In-Class Evaluation

21_AIE_212

Introduction to Computer Networks– SEM-IV

Professor – Vinith R

Submitted By: Vikhyat Bansal [CB.EN.U4AIE21076]

AMRITA VISHWA VIDYAPEETHAM

DEEMED TO BE UNIVERSITY

EXAMPLE 1 *Coin-row problem* There is a row of n coins whose values are some positive integers $c1, c2, \ldots, cn$, 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.

Bottom-Up Approach

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 that include the last coin and those without it.
The largest amount we can get from the first group is equal to
$cn + F(n-2)$—the value of the nth coin plus the maximum amount we can pick up from the first $n-2$ coins. The maximum amount we can get from the second group is equal to $F(n-1)$ by the definition of $F(n)$.
Thus, we have the following recurrence subject to the obvious initial conditions:

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

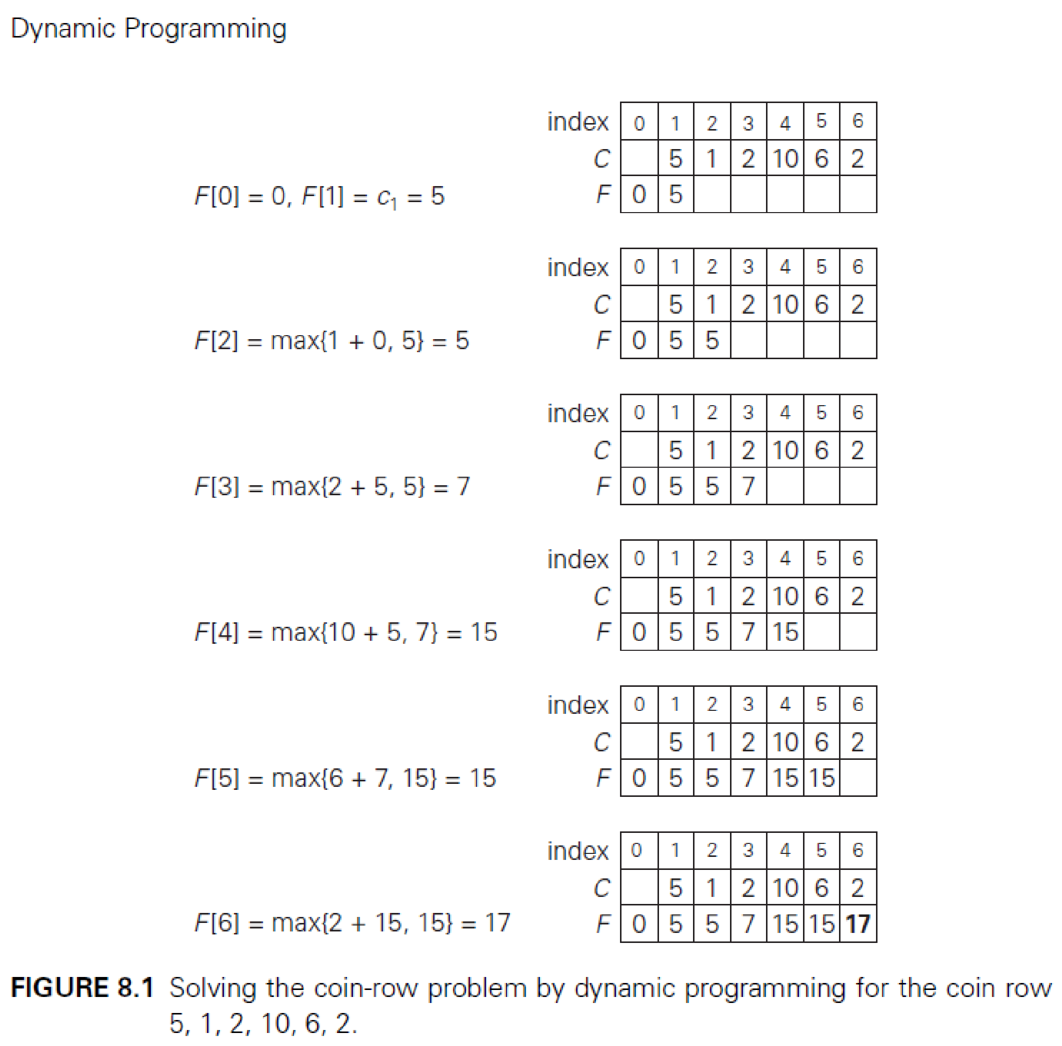
$$F(0) = 0, F(1) = c1$$

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

$F[0] \leftarrow 0$; $F[1] \leftarrow C[1]$
for $i \leftarrow 2$ to $n$ do
$F[i] \leftarrow \max(C[i] + F[i-2], F[i-1])$
return $F[n]$
```

Explanation with an example:

The application of the algorithm to the coin row of denominations 5, 1, 2, 10,6, 2 is shown in Figure 8.1


Dynamic Programming

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

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**

FIGURE 8.1 Solving the coin-row problem by dynamic programming for the coin row 5, 1, 2, 10, 6, 2.

To find the coins with the maximum total value found, we need to backtrace the computations to see which of the two possibilities—$c_n + F(n-2)$ or $F(n-1)$—produced the maxima in formula (8.3).

In the last application of the formula, it was the sum $c6 + F(4)$, which means that the coin $c6 = 2$ is a part of an optimal solution. Moving to computing $F(4)$, the maximum was produced by the sum $c4 + F(2)$, which means that the coin $c4 = 10$ is a part of an optimal solution as well.

Finally, the maximum in computing $F(2)$ was produced by $F(1)$, implying that the coin $c2$ is not the part of an optimal solution and the coin $c1 = 5$ is.

Thus, the optimal solution is $\{c1, c4, c6\}$. To avoid repeating the same computations during the backtracing, the information about which of the two terms in (8.3) was larger can be recorded in an extra array when the values of F are computed.

Top Down with Memoization Approach

To find the maximum value, what can be done is just divide the array into two parts in such a way that when number of integer are more than 2 in the array, there will be two totals that will be calculated. Firstly, the total will include the first element and will miss the next element and the second total will contain the second element of the array where it misses out on second element and after getting both the totals, we will get the max(total1,total2).

ALGORITHM *CoinRow(C[1..n])*
//Applies formula top down 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] ← 0; F[1] ← C[1]
F[2] ← return max(C[1],C[2])

First_total = C[1] + CoinRow(C[3....n])
Second_total = C[2] + CoinRow(C[4.....n])
Return(max(First_total,Second_total))
```

Explanation with example:

The application of the algorithm to the coin row of denominations 5, 1, 2,10,6, 2 is shown in Figure 8.1.

Problem has been broken down into a seq. of pairs where we treat the list as binary tree [5, 1, 2, 10,6, 2]

```
      First total                    Second total

           5                               1
          / \                             /   \
        2    10                         10     6
       / \     \                       /  \   / \
      6   2     2                     2   -   -  -
```

EXAMPLE 2 *Change-making problem* Consider the general instance of the following well-known problem. Give change for amount n using the minimum number of coins of denominations $d1<d2 < . . .<dm$. For the coin denominations used in the United States, as for those used in most if not all other countries, there is a very simple and efficient algorithm discussed in the next chapter. Here, we consider a dynamic programming algorithm for the general case, assuming availability of unlimited quantities of coins for each of the m denominations $d1< d2 < . . . < dm$ where $d1 = 1$.

Bottom-Up Approach

Let $F(n)$ be the minimum number of coins whose values add up to n; it is convenient to define $F(0) = 0$. The amount n can only be obtained by adding one coin of denomination dj to the amount $n - dj$ for $j = 1, 2, . . . , m$ such that $n \geq dj$. Therefore, we can consider all such denominations and select the one minimizing $F(n - dj) + 1$. Since 1 is a constant, we can, of course, find the smallest $F(n - dj)$ first and then add 1 to it. Hence, we have the following recurrence for $F(n)$:

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

$F(0) = 0.$

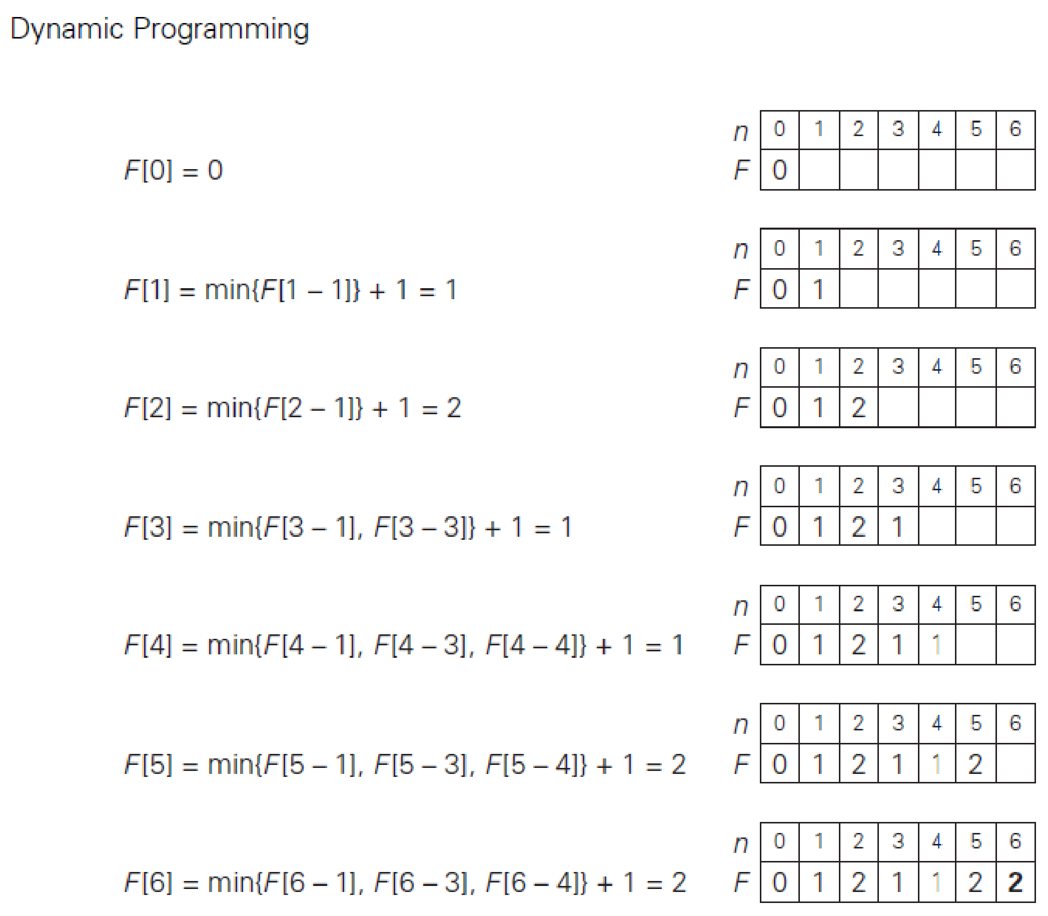
We can compute $F(n)$ by filling a one-row table left to right in the manner similar to the way it was done above for the coin-row problem, but computing a table entry here requires finding the minimum of up to m numbers.

```
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 using bottom up approach
//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]
```

Explanation with example:

The application of the algorithm to amount $n = 6$ and denominations 1, 3, 4 is shown in Figure 8.2.



Dynamic Programming

$F[0] = 0$

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**

FIGURE 8.2 Application of Algorithm *MinCoinChange* to amount $n = 6$ and coin denominations 1, 3, and 4.

Find the coins of an optimal solution, we need to backtrace the computations to see which of the denominations produced the minima in formula (8.4). For the instance considered, the last application of the formula (for $n = 6$), the minimum was produced by $d2 = 3$. The second minimum (for $n = 6 - 3$) was also produced for a coin of that denomination. Thus, the minimum-coin set for $n = 6$ is two 3's.

Top-Down Approach with Memoization

CoinChangeTDA accepts four parameters: n is the amount, coins array contains the coins from which the amount must be computed, t is the total number of coins in the denomination array, and dp array stores all previously calculated possible combinations in the similar manner as shown in Figure 8.2.
Return 0 if the quantity is 0. If the value has already been calculated, the dp array should return the result. If the value cannot be calculated, recursively invoke the CoinChangeTopDownApproach.

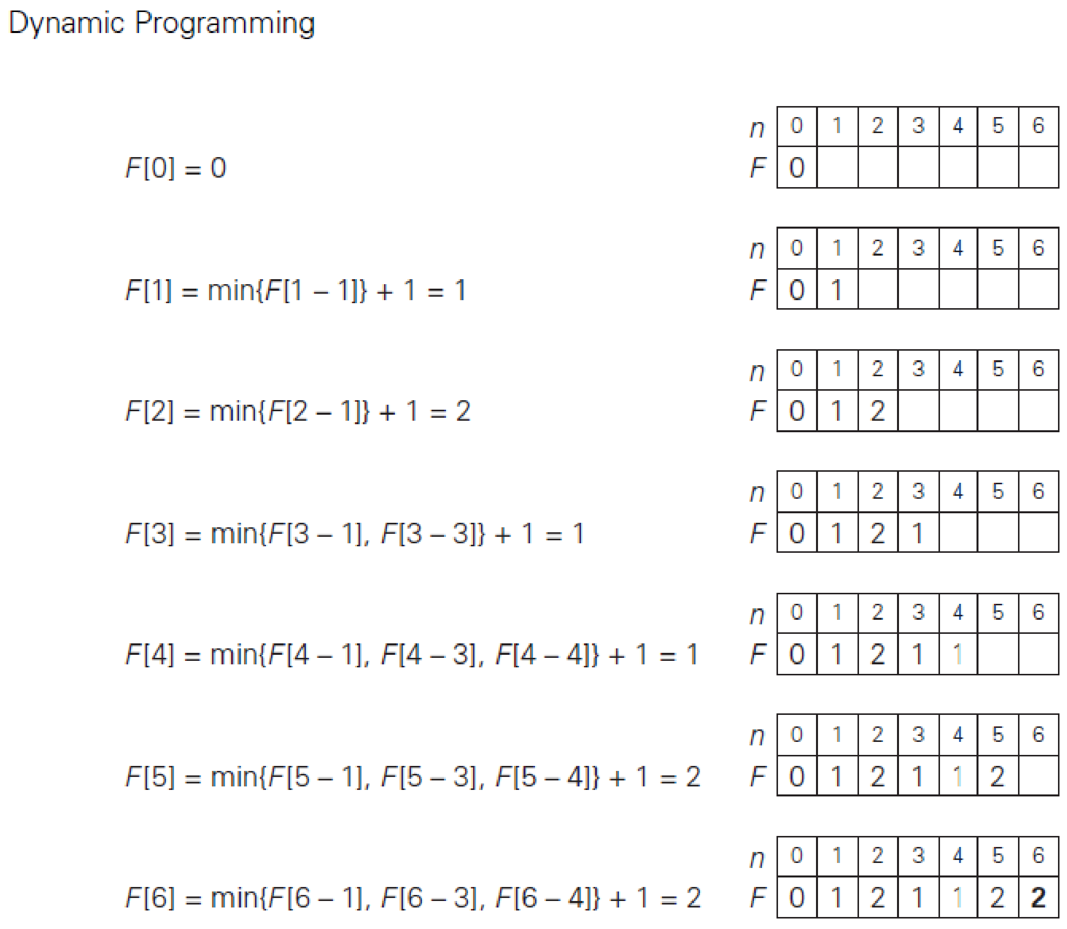
```
ALGORITHM CoinChangeTDA(n,D[1..m],t,dp[])
//Applies dynamic programming to find the minimum number of coins
//of denominations d1< d2 < . . . < dm where d1 = 1 that add up to a
//given amount n using top down approach
//Input: Positive integer n and array D[1..m] of increasing positive
// integers indicating the coin denominations where D[1]= 1
//Output: The minimum number of coins that add up to n

If (n==0) <- return 0
If (dp[n] != 0) <- return dp[n]
ans = infinite
for (i <- 0 to t-1 ) do
        if (n – D[i] > = 0)
                count = CoinChangeTDA(n-D[i] , D[1..m] , t, dp)
                ans = Min(ans,count + 1)
dp[n] = ans
return dp[n]
```

Explanation with example:

To find the minimum amount of coin denomination required for which the amount is completed can be calculated by recursively calling the function and storing the value at each step in an array and then accessing the stored value in the array to get the final output.

Lets take for example D = [1,3,4] , N = 6, t = D.length, dp = int.array[N+1]

In this case n >= 0 and we are given with D = [1,3,4], what will be done is initially an integer ANS is initiated with some infinite value and then a loop is run where value of I should be less then the length of Denomination array and an IF condition is passed where (n-coins[i] >= 0) where a recursive call is being passed where in an additional array the number of minimum coins of different denomination at different value of n will be stored.



Dynamic Programming

$F[0] = 0$

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**

FIGURE 8.2 Application of Algorithm *MinCoinChange* to amount $n = 6$ and coin denominations 1, 3, and 4.

EXAMPLE 3 *Coin-collecting 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. Design an algorithm to find the maximum number of coins the robot can collect and a path it needs to follow to do this.

Bottom-Up Approach

Let $F(i, j)$ be the largest number of coins the robot can collect and bring to the cell (i, j) in the ith row and jth column of the board. 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.
There are no adjacent cells to the left in the first column and similarly there are no cell above the first row, for those we assume that the F is 0.
Therefore, the largest number of coins the robot can bring to cell (i, j) is the maximum of these two numbers plus one possible coin at cell (i, j) itself.

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

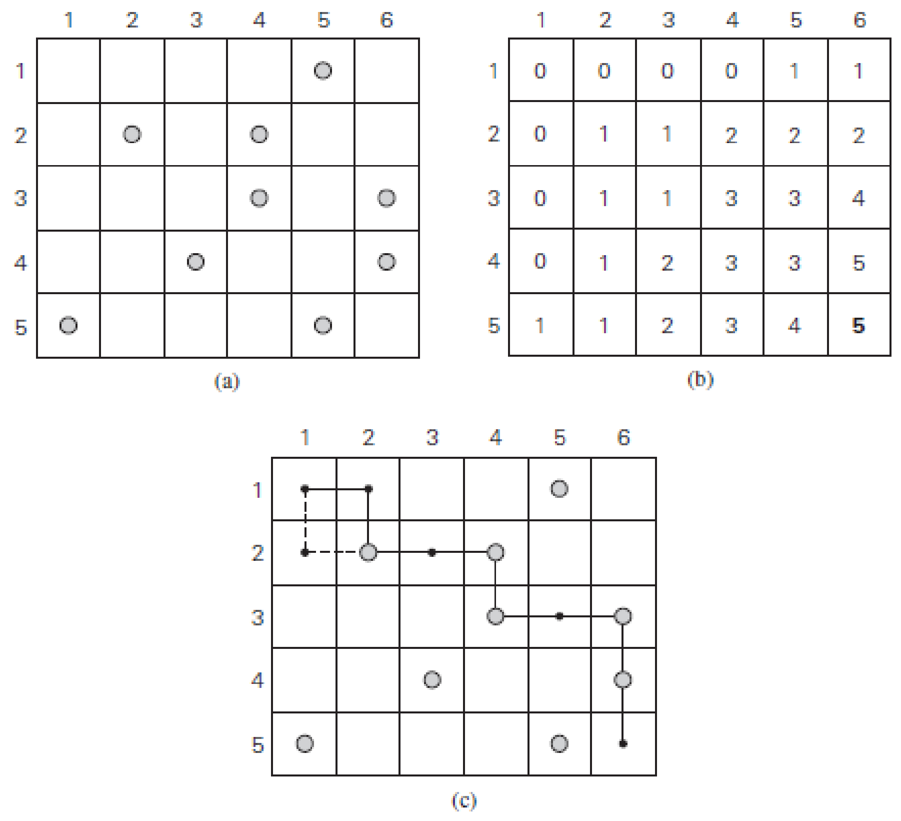
$F(0, j) = 0$ for $1 \le j \le m$ and $F(i, 0) = 0$ for $1 \le i \le n$,
where cij
$= 1$ if there is a coin in cell (i, j), and cij
$= 0$ otherwise.

```
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]
```

Explanation with example:

Dynamic Programming



	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**

FIGURE 8.3 (a) Coins to collect. (b) Dynamic programming algorithm results. (c) Two paths to collect 5 coins, the maximum number of coins possible.

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

From the picture above, it can be noticed that when starting from (1,1) there is no coin, now after first step the robot can move down or right as there is no coin in both cells.

After first step, while going for 2nd step the robot can move either down or right, it can be seen that when robot moves down there will be one coin that can be accessed by it and because of that there will +1 in the count.

Similar process will be repeated by the help of FOR loop until the maximum amount coins that can be collected are calculated and the output will be returned.

<u>Top-Down Approach with Memoization</u>

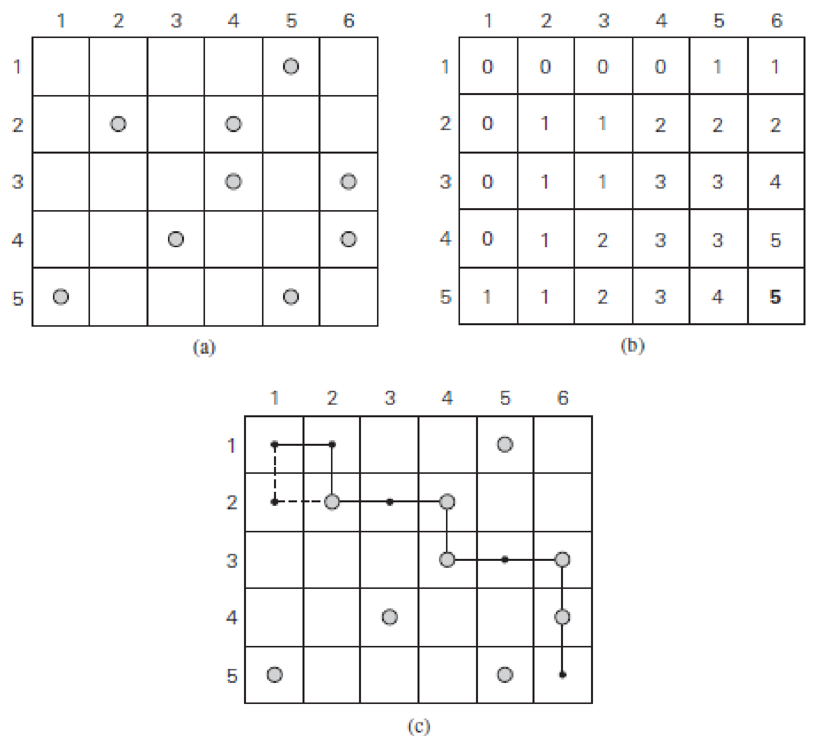
In a simple manner, one can create a matrix which will update each cell with +1 as robot goes on collecting the coins. Now to do same the input matrix is traversed and recursive call takes place in such a manner that coins are being collected by the robot and the max coin matrix is updated by removing each row and each column as a possibility as it moves on.

```
ALGORITHM RobotCoinCollection(n,m,C[1….r,1…c])
//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 $(n, m)$


// New zero matrix is created with n+1 and m+1
```

```
Max_coin = [[0]*(n+1) for f in range (m+1)]
for i←1 to n+1
    for j← 1 to m+1
        Max_coin[i][j] = max(RobotCoinCollection[i-1][j],RobotCoinCollection[i][j-1]) + C[n-1][m-1]
Return Max_coin[n][m]
```

Explanation with example

Dynamic Programming



	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**

FIGURE 8.3 (a) Coins to collect. (b) Dynamic programming algorithm results. (c) Two paths to collect 5 coins, the maximum number of coins possible.

In layman terms, its just simple matrix updation taking place because of recursion and at final step return of Max_Coin last element will be given as last element of that matrix is the max number of coins collected as seen from above picture.

THANK YOU!!