Dynamic Programming

1

CSI456 – Algorithm Analysis & Design

University of Ashesi, Brekuso

E/R, Ghana

March 2024

https://www.youtube.com/watch?v=Hdr64lKQ3e4

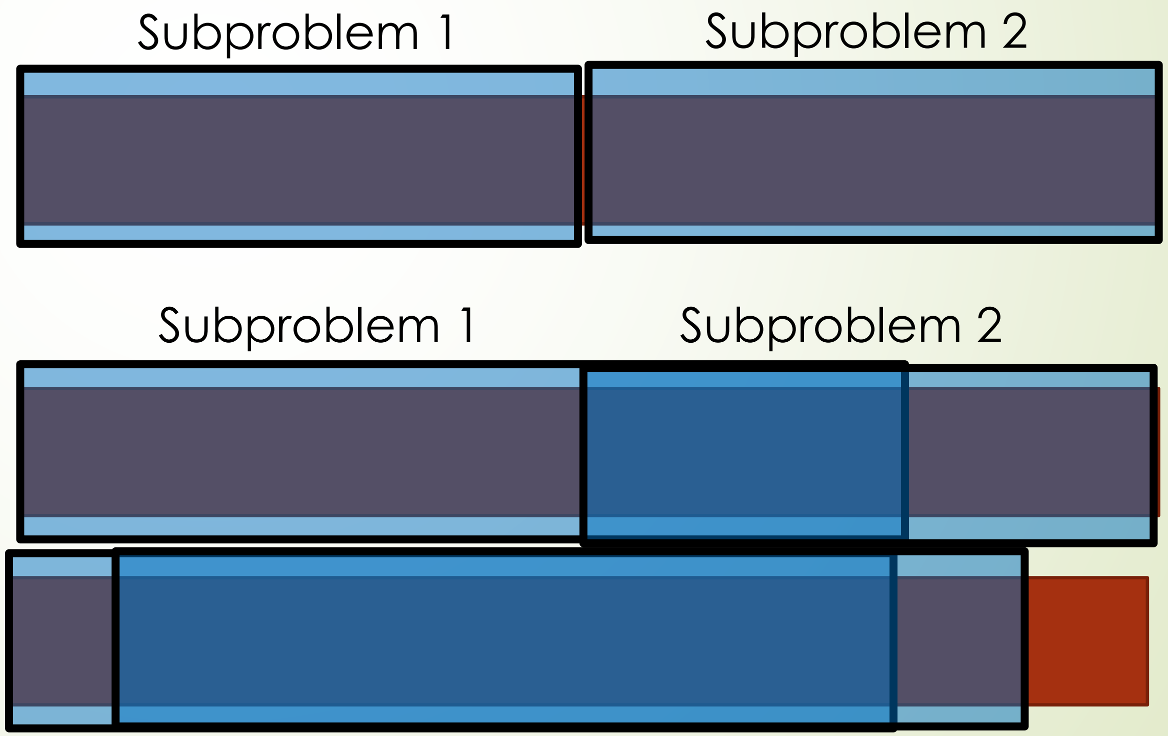
Please watch the above link at home (about 19 minutes)

Dynamic Programming

Dynamic Programming is a general algorithm design technique for solving problems defined by **recurrences** with **overlapping subproblems**

Divide & conquer:
non-overlapping subproblems

Dynamic programming:
overlapping subproblems



Example 1: Fibonacci numbers

- Recall definition of Fibonacci numbers:

  $F(n) = F(n\text{-}1) + F(n\text{-}2)$
  $F(0) = 0$
  $F(1) = 1$

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

$F(n)$

$F(n\text{-}1) \quad + \quad F(n\text{-}2)$

$F(n\text{-}2) \quad + \quad F(n\text{-}3) \qquad F(n\text{-}3) \quad + \quad F(n\text{-}4)$

…

$T(n) = O(2^n)$ using brute force

Example 1: Fibonacci numbers (cont.)

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

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

…

$F(n-2) =$

$F(n-1) =$

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

Efficiency:
- time?
- space?

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

Example: F(6)

0	1	1	2	3	5	8

Top-Down Approach(Recursion) to Fibonacci -Memoization

```
int Fib-Top-Down(int N)
 //initialize array cells to -1 for all values before start
   if result[N] == -1
       if (N <= 1)
           result[N] = N      //memoize = cache it
       else
           result[N] = Fib-Top-Down(N-1) + Fib-Top-Down(N-2)
   return result[N]
```

Fast Fibonacci – (Bottom-Up (Iterative)-Fibonacci) - Tabulation

```
Algorithm Fast-Fibonacci(n)
//int[] fibo = new int[n+1]; //tabulation
fib[0]   = fib[1] = 1.
  for (int  i = 2, i <= n; i++)
      fib[i] =  fib[i - 2] + fib[i - 1].
 end forloop
return fib[n].
EndAlg
```

Compare this with the original recursive fib(n) algorithm using brute force.
Time complexity for this is O(n)

Space Complexity : O(n) because of the array we must keep

Example 2: Coin-row problem

There is a row of *n* coins whose values are some positive integers $c_1, c_2, \dots, c_n$, not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.

E.g.: 5, 1, 2, 10, 6, 2. What is the best selection?

DP solution to the coin-row problem

Let F(*n*) be the maximum amount that can be picked up from the row of n coins. To derive a recurrence for F(*n*), we partition all the allowed coin selections into two groups:

those without last coin – the max amount is ?

those with the last coin -- the max amount is ?

$c_1, c_2, c_3, c_4, c_5, c_6, \ldots, c_{n-4}, c_{n-3}, c_{n-2}, c_{n-1}, c_n$

Thus we have the following recurrence

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

$F(0) = 0,\ F(1) = c_1$

DP solution to the coin-row problem (cont.)

$$F(n) = \max\{c_n + F(n-2),\ F(n-1)\} \text{ for } n > 1,$$

$$F(0) = 0,\ F(1) = c_1$$

index	0	1	2	3	4	5	6
Coins(c_i)	--	5	1	2	10	6	2
F( )	0	5	5				

$$F(2) = \max\{c_2 + F(0),\ F(1)\}$$

DP solution to the coin-row problem (cont.)

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

$F(0) = 0,\ F(1)=c_1$

index	0	1	2	3	4	5	6
Coins(c_i)	--	5	1	2	10	6	2
F( )	0	5	5	7			

$F(3) = \max\{c_3 + F(1),\ F(2)\}$

DP solution to the coin-row problem (cont.)

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

$F(0) = 0,\ F(1)=c_1$

index	0	1	2	3	4	5	6
Coins(c_i)	--	5	1	2	10	6	2
F( )	0	5	5	7	15		

$F(4) = \max\{c_4 + F(2),\ F(3)\}$

DP solution to the coin-row problem (cont.)

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

$F(0) = 0,\ F(1)=c_1$

index	0	1	2	3	4	5	6
Coins(c_i)	--	5	1	2	10	6	2
F( )	0	5	5	7	15	15	

$F(5) = \max\{c_5 + F(3),\ F(4)\}$

DP solution to the coin-row problem (cont.)

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

$F(0) = 0,\ F(1)=c_1$

index	0	1	2	3	4	5	6
coins	--	5	1	2	10	6	2
F( )	0	5	5	7	15	15	17

$F(6) = \max\{c_6 + F(4),\ F(5)\}$

DP solution to the coin-row problem (cont.)

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

$F(0) = 0,\ F(1) = c_1$

index	0	1	2	3	4	5	6
coins	--	5	1	2	10	6	2
F( )	0	5	5	7	15	15	17

Max amount: 17
Coins of optimal solution: C_6, C_4, C_1
Time efficiency: $O(n)$
Space efficiency: $O(n)$

Note: All smaller instances were solved.

Pseudocode for CoinROw Problem using Dynamic Programming

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

//Auxiliary array (space) F - hence space complexity is O(n)

```
F [0] ← 0;
F [1] ← C[1]
for i ← 2 to n do
    F [i] ← max(C[i] + F [i − 2], F [i − 1])
return F [n]
EndAlg
```

Time Complexity : O(n)

Space Complexity : O(n)

Example 3: Change Making Problem

- Give change for amount *n* using the minimum number of coins of denominations d1 < d2 < . . . < dm.
- Example: What's the minimum number of coins needed to give change for 97 pesewas, using the coin denominations in Ghana's currency:

1p 5p 10p 20p 50p ¢1 (100p)

Answer: 6 coins: 50p, 20p, 20p, 5p, 1p, 1p

Example 3: Change Making Problem

- Give change for amount *n* using the minimum number of coins of denominations d1 < d2 < . . . < dm.
- Example: What's the minimum number of coins needed to give change for 97 pesewas, using the coin denominations in Ghana's currency:

1p 5p 10p 20p 50p ¢1 (100p)

- Suppose we had a 48p coin, in addition to existing coin denominations:

1p 5p 10p 20p 48p 50p ¢1 (100p)

Now need 3 coins:

48p, 48p, 1p

Change Making Problem (contd.)

- With "standard" denominations, the change-making problem can be solved using a "greedy" algorithm
  - More on this later!
- However, the general form of the problem with denominations of arbitrary values needs to be solved with dynamic programming

Change Making Problem (contd.)

- Let **F(*n*)** be the minimum number of coins whose value adds up to *n*
- Observation: $n = d_j + (n - d_j)$
- Example: with n = 97p:
  - **97p** = 50p + (97p – 50p) =50p + **47p**
  - OR **97p** = 48p + (97p – 48p) = 48p + **49p**
  - OR **97p** = 20p + (97p – 20p) = 20p + **77p**
  - OR **97p** = 10p + (97p – 10p) = 10p + **87p**
  - OR **97p** = 5p + (97p – 5p) = 5p + **92p**
  - OR **97p** = 1p + (97p - 1p) = 1p + **96p**

1	5	10	20	48	50

Each of these options involve selecting **1 coin** of a given denomination (shown in red), and then recursively finding the number of coins required to make change for a subproblem (shown in blue)

- Thus, F(n) = 1 + $\min_{j:dj \le n} \{F(n - d_j)\}$ for $n > 0$

  F(0) = 0

We can use the following recursive relation.
Can be computed by filling from left to right, a 1-row table with *n* entries

Change Making by Dynamic Programming

- Example: n = 6, Denominations = 1, 3, 4

int F[n+1] = F[6+1] = F[7] i.e. create an array of n+1 spaces
n = is the amount of money whose change we seek

$F[0] = 0$

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 Changing Algorithm


```
Algorithm CoinChange(int d[], int n, int k)
//k = length of d, d contains the denominations
minCoins = MAXINT; //some huge value
int[] F = new int[n+1];// change array
for j in 1 to n // change for n
    for i in 1 to k //
    if j  >= d[i]
      minCoins = min(minCoins, 1 + F[j-d[i]])
     endif
   endFor
 EndFor
  return minCoins;
EndAlg
```

Analysis of Coin Changing Algorithm

Time complexity = O(nk)

Space Complexity = O(k) -- length of array d

Coin Changing Algorithm

```
CHANGE(d, k, n)
C[0] ← 0
for p ← 1 to n
    min ← ∞
    for i ← 1 to k
        if d[i] ≤ p then
            if 1 + C[p − d[i]] < min then
                min ← 1 + C[p − d[i]]
                coin ← i
    C[p] ← min
    S[p] ← coin
return C and S
```

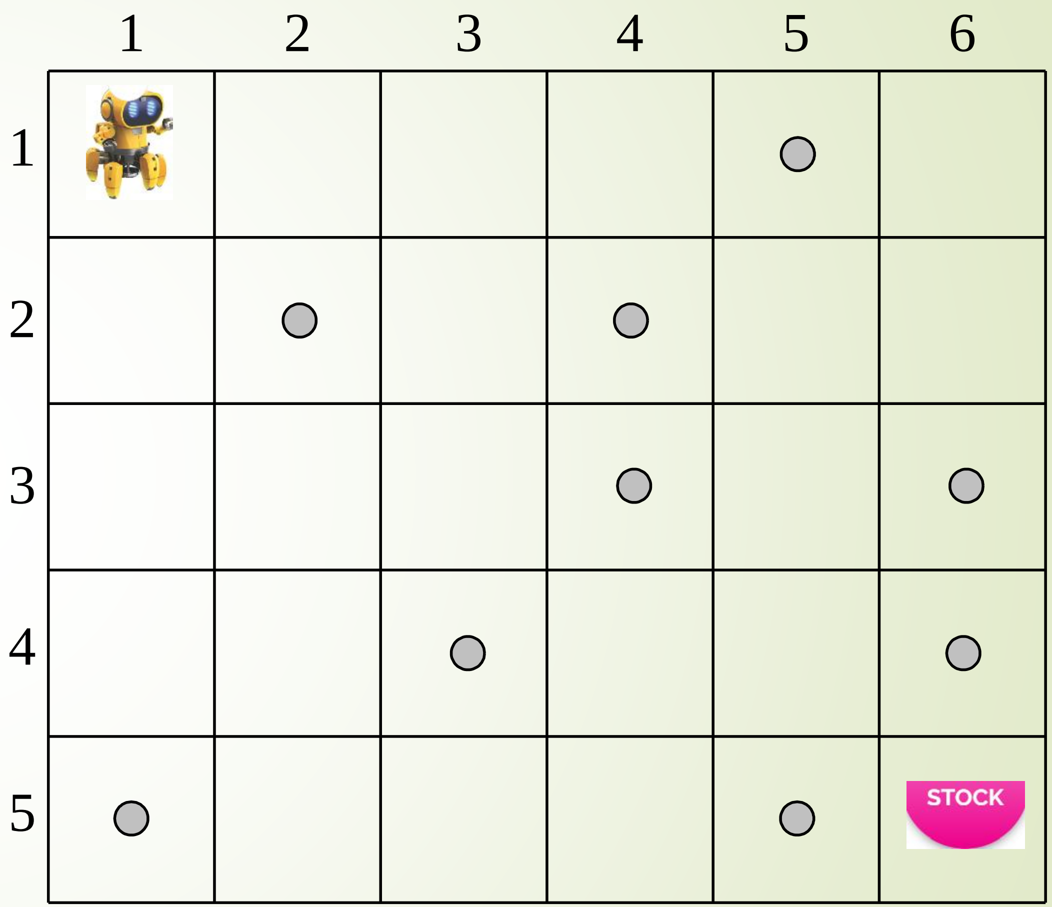
IF we want to return both the minimum coins as well as those denominations that produced the number of the minimum coins, we can use the following algorithm.

In that case we have time Complexity as being O(nk)

Space complexity = O(n)+O(n) = O(n)

Example 4: Coin-collecting by robot

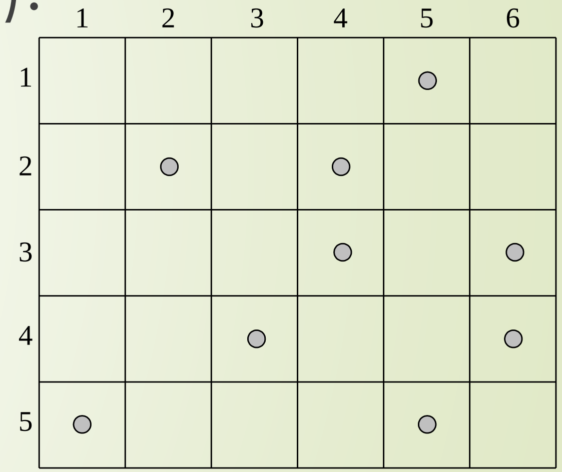
Several coins are placed in cells of an $n \times m$ board. A robot, located in the upper left cell of the board, needs to collect as many of the coins as possible and bring them to the bottom right cell. On each step, the robot can move either one cell to the right or one cell down from its current location.



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 *i*th row and *j*th column.

The largest number of coins that can be brought to cell (*i*,*j*):



from the left neighbor ?

from the neighbor above?

The recurrence:

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

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

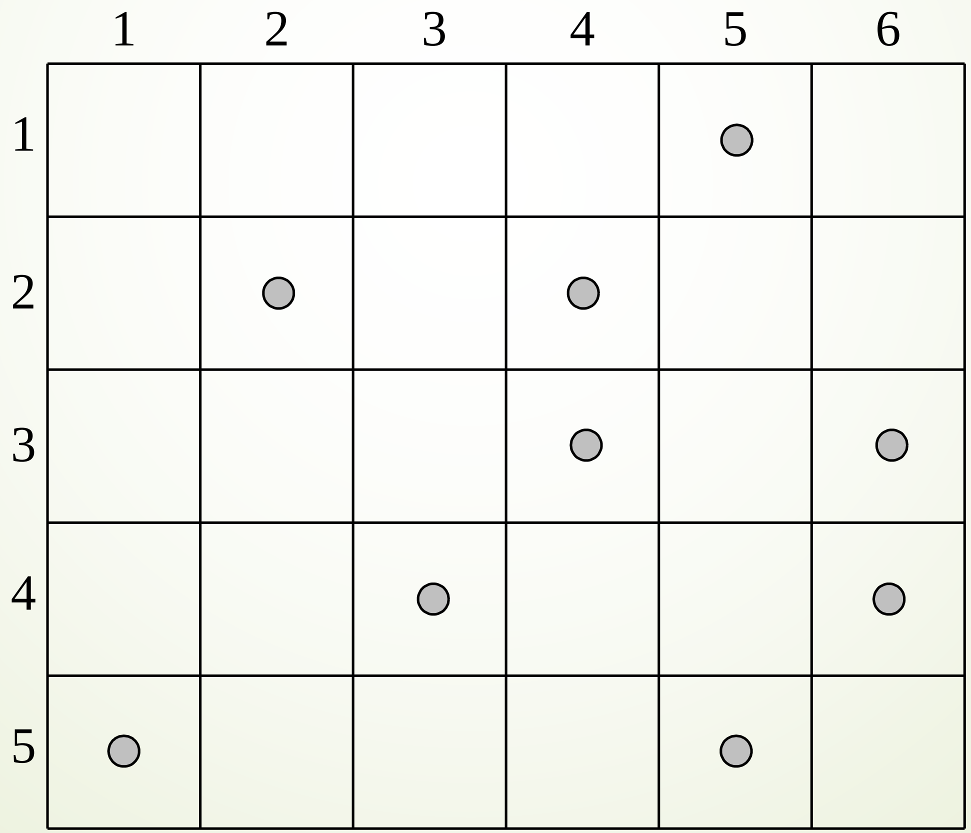
$F(0, j) = 0$ for $1 \leq j \leq m$ and $F(i, 0) = 0$ for $1 \leq i \leq n$.

Solution to the coin-collecting problem (cont.)

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

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

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

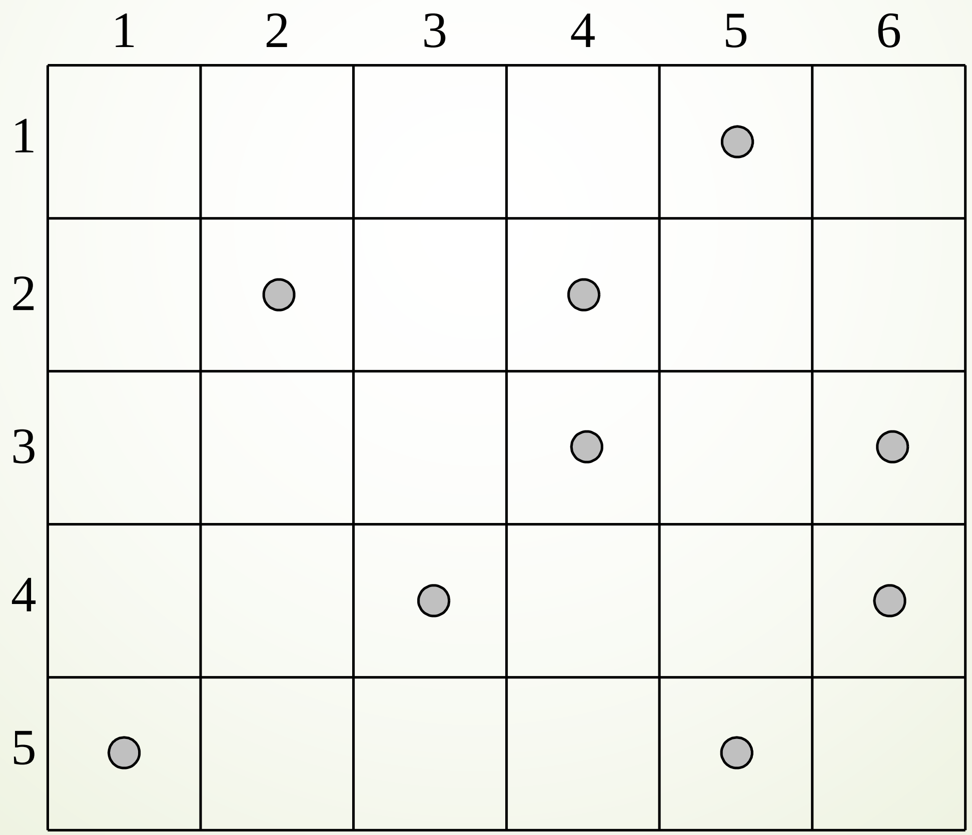


Solution to the coin-collecting problem (cont.)

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

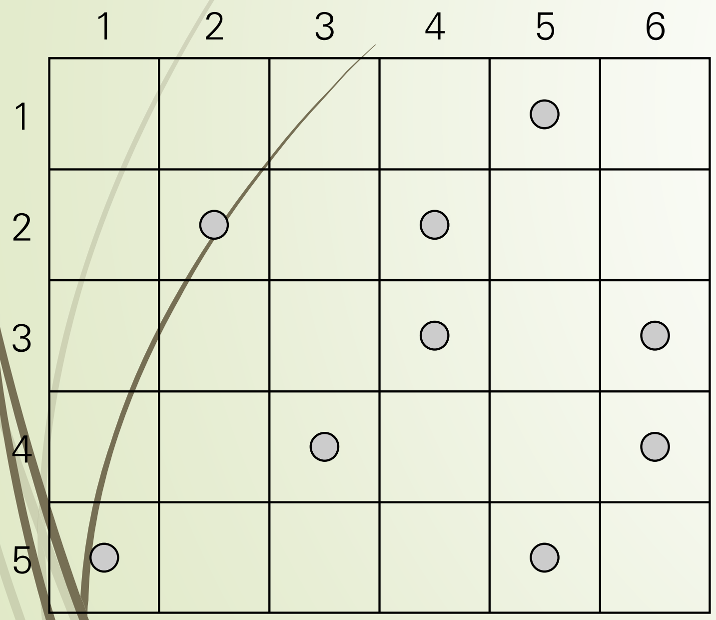


Solution to the coin-collecting problem (cont.)

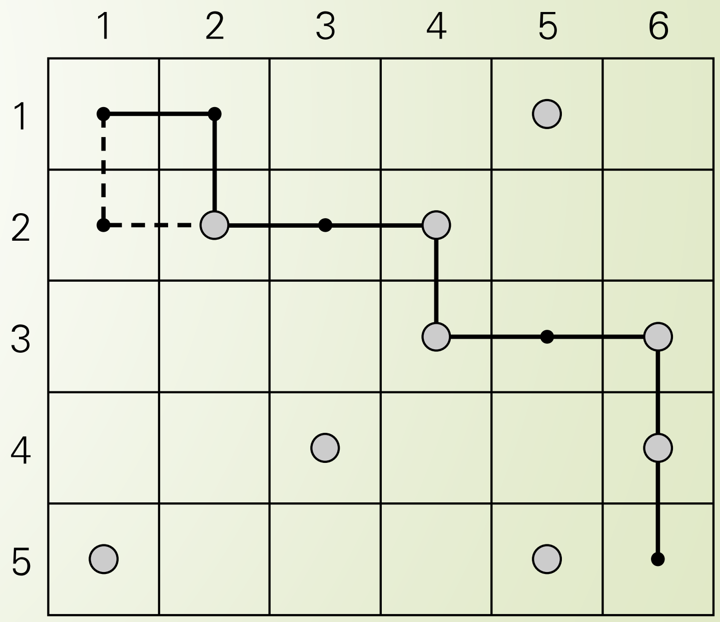
$$F(i, j) = \max\{F(i-1, j),\ F(i, j-1)\} + c_{ij} \qquad \text{for } 1 \le i \le n,\ 1 \le j \le m$$

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**



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 Complexity: Θ(nm)
Space Complexity: Θ(nm)

In-class exercises - Coin-row problem : 1

- Given a row of coins of the following values: 7, 5, 2, 10, 6, 3, 4, 8, 1, pick up coins with a maximum value subject ***to no adjacent coins can be picked***.
- Produce the optimal solutions in terms of F array with a linear algorithm (i.e.. Use dynamic programming techniques to solve this problem)

In-class exercises - Change-making problem - 2

- Given m coin denominations: **1, 5, 6, 8,** find the minimum number of coins added up to **n = 20.** Produce the optimal solutions in terms of ***F*** array with a **O(nm)** algorithm. In addition, show the coins used for ***F(13)*** and ***F(19)*.** You may use dynamic programming technique to solve this problem, by first getting your recurrence relation down, then using tabulation to compute the values for the requested cells

In-class exercises - Coin-collecting problem - 3

	1	2	3	4	5	6	7	8
1	1				9			3
2		2		3			10	
3	8		3	2		7		8
4			5			2		
5	9			3	4			6
6			11				5	

Given the following matrix C where *C(i,j)* represents the coin value at cell *(i,j)*, compute the optimal solution matrix *F*, where *F(i,j)* stores the largest coin values collected at place *(i,j)*. show the coins picked up at cell *(6,8)*