follows
 - A first piece followed by some decomposition of the remainder
 - i length of cut, p_i revenue of the i length of cut
 - r_{n-i} remainder length
 - $i = n$ corresponds to no cut, revenue p_n , remainder has size 0, $r_0 = 0$

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

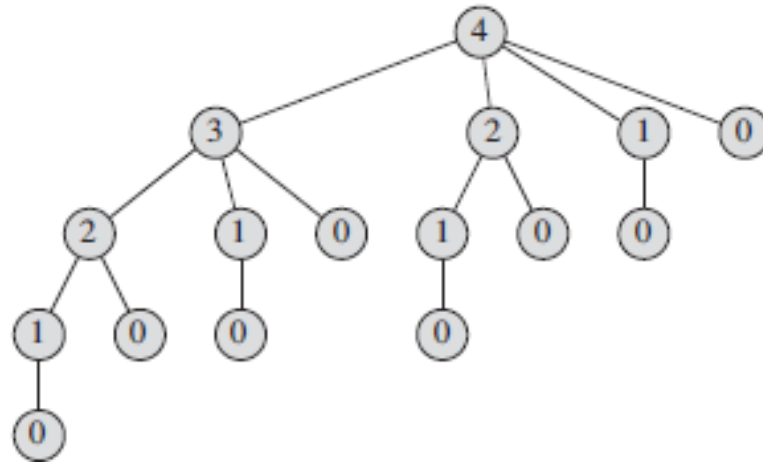


Recursive Top-down Implementation

```
CUT-Rod (p, n)
  if n == 0 then
    return 0
  q =  $-\infty$ 
  for i = 1 to n
    q = max(q; p[i] + CUT-Rod(p; n-i))
  return q
```

Analysis

- ▶ An example with $n = 4$, recursive call tree



- ▶ In general, this recursion tree has 2^n nodes and 2^{n-1} leaves
- ▶ CUT-Rod explicitly considers all the 2^{n-1} possible ways of cutting up a rod of length n .

DP for Optimal Rod-Cutting

► Top-down with memorization

```
MEMOIZED-CUT-ROD(p, n)
```

```
  r[0...n]: an array
```

```
  for i = 0 to n
```

```
    r[i] =  $-\infty$ 
```

```
  return MEMOIZED-CUT-ROD-AUX (p, n, r)
```

```
MEMOIZED-CUT-ROD-AUX (p, n, r)
```

```
  if r[n] > 0 then
```

```
    return r[n]
```

```
  if n == 0 then
```

```
    q = 0
```

```
  else
```

```
    q =  $-\infty$ 
```

```
    for i = 1 to n
```

```
      q = max (q, ; p[i] + MEMOIZED-CUT-ROD-AUX (p, n-i, r)
```

```
    r[n] = q
```

```
  return q
```




DP for Optimal Rod-Cutting...

► Bottom-up approach

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

$r[0..n]$: a new array

$r[0] = 0$

for $j = 1$ to n

$q = -\infty$

 for $i = 1$ to j

$q = \max(q, p[i] + r[j-i])$

$r[j] = q$;

return $r[n]$

Analysis



- ▶ Complexity
 - Top-down: $\Theta(n^2)$
 - Bottom-up: $\Theta(n^2)$
- ▶ Our dynamic-programming solutions to the rod-cutting problem return the value of an optimal solution, but they do not return an actual solution
 - a list of piece sizes.

Reconstruction a solution

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

$r[0..n]$ and $s[0..n]$: arrays

$r[0]=0$

for $j = 1$ to n

$q = -\infty$

 for $i = 1$ to j

 if $q < p[i] + r[j-i]$ then

$q = p[i] + r[j-i]$

$s[j] = i$

$r[j] = q$

return r and s

Reconstruction a solution...

```
PRINT-CUT-ROD-SOLUTION (p, n)
```

```
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD (p, n)
```

```
while n > 0
```

```
    print s[n]
```

```
    n = n - s[n]
```

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
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

► $n = 10$, no cut

► $n = 7$, cuts 1, 6

Related References



- ▶ Introduction to the Design and Analysis of Algorithms, A. Levitin, 3rd Ed., Pearson 2011.
 - Chapters 8.1
- ▶ Introduction to Algorithms, T. H. Cormen, 3rd Ed, MIT Press 2009.
 - Chapters 15.1