13 - Dynamic programming

[KOMS119602] & [KOMS120403]

Design and Analysis of Algorithm (2021/2022)

Dewi Sintiari

Prodi S1 Ilmu Komputer
Universitas Pendidikan Ganesha

Week 21-25 March 2022

Table of contents

Figure: Richard Ernest Bellman (1920-1984)

Principal of dynamic programming (1)

- Programming does not refer to computer programming, it means "planning"
- Dynamic is used to capture the time-varying aspect of the problems

Dynamic programming is a problem solving method by outlining the solution into a set of *stages*, such that the solution of the problem can be seen as **a sequence of decisions**.

Principal of dynamic programming (2)

- Usually applied for *optimization problems* (maximization/minimization).
- Typically, these subproblems arise from a **recurrence** relating a given problems solution to solutions of its smaller subproblems.
- Rather than solving overlapping subproblems again and again, dynamic programming suggests solving each of the smaller subproblems only once and recording the results in a table from which a solution to the original problem can then be obtained.

Fibonacci numbers revisited

Recall that Fibonacci sequence is defined as follows.

$$F(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2 \\ F(n-1) + F(n-2), & n \geq 3 \end{cases}$$

Fibonacci sequence: $1, 1, 2, 3, 5, 8, 13, 21, \ldots$

Recursive algorithm:

Algorithm 1 Fibonacci sequence recursively

```
1: procedure FIB(n)
2:     if n ≤ 2 then return 1
3:     end if
4:     return (FIB(n − 1) + FIB(n − 2))
5: end procedure
```

Fibonacci numbers revisited

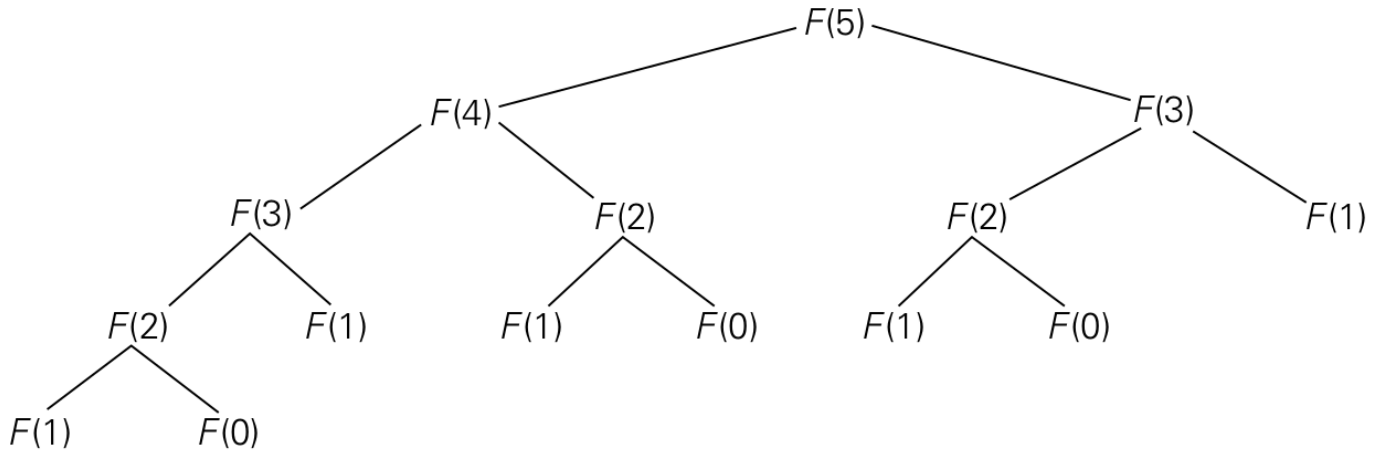


Figure: Tree of recursive calls for computing the 5th Fibonacci number by the definition-based algorithm.

Fibonacci numbers revisited

How to handle **overlapping computations**?

- Create a 1-dim array, and fill with the $n+1$ consecutive values of $F(n)$, starting from $F(0)$, and the last element will be $F(n)$

Algorithm 2 Fibonacci sequence iteratively

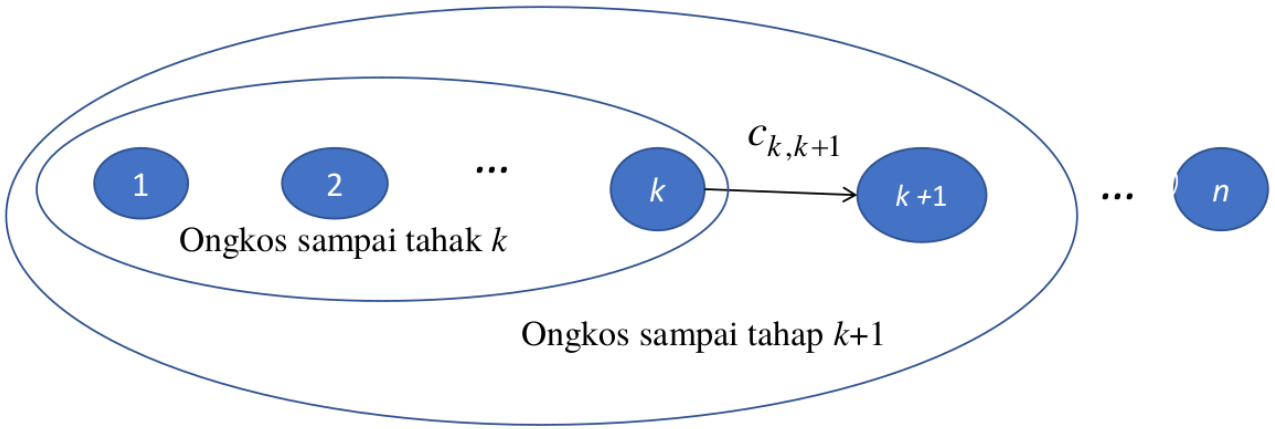
```
1: procedure FIB2(n)
2:     F[0] ← 0;  F[1] ← 1
3:     for i ← 2 to n do
4:         F(i) ← F(i − 1) + F(i − 2)
5:     end for
6:     return F[n]
7: end procedure
```

Principle of optimality

A problem is said to satisfy the Principle of Optimality **if an optimal solution to any instance of an optimization problem is composed of optimal solutions to its sub-instances**; i.e. if the total solution is optimal, then the part of solution up to step k is also optimal.

- Implication: if we work from step k to step $k+1$, we can use the optimal solution up to step k, without going back to the initial state.

Cost at step $k+1$ = cost at step k + cost from step k to step $k+1$



Requirements

Generally, **there are several requirements to apply dynamic programming to a problem** [Algorithm Design, Kleinberg and Tardos]:

1. Solution to the original problem can be computed from solutions to (independent) subproblems.
2. There are polynomial number of subproblems.
3. There is an ordering of the subproblems such that the solution to a subproblem depends only on solutions to subproblems that precede it in this order.

Dynamic Programming Steps

1. **Define subproblems:** This is the critical step. Usually the recurrence structure follows naturally after defining the subproblems.
2. **Recurrence:** Write solution to (sub)problem in forms of solutions to smaller subproblems, i.e., recursion. This will give the algorithm.
3. **Correctness:** Prove the recurrence is correct, usually by *induction*.
4. **Complexity:** Analyze the runtime complexity. Usually:

$$runtime = \#subproblems \times timeToSolveEachOne$$

Two approaches

1. **Forward/top-down** approach

   Calculations are carried out from stages $1, 2, \ldots, n-1, n$.
   Sequence of the decision variables: $x_1, x_2, \ldots, x_n$.

2. **Backward/bottom-up** approach

   Calculations are carried out from stages $n, n-1, \ldots, 2, 1$.
   Sequence of the decision variables: $x_n, x_{n-1}, \ldots, x_1$.

Some basic examples

1. Coin-row problem
2. Change-making problem
3. Coin-collecting problem

Coin-row problem

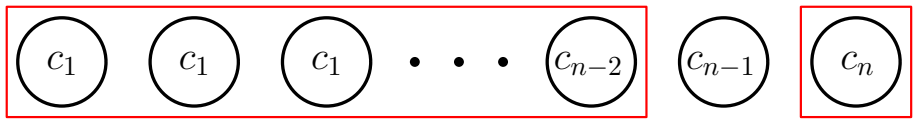
Coin-row problem (1)

Problem

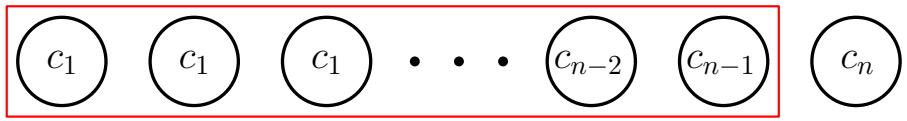
There is a row of n coins whose values are some positive integers $c_1, c_2, \ldots, c_n$, not necessarily distinct. The goal is to pick up the maximum amount of money subject to the constraint that no two coins adjacent in the initial row can be picked up.

Strategy: Let $F(n)$: the maximum amount that can be picked up from the row of n coins. How to derive a recursive formula for $F(n)$?

- Two partitions of allowed coin selections:
  1. Group that includes the last coin
  2. Group that does not include the last coin



$$c_n + F(n-2)$$



$$F(n-1)$$

Coin-row problem (2)

The recursive function:

$$\begin{cases} F(n) & = \max\{c_n + F(n-2), F(n-1)\} \text{ for } n > 1 \\ F(0) & = 0, \ F(1) = c_1 \end{cases}$$

So, $F(n)$ can be computed as in Fibonacci sequence.

Algorithm 3 Coin row

```
1: procedure CoinRow(C[1..n])
2:     F[0] ← 0;  F[1] ← C[1]
3:     for i ← 2 to n do
4:         F(i) ← max{C[i] + F(i − 2), F(i − 1)}
5:     end for
6:     return F[n]
7: end procedure
```

Coin-row problem (3)

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

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: Solving the coin-row problem by dynamic programming for the coin row 5, 1, 2, 10, 6, 2, with optimal solution $\{c_1, c_4, c_6\}$ and optimal value 17.

Change-making problem

Change-making problem (1)

Problem

Give change for amount n using the minimum number of coins of denominations $d_1 < d_2 < \cdots < d_m$. Assume that we have availability of unlimited quantities of coins for each of the m denominations $d_1 < d_2 < \cdots < d_m$ where $d_1 = 1$.

Strategy: Let $F(n)$: the minimum number of coins whose values add up to n, and define $F(0) = 0$. How to derive a recursive formula for $F(n)$?

- The amount n can only be obtained by adding one coin of denomination d_j to the amount $n - d_j$ for $j = 1, 2, \ldots, m$ such that $n \geq d_j$.
- Minimizing $F(n - d_j) + 1$

The recursive function:

$$\begin{cases} F(n) & = \min_{j; n \geq d_j} F(n - d_j) + 1 \text{ for } n > 0 \\ F(0) & = 0 \end{cases}$$

Change-making problem (2)

Algorithm 4 ChangeMaking

```
1:  procedure MinCoinChange(D[1..m], n)
2:      input: positive integer n, and array D[1..m] of increasing positive integers indicating the coin denominations where D[1] = 1
3:      output: the minimum number of coins that add up to n
4:      F[0] ← 0
5:      for i ← 1 to n do
6:          temp ← ∞;  j ← 1
7:          while j ≤ m and i ≥ D[j] do
8:              temp ← min(F[i − D[j]], temp)
9:              j ← j + 1
10:         end while
11:         F[i] ← temp + 1
12:     end for
13:     return F[n]
14: end procedure
```

Change-making problem (3)

$F[0] = 0$

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

Coin-collecting problem

Coin-collecting problem (1)

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.

Strategy: 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.

- (i, j) can be reached from cell $(i - 1, j)$ (above) or cell $(i, j - 1)$ (left).
- For cell in the first row (resp. first column), assume that $F(i - 1, j) = 0$ (resp. $F(i, j - 1) = 0$).

Coin-collecting problem (2)

The recursive function:

$$\begin{cases} F(i,j) & = \max\{F(i-1,j), F(i,j-1)\} + c_{ij} \text{ for } 1 \le i < n,\ 1 \le j \le m \\ F(0,j) & = 0 \text{ for } 1 \le j \le m \text{ and } F(i,0) \text{ for } 1 \le i \le n \end{cases}$$

Algorithm 5 Robot coin collection

```
1:  procedure RobotCoinCollection(C[1..n, 1..m])
2:      F[1,1] ← C[1,1]
3:      for j ← 2 to m do
4:          F[1,j] ← F[1,j−1] + C[1,j]
5:      end for
6:      for i ← 2 to n do
7:          F[i,1] ← F[i−1,1] + C[i,1]
8:          for j ← 2 to m do
9:              F[i,j] ← max{F[i−1,j], F[i,j−1]} + C[i,j]
10:         end for
11:     end for
12:     return F[n,m]
13: end procedure
```

Coin-collecting problem (3)

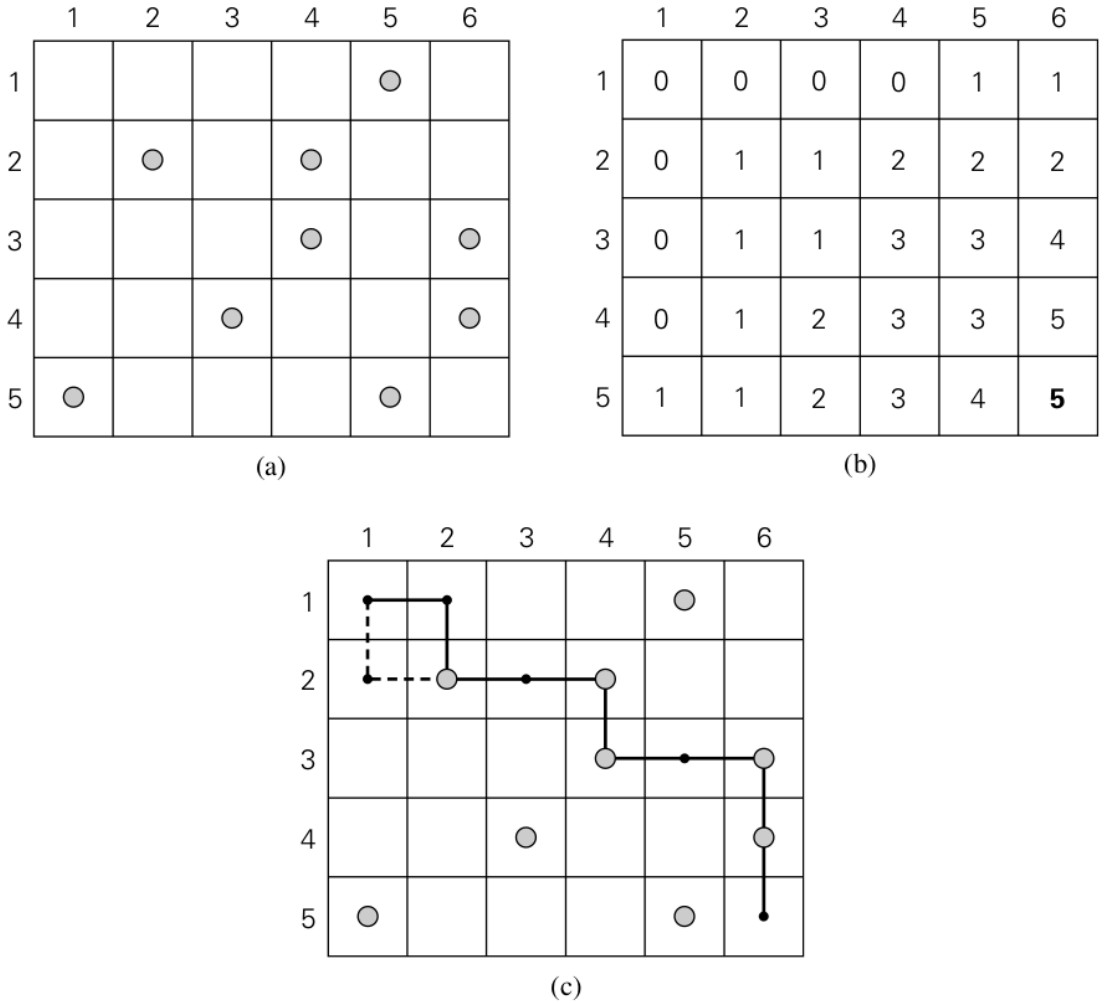


Figure: (a) Coins to collect. (b) Dynamic programming algorithm results. (c) Two paths to collect 5 coins, the maximum number of coins possible.

The knapsack problem

(dynamic-programming approach)

Knapsack problem

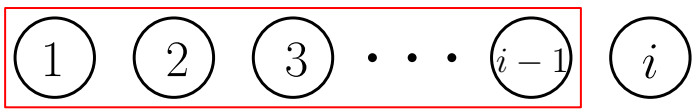
Problem

Given n items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity W. Find the most valuable subset of the items that fit into the knapsack.

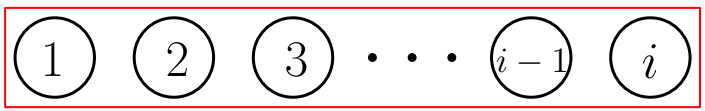
Strategy: Let $F(i, j)$ be the value of an optimal solution to instance defined by the first i items $1 \leq i \leq n$,

- with weights $w_1, \ldots, w_i$ and values $v_1, \ldots, v_i$
- knapsack capacity j, with $1 \leq j \leq W$.

We consider whether the item i is included/not.



$F(i-1, j)$



$v_i + F(i-1, j-w_i)$

Knapsack problem

$$F(i,j) = \begin{cases} \max\{F(i-1,j), v_i + F(i-1,j-w_i)\} & \text{if } j - w_i \geq 0 \\ F(i-1,j), & \text{if } j - w_i < 0 \end{cases}$$

with initial conditions:

$$F(0,j) = 0 \text{ for } j \geq 0 \text{ and } F(i,0) = 0 \text{ for } i \geq 0$$

Goal: to find $F(n,W)$, the maximal value of a subset of the n given items that fit into the knapsack of capacity W, and an optimal subset.

		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1, j-w_i)$	$F(i-1, j)$	
w_i, v_i	i	0		$F(i, j)$	
	n	0			goal

Figure: Table for solving the knapsack problem by dynamic programming.

Example

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3,\ v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2,\ v_4 = 15$	4	0	10	15	25	30	**37**

Figure: Example of solving an instance of the knapsack problem by the dynamic programming algorithm.

Memory functions

(study case in the Knapsack problem)

Memory functions principal

Dynamic programming solves problems that have a recurrence relation.

- Using the recurrences directly in a recursive algorithm has the disadvantage that it solves common sub problem multiple times, yielding exponential complexity.
- The dynamic programming technique is has the disadvantage that some of the sub-problems may not have been necessary to solve.

Our goal: to have the best of both approaches, i.e. all the necessary sub-problem solved only once.

Memory functions principal

- The technique uses a top-down approach, recursive algorithm, with table of sub-problem solution.
- Before determining the solution recursively, the algorithm checks if the sub problem has already been solved by checking the table.
- If the table has a valid value then the algorithm uses the table value else it proceeds with the recursive solution.

Recall the definition of $F(i, j)$, the value of an optimal solution to instance defined by the first i items with knapsack capacity j.

$$F(i,j) = \begin{cases} \max\{F(i-1,j), v_i + F(i-1, j-w_i)\} & \text{if } j - w_i \geq 0 \\ F(i-1,j), & \text{if } j - w_i < 0 \end{cases}$$

Algorithm 6 MF Knapsack

1: **procedure** $\mathrm{MFK}(i, j)$
2: **input**: $i \in \mathbb{Z}_{\geq 0}$ indicating the number of the first items being considered and $j \in \mathbb{Z}_{\geq 0}$ indicating the knapsack capacity
3: **output**: The value of an optimal feasible subset of the first i items
4: **variables**: global variables input arrays weights Wt[1..n], values V[1..n], and table $F[0..n, 0..W]$ whose entries are initialized with -1s except for row 0 and column 0 initialized with 0s.

5: **if** $F[i, j] < 0$ **then**
6: **if** $j <$ Wt[i] **then**
7: value $\leftarrow \mathrm{MFK}(i-1, j)$
8: **else**
9: value $\leftarrow \max\{\mathrm{MFK}(i-1, j), \mathrm{V}[i] + \mathrm{MFK}(i-1, j - \mathrm{Wt}[i])\}$
10: **end if**
11: $F[i, j] \leftarrow$ value
12: **end if**
13: **return** $F[i, j]$
14: **end procedure**

Memory functions example

Let's implement the MFK procedure on the previous example:

item	weight	value
1	2	\$12
2	1	\$10
3	3	\$20
4	2	\$15

capacity $W = 5$

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2,\ v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1,\ v_2 = 10$	2	0	—	12	22	—	22
$w_3 = 3,\ v_3 = 20$	3	0	—	—	22	—	32
$w_4 = 2,\ v_4 = 15$	4	0	—	—	—	—	**37**

Figure: Example of solving an instance of the knapsack problem by the memory function algorithm.

Memory functions example

The step-by-step implementation of the algorithm is as follows:

- $N = 4, W = 5, w_1 = 2, w_2 = 1, w_3 = 3, w_4 = 2, v_1 = 12, v_2 = 10, v_3 = 20, v_4 = 15$
- $V[4,5] = \max\{V[3,5], 15 + V[3,3]\}$ (since $v_4 = 15, w_4 = 2, j = 15, w_4 < j \rightarrow$ apply 1st case)
- $V[3,5] = \max\{V[2,5], 20 + V[2,2]\}$
- $V[3,3] = \max\{V[2,3], 20 + V[2,0]\}$
- $V[2,5] = \max\{V[1,5], 10 + V[1,4]\}$
- $V[2,2] = \max\{V[1,2], 10 + V[1,1]\}$
- $V[2,3] = \max\{V[1,3], 10 + V[1,2]\}$
- $V[2,0] = V[1,0] = 0$ (since $w_2 = 1, j = 0, w_2 > j \rightarrow$ apply 2nd case)
- $V[1,5] = \max\{V[0,5], 12 + V[0,3]\} = \max\{0, 12 + 0\} = 12$
- $V[1,4] = \max\{V[0,4], 12 + V[0,2]\} = \max\{0, 12 + 0\} = 12$
- $V[1,2] = \max\{V[0,2], 12 + V[0,0]\} = \max\{0, 12 + 0\} = 12$
- $V[1,1] = V[0,1] = 0$
- $V[1,3] = \max\{V[0,3], 12 + V[0,1]\} = \max\{0, 12 + 0\} = 12$

Now, substitute backwards:

- $V[2,3] = \max\{V[1,3], 10 + V[1,2]\} = \max\{12, 10 + 12\} = 22$
- $V[2,2] = \max\{V[1,2], 10 + V[1,1]\} = \max\{12, 10 + 0\} = 12$
- $V[2,5] = \max\{V[1,5], 10 + V[1,4]\} = \max\{12, 10 + 12\} = 22$
- $V[3,3] = \max\{V[2,3], 20 + V[2,0]\} = \max\{22, 20 + 0\} = 22$
- $V[3,5] = \max\{V[2,5], 20 + V[2,2]\} = \max\{22, 20 + 12\} = 32$
- $V[4,5] = \max\{V[3,5], 15 + V[3,3]\} = \max\{32, 15 + 22\} = 37$