Dynamic Programming

Dynamic Programming

- A general algorithm design technique for solving problems defined by recurrences with overlapping subproblems
- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS
- "Programming" here means "planning"
- 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

Example: Fibonacci numbers

- Recall definition of Fibonacci numbers:

$$F(n) = F(n-1) + F(n-2)$$
$$F(0) = 0$$
$$F(1) = 1$$

- Computing the n^{th} Fibonacci number recursively (top-down):

$F(n)$

$F(n-1) \quad + \quad F(n-2)$

$F(n-2) + F(n-3) \quad F(n-3) + F(n-4)$

...

Example: Fibonacci numbers

Computing the n^{th} Fibonacci number using bottom-up iteration and recording results:

$F(0) = 0$

$F(1) = 1$

$F(2) = 1+0 = 1$

…

$F(n-2) =$

$F(n-1) =$

$F(n) = F(n-1) + F(n-2)$

0	**1**	**1**	**. . .**	***F*(*n*-2)**	***F*(*n*-1)**	***F*(*n*)**

Efficiency:

- time: $\Theta(n)$
- space: $\Theta(n)$

Example: Fibonacci numbers

Computing the n^{th} Fibonacci number using bottom-up iteration and recording results just the last 2 values:

$F(0) = 0$
$F(1) = 1$
$F(2) = 1+0 = 1$
…
$F(n-2) =$
$F(n-1) =$
$F(n) = F(n-1) + F(n-2)$

$F(n-2)$	$F(n-1)$	$F(n)$

Efficiency:
- time: $\Theta(n)$
- space: $\Theta(1)$

Dynamic Programming

- Majority of dynamic programming deals with optimization
- Principal of optimization:
- An optimization solution to any instance of an optimization problem is composed of optimal solutions to its sub-instances
- There are some rare exceptions:
  - Find shortest path

Coin-Row Problem

- Pick most value of a row of n coins with values $c_1, c_2, \ldots, c_n$
- Values are not necessarily distinct
- No two adjacent coins in the initial row can be picked

- $F(n)$: $\max$ *amount pickable from a row of n coins*
  $F(n-2) + c_n$: *solution contains the last coin*
  $F(n-1)$: *solution does not contain the last coin*

- $F(0) = 0$
- $F(1) = c_1$
- $F(n) = \max\{\, c_n + F(n-2), F(n-1)\}$

Coin-Row Problem

ALGORITHM $CoinRow(C[1..n])$

//Applies formula (8.3) bottom up to find the maximum amount of money
//that can be picked up from a coin row without picking two adjacent coins
//Input: Array $C[1..n]$ of positive integers indicating the coin values
//Output: The maximum amount of money that can be picked up
$F[0] \leftarrow 0; \quad F[1] \leftarrow C[1]$
for $i \leftarrow 2$ **to** n **do**
$\quad F[i] \leftarrow \max(C[i] + F[i-2], F[i-1])$
return $F[n]$

Time efficiency: $\theta(n)$
Space efficiency: $\theta(n)$

Coin-Row Problem

- $F(0) = 0$
- $F(1) = c_1$
- $F(n) = \max\{ c_n + F(n-2), F(n-1)\}$

$F[0] = 0$, $F[1] = c_1 = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5					

$F[2] = \max\{1 + 0, 5\} = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5				

$F[3] = \max\{2 + 5, 5\} = 7$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7			

$F[4] = \max\{10 + 5, 7\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15		

$F[5] = \max\{6 + 7, 15\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	

$F[6] = \max\{2 + 15, 15\} = 17$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	**17**

Change-Making Problem

- Give change for amount m
- Using the minimum amount of coins $(d_1 < d_2 < \cdots < d_m)$
- Unlimited number of each coin is available

- $F(n)$: *minimum number of coins whose value adds up to n*
  $F(n - d_j) + 1$: *use coin* d_j *to give change for amount of n*

- $F(0) = 0$
- $F(n) = \min\{F(n - d_j)\}, j: n \geq d_j$

Change-Making Problem

ALGORITHM *ChangeMaking*$(D[1..m], n)$

```
//Applies dynamic programming to find the minimum number of coins
//of denominations d1 < d2 < ··· < dm where d1 = 1 that add up to a
//given amount n
//Input: Positive integer n and array D[1..m] of increasing positive
//       integers indicating the coin denominations where D[1] = 1
//Output: The minimum number of coins that add up to n
F[0] ← 0
for i ← 1 to n do
    temp ← ∞;  j ← 1
    while j ≤ m and i ≥ D[j] do
        temp ← min(F[i − D[j]], temp)
        j ← j + 1
    F[i] ← temp + 1
return F[n]
```

Time efficiency: $O(nm)$
Space efficiency: $\theta(n)$

Change-Making Problem

- $F(0) = 0$
- $F(n) = \min\{F(n - d_j)\}, j: n \geq d_j$

$F[0] = 0$

n	0	1	2	3	4	5	6
F	0						

$F[1] = \min\{F[1 - 1]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1					

$F[2] = \min\{F[2 - 1]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2				

$F[3] = \min\{F[3 - 1], F[3 - 3]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1			

$F[4] = \min\{F[4 - 1], F[4 - 3], F[4 - 4]\} + 1 = 1$

n	0	1	2	3	4	5	6
F	0	1	2	1	1		

$F[5] = \min\{F[5 - 1], F[5 - 3], F[5 - 4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	

$F[6] = \min\{F[6 - 1], F[6 - 3], F[6 - 4]\} + 1 = 2$

n	0	1	2	3	4	5	6
F	0	1	2	1	1	2	**2**

Coin-Collecting Problem

- An $n \times m$ board with a coin in some cells
- Start from upper left and reach bottom right cell
- Each time move one cell to right or one cell to down
- Collect maximum number of coins

- $F(i,j)$: $maximun\ number\ of\ coins\ collectable\ up\ to\ cell\ (i,j)$
- $c_{i,j} = \begin{cases} 1 \rightarrow coin\ exist\ in\ cell\ (i,j) \\ 0\ \rightarrow coin\ not\ in\ cell\ (i,j) \end{cases}$

- $F(0,j) = 0, 1 \le j \le m$
- $F(i,0) = 0, 1 \le i \le n$
- $F(n) = \mathrm{m}ax\{F(i-1,j), F(i,j-1)\} + c_{i,j}$

Coin-Collecting Problem

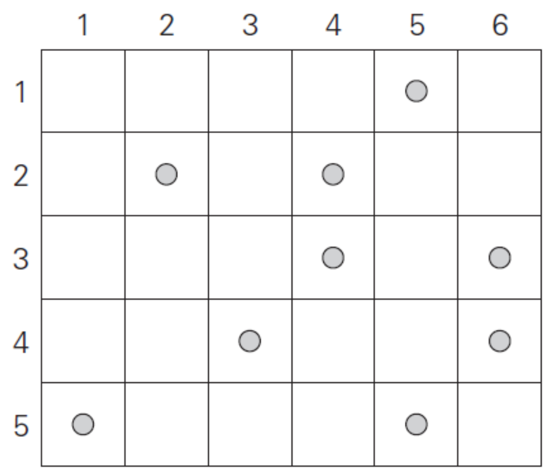
ALGORITHM *RobotCoinCollection*($C[1..n, 1..m]$)

//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 $F[n, m]$

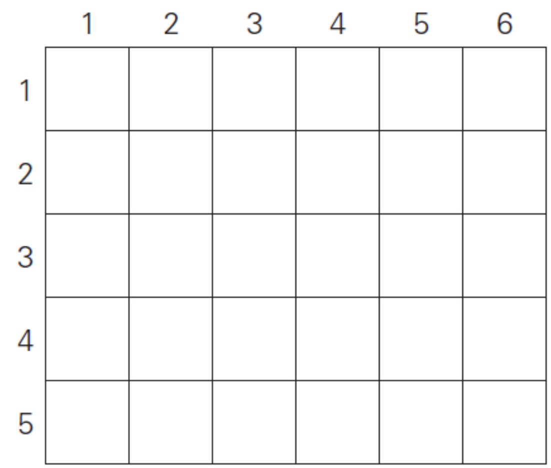
Time efficiency: $\theta(nm)$
Space efficiency: $\theta(nm)$
Optimal path: $\theta(n+m)$

Coin-Collecting Problem

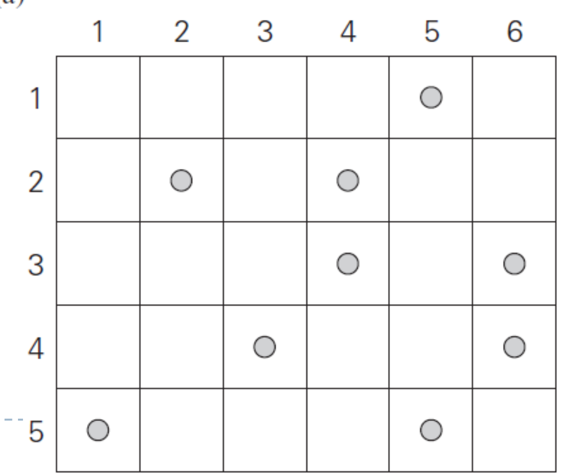
- $F(0, j) = 0, 1 \leq j \leq m$
- $F(i, 0) = 0, 1 \leq i \leq n$
- $F(n) = \max\{F(i-1, j), F(i, j-1)\} + c_{i,j}$



(a)

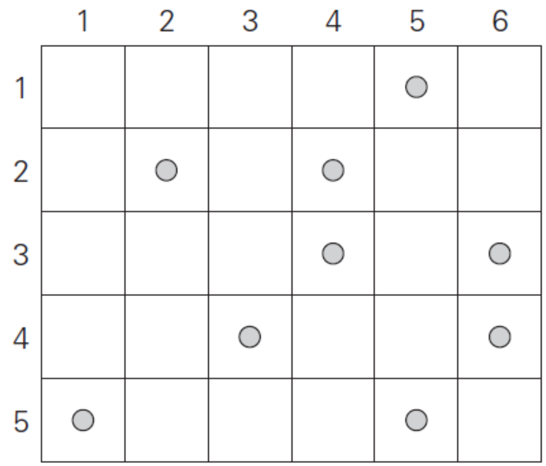


(b)



Coin-Collecting Problem

- $F(0, j) = 0, 1 \leq j \leq m$
- $F(i, 0) = 0, 1 \leq i \leq n$
- $F(n) = \max\{F(i-1, j), F(i, j-1)\} + c_{i,j}$



(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)