More On Dynamic Programming

Rod Cutting Problem

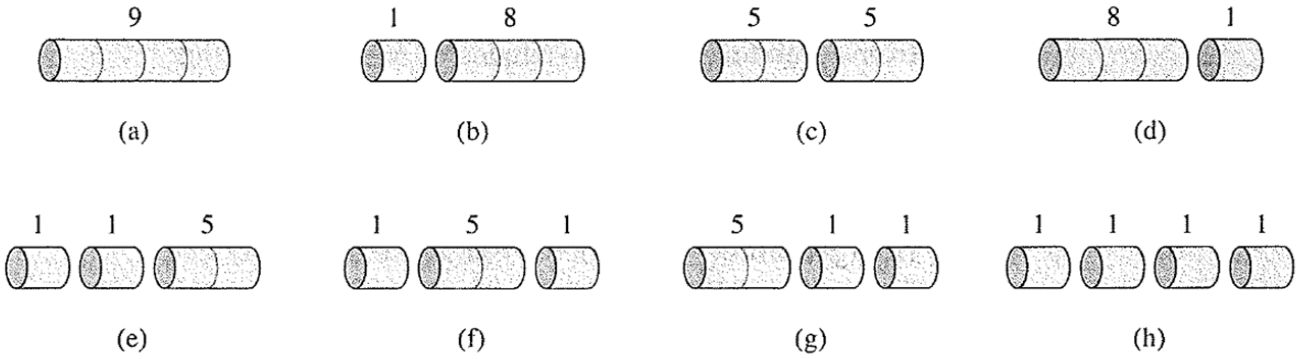
- You are given a rod of length n ≥ 0 (n in inches)
- A rod of length *i* inches will be sold for p_i dollars
- Cutting cost is free (simplifying assumption)
- Problem: given a table of prices p_i , determine the maximum revenue r_n obtainable by cutting up the rod into pieces and selling each of the pieces.

Rod Cutting Example

- We are given prices p_i and rods of length 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

- How many ways to cut a rod of length 4 ?



Rod Cutting Example

- how many different ways can we cut a rod of length n?

Proof Details: a rod of length *n* can have exactly *n*-1 possible cut positions – choose $0 \le k \le n-1$ actual cuts. We can choose the *k* cuts (without repetition) anywhere we want, so that for each such *k* the number of different choices is

$$\binom{n-1}{k}$$

When we sum up over all possibilities (*k* = 0 to *k* = *n*-1):

$$\sum_{k=0}^{n-1}\binom{n-1}{k} = \sum_{k=0}^{n-1}\frac{(n-1)!}{k!(n-1-k)!} = (1+1)^{n-1} = 2^{n-1}.$$

For a rod of length n: 2^{n-1}.

⇨ Exponential: infeasible for large n

Characterizing an Optimal Solution

- We can calculate the maximum revenue r_n in terms of optimal revenues for shorter rods
- $r_n = \max(p_n , r_1 + r_{n-1} , r_2 + r_{n-2} , ..., r_{n-1} + r_1)$
  - p_n if we do not cut at all
  - $r_1 + r_{n-1}$ if we take the sum of optimal revenues for 1 and n-1
  - $r_2 + r_{n-2}$ if we take the sum of optimal revenues for 2 and n-2
  - . . .

A different view of the problem

- Decomposition in
  - A first, left-hand piece of length i
  - A right-hand reminder of length n-i
  - Only the reminder is further divided
  - Set $r_0 = 0$ and

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

  - Thus, need solution to only one subproblem

Top-down implementation

```
CUT-ROD(p,n)
    if n==0
        return 0
    q = -∞
    for i=1 to n
        q=max{q,p[i]+CUT-ROAD(p,n-i)}
    return q
```

- Time complexity: $T(n)=1+T(1)+T(2)+\ldots+T(n-1)$
  - $T(n)=O(2^n)$

Dynamic Programming Solution for Rod cutting

- p_i are the problem inputs.
- r_i is max profit from cutting rod of length i .
- Goal is to calculate r_n
- r_i defined by

$$r_1 = 1 \text{ and } r_n = \max_{1 \le i \le n}(p_i + r_{n-i})$$

- Iteratively fill in r_i table by calculating r_1, r_2, r_3 .....
- r_n is final solution

i	1	2	3	4	. . . . .	n
$\mathbf{r_i}$	p_1				. . . . .	

Bottom-Up-Cut-Rod(p, n)

```
r[0] = 0; // Array r[0...n] stores the computed optimal values
for j = 1 to n do
    // Consider problems in increasing order of size
    q = -∞;
    for i = 1 to j do
        // To solve a problem of size j, we need to consider all
           decompositions into i and j - i
        q = max(q, p[i] + r[j - i]);
    end
    r[j] = q;
end
return r[n];
```

Outputting the Cutting

- Algorithm only computes ri . It does not output the cutting.
- Easy fix
  - When calculating $r_j = \max_{1 \le i \le j}(p_i + r_{j-i})$
    store value of i that achieved this max in new array s[j]
  - This j is the size of last piece in the optimal cutting.
- After algorithm is finished, can reconstruct optimal cutting by unrolling the sj .

Extended-Bottom-Up-Cut-Rod(p, n)

```
// Array s[0...n] stores the optimal size of the first piece to
   cut off
r[0] = 0; // Array r[0...n] stores the computed optimal values
for j = 1 to n do
    q = -∞;
    for i = 1 to j do
        // Solve problem of size j
        if q < p[i] + r[j - i] then
            q = p[i] + r[j - i];
            s[j] = i; // Store the size of the first piece
        end
    end
    r[j] = q;
end
while n > 0 do
    // Print sizes of pieces
    Print s[n];
    n = n - s[n];
end
```

Output Example

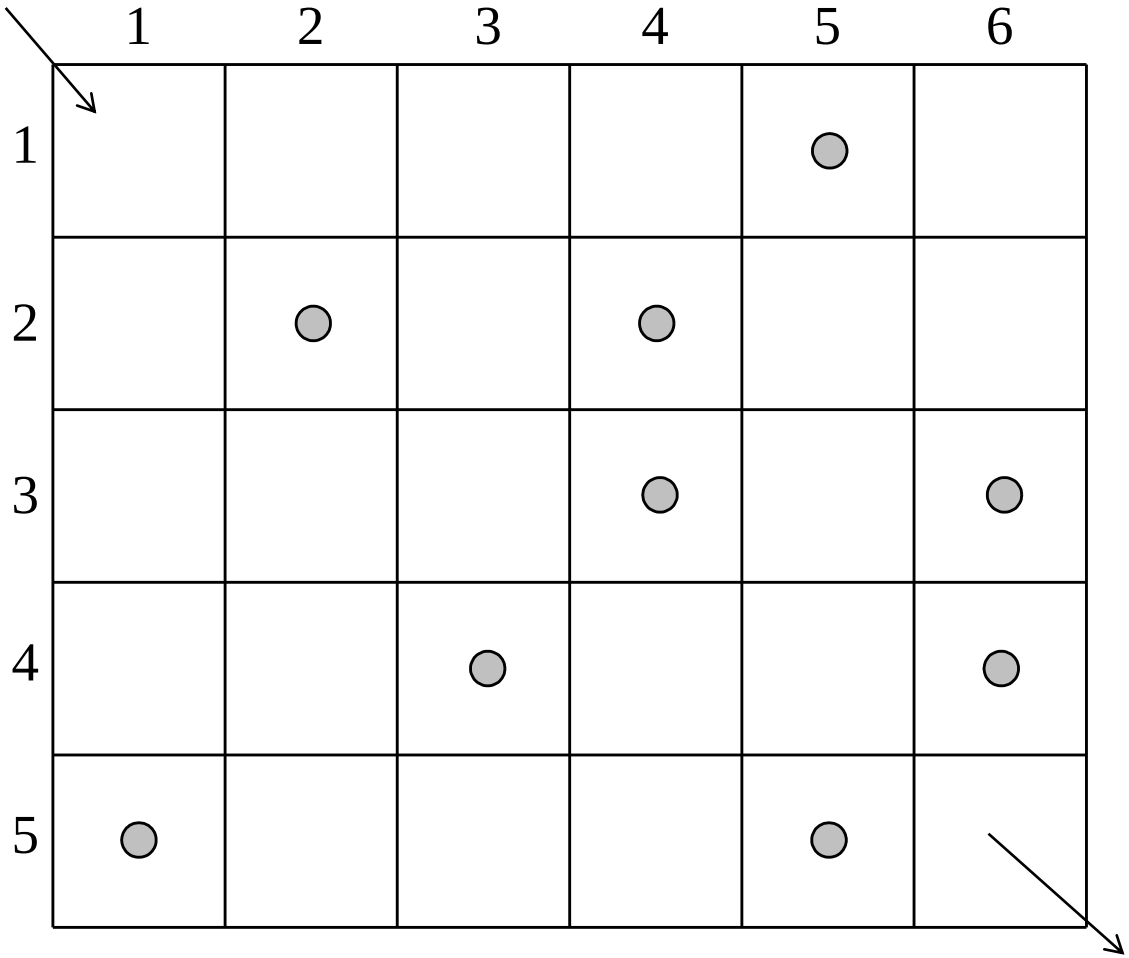
Example:

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

Coin Collecting Problem by Robot

- Several coins are placed in cells of an $n \times m$ board. 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.
- Design an algorithm to find the maximum number of coins the robot can collect and a path it needs to follow to do this

Coin Collecting Problem by Robot



Solution to the coin-collecting problem

- Let F(i,j) be the largest number of coins the robot can collect and bring to cell (i,j) in the ith row and jth column.
- The largest number of coins that can be brought to cell (i,j):
  - from the left neighbor ?
  - from the neighbor above?
- The recurrence:

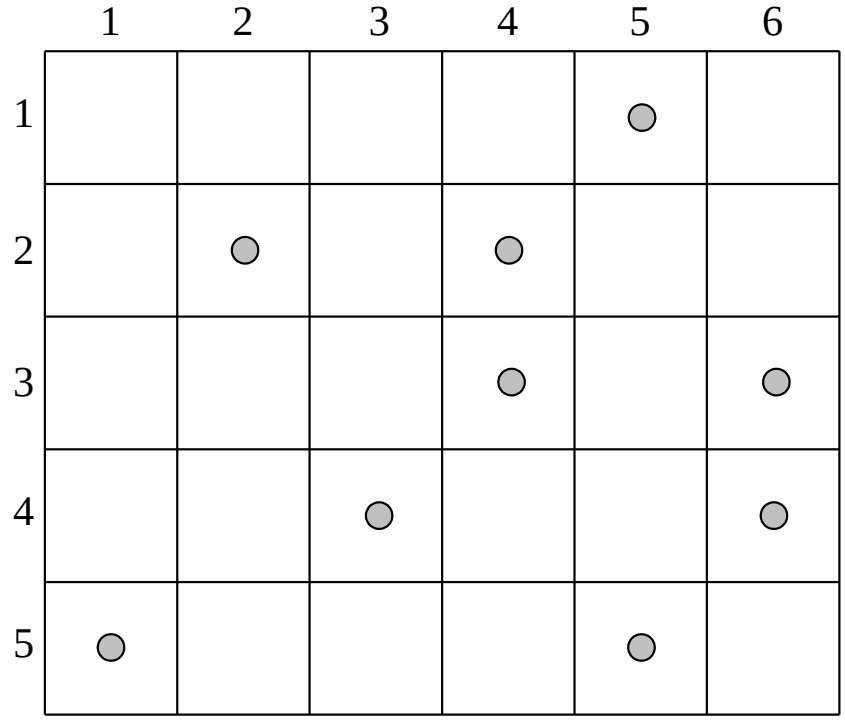
  $F(i, j) = \max\{F(i\text{-}1, j),\ F(i, j\text{-}1)\} + c_{ij}$ for $1 \le i \le n$, $1 \le j \le m$
- where $c_{ij} = 1$ if there is a coin in cell (i,j),
  and $c_{ij} = 0$ otherwise
  - $F(0, j) = 0$ for $1 \le j \le m$ and $F(i, 0) = 0$ for $1 \le i \le n$

Solution to the coin-collecting problem

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

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

$F(0, j) = 0$ for $1 \le j \le m$ and $F(i, 0) = 0$ for $1 \le i \le n$.



	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**

Optimal Path

Tracing the computation ***backward*** *makes it possible to get an optimal path*:

1. if $F(i-1, j)_{top} > F(i, j-1)_{left}$, an optimal path to cell (i, j) must come down from the adjacent cell above it. i.e, cell (i – 1, j);
2. if $F(i-1, j) < F(i, j-1)$, and optimal path to cell (i, j) must come from the adjacent cell on the left, cell (i, j-1); and
3. if $F(i-1, j) = F(i, j-1)$, it can reach cell (i, j) from either direction.

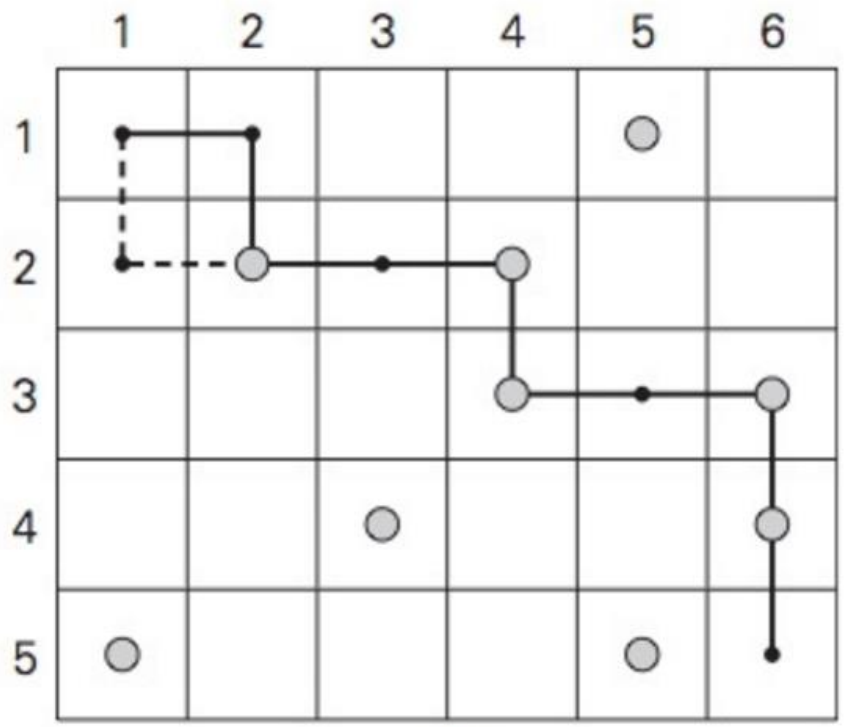
Solution to the coin-collecting problem

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

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

$F(0, j) = 0$ for $1 \le j \le m$ and $F(i, 0) = 0$ for $1 \le i \le n$.

	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**



Robot Coin Collection Algorithm

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)

$F[1, 1] \leftarrow C[1, 1]$; **for** $j \leftarrow 2$ **to** m **do** $F[1, j] \leftarrow F[1, j-1] + C[1, j]$
for $i \leftarrow 2$ **to** n **do**
    $F[i, 1] \leftarrow F[i-1, 1] + C[i, 1]$
    for $j \leftarrow 2$ **to** m **do**
        $F[i, j] \leftarrow \max(F[i-1, j], F[i, j-1]) + C[i, j]$
return $F[n, m]$