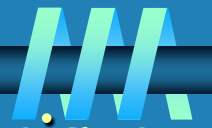
Dynamic Programming



D*ynamic Programming* is 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"**

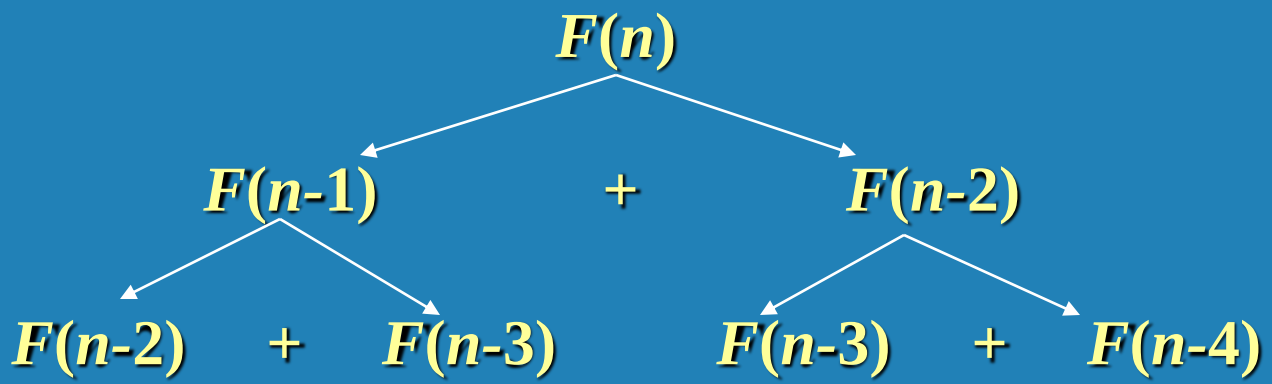
- **Main idea:**
  - **set up a recurrence relating a solution to a larger instance to solutions of some smaller instances**
  - **solve smaller instances once**
  - **Memoization (technical term for)**
    - **record sub-solutions in a table (forming a memo)**
    - **reuse recorded solutions without recomputing**

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):**



Example 1: Fibonacci numbers (cont.)

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

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

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

Efficiency:
- **time**
- **space**

Example 2: Coin-row 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.

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 ?

Thus we have the following recurrence

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

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

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

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

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

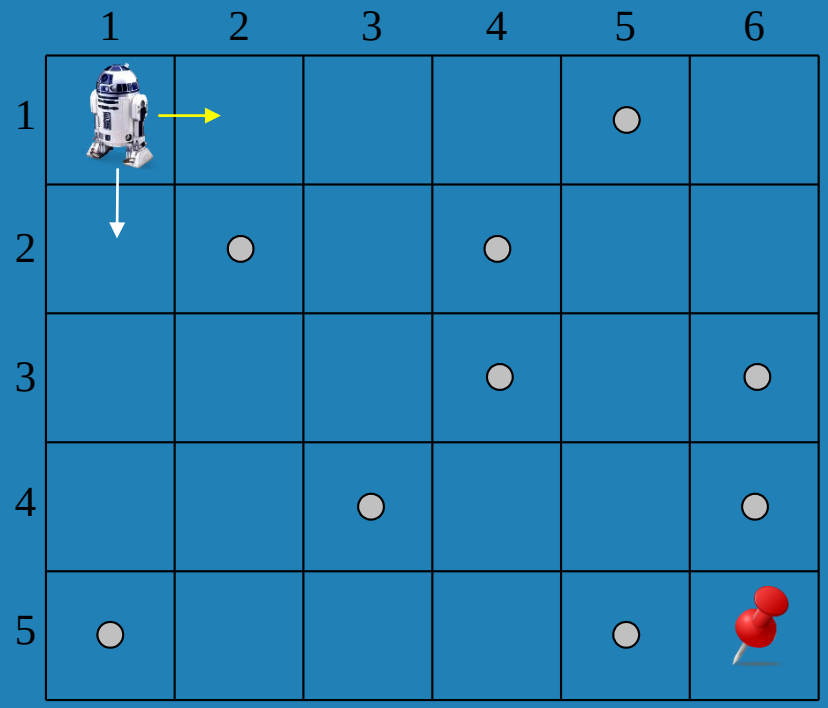
index	0	1	2	3	4	5	6
coins	--	5	1	2	10	6	2
F( )	0	5	5	7	15	15	17

Max amount: 17
Coins of optimal solution: c1, c4, c6
Efficiency:
- time : there is only n-1 iteration so, it is linear. Θ(n)
- space: Θ(n) space

Note: All smaller instances were solved.

Example 3: 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 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:

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

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

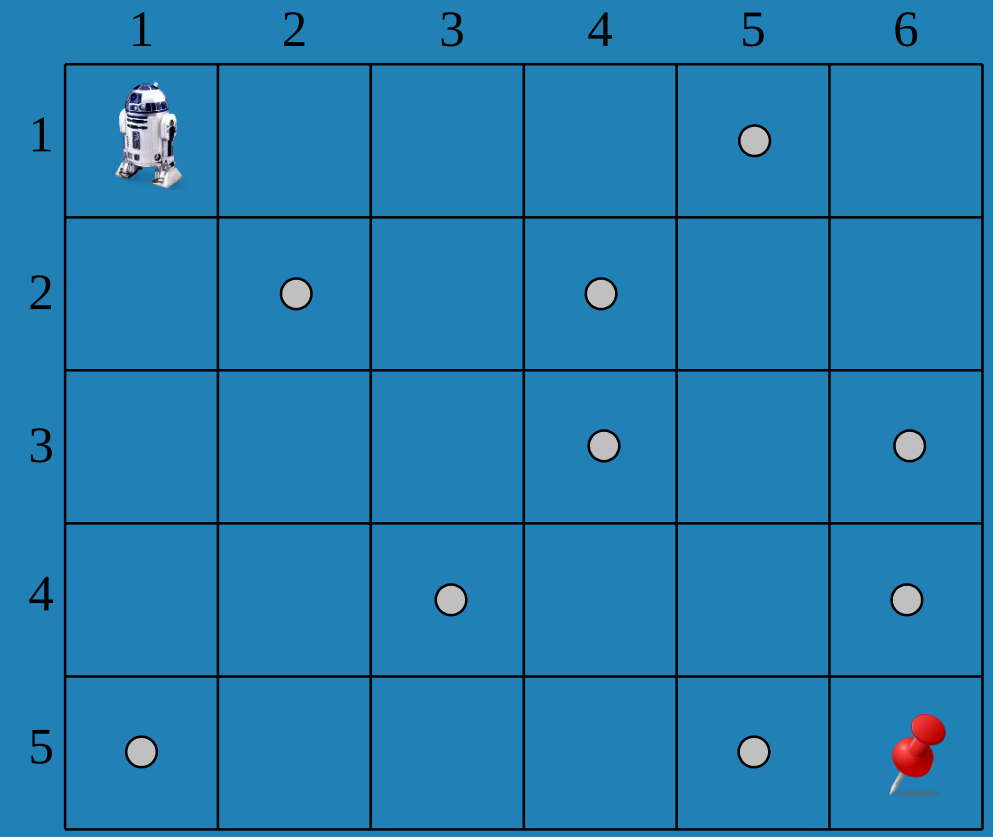
$$F(0, j) = 0 \text{ for } 1 \le j \le m \quad \text{and } F(i, 0) = 0 \text{ for } 1 \le i \le n.$$

Solution to the coin-collecting problem (cont.)

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

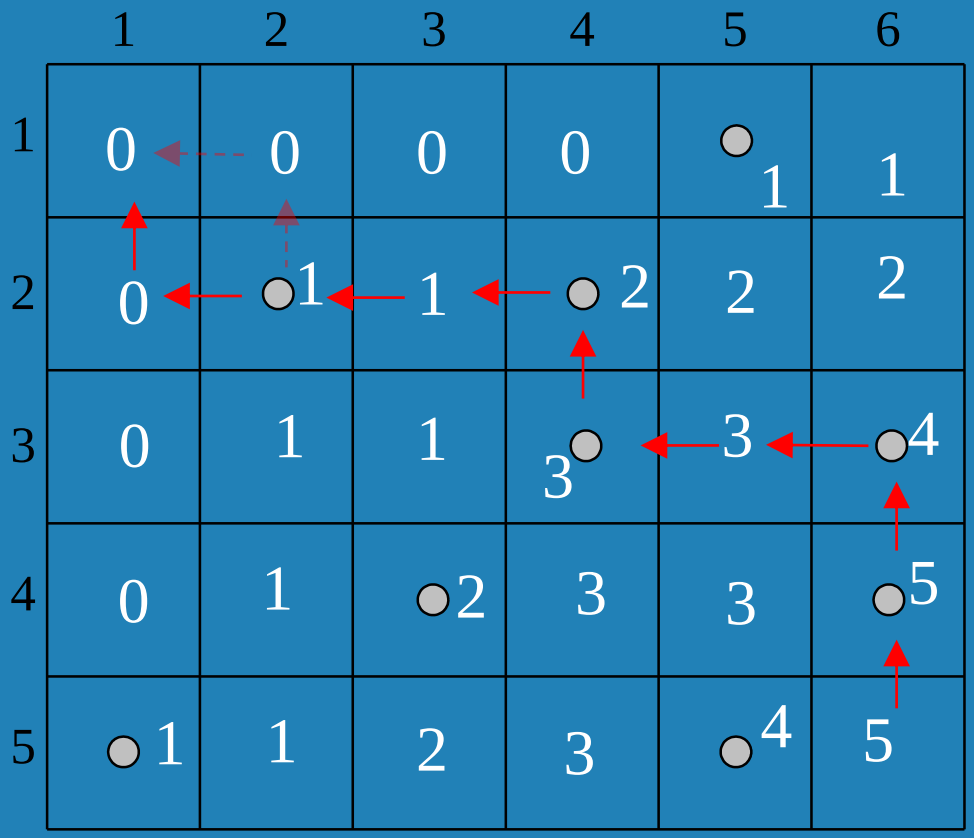
	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	5

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

```
//Applies dynamic programming to compute the largest number of
//coins a robot can collect on an n × 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]
```

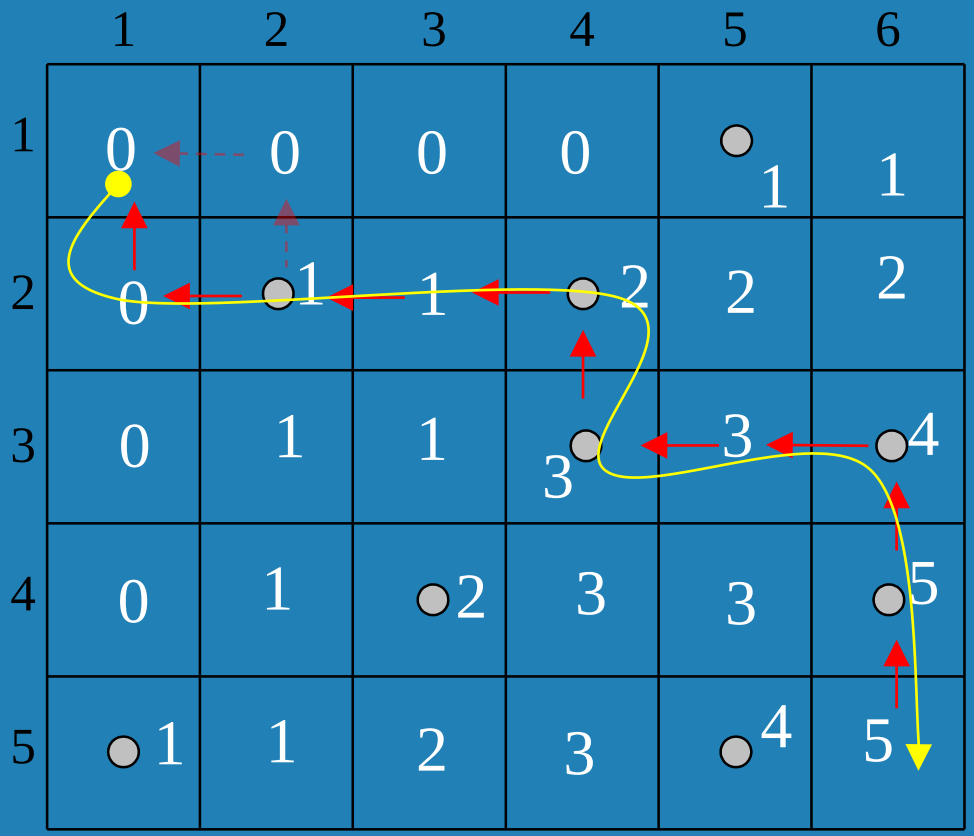
Find an Optimal Path

Tracing the computations backward makes it possible to get an optimal path. If the cells from left and top are equal, it yields two optimal paths, for instance, cell(2,2).



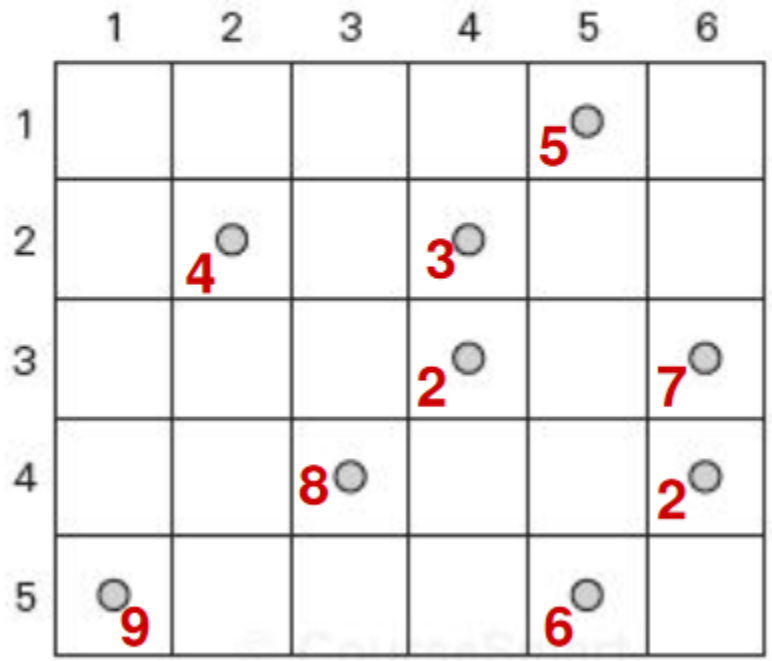
Find an Optimal Path

Tracing the computations backward makes it possible to get an optimal path. If the cells from left and top are equal, it yields two optimal paths, for instance, cell(2,2).



coin-collecting problem

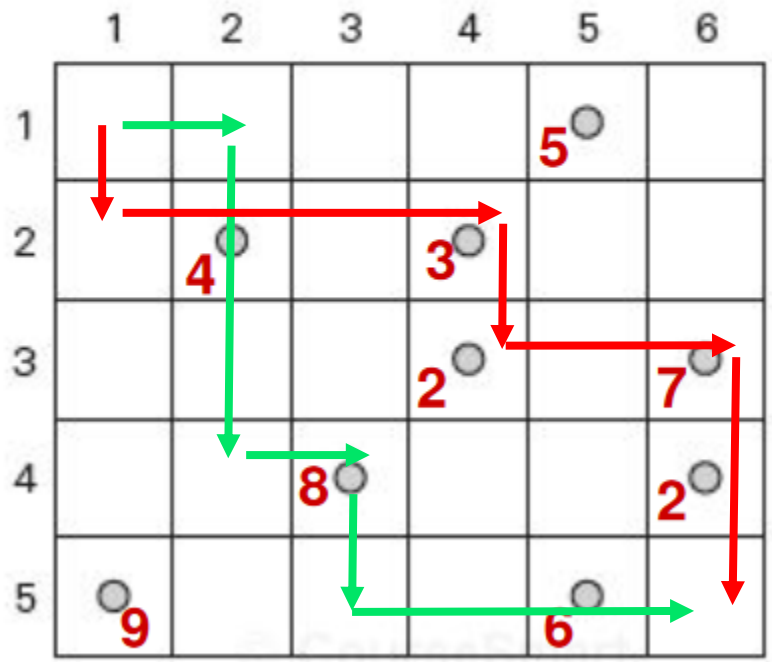
Coin-Collecting Problem: Ex-1



	1	2	3	4	5	6
1	0	0	0	0	5	5
2	0	4	4	7	7	7
3	0	4	4	9	9	16
4	0	4	12	12	12	18
5	9	9	12	12	18	18

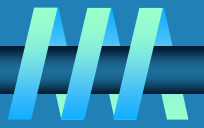
coin-collecting problem

Coin-Collecting Problem: Ex-1



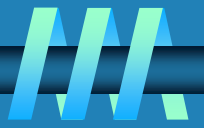
	1	2	3	4	5	6
1	0	0	0	0	5	5
2	0	4	4	7	7	7
3	0	4	4	9	9	16
4	0	4	12	12	12	18
5	9	9	12	12	18	18

Other examples of DP algorithms



- Computing a binomial coefficient (# 9, Exercises 8.1)
- General case of the change making problem (Sec. 8.1)
- Some difficult discrete optimization problems:
  - knapsack (Sec. 8.2)
  - traveling salesman
- Constructing an optimal binary search tree (Sec. 8.3)
- Warshall's algorithm for transitive closure (Sec. 8.4)
- Floyd's algorithm for all-pairs shortest paths (Sec. 8.4)

Knapsack Problem by DP



Given n items of

integer weights: $w_1 \quad w_2 \quad \dots \quad w_n$

values: $v_1 \quad v_2 \quad \dots \quad v_n$

a knapsack of integer capacity W

Find most valuable subset of the items that fit into the knapsack

Consider instance defined by first i items and capacity j ($j \le W$).

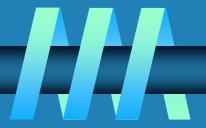
Let $F[i,j]$ be optimal value of such instance.

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

It is convenient to define the initial conditions as follows:

$$F(0, j) = 0 \text{ for } j \ge 0 \quad \text{and} \quad F(i, 0) = 0 \text{ for } i \ge 0.$$

Knapsack Problem by DP (example)



Knapsack of capacity $W = 5$

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

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

It is convenient to define the initial conditions as follows:

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

		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**

Knapsack Problem by DP (example)

- Backtracking the Table to find out the items in the optimal set.

		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

- item 4 is included in the optimal solution since the value goes up from 32 to 37. Find the items in F(4-1, 5-2) = F(3,3)

Knapsack Problem by DP (example)

- Backtracking the Table to find out the items in the optimal set.

		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**

- item 3 is not included in the optimal solution since the value (22) remains the same. Find the items in F(3-1, 3) = F(2,3)

Knapsack Problem by DP (example)

- Backtracking the Table to find out the items in the optimal set.

		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**

- item 2 is included in the optimal solution since the value changes from 12 to 22. Find the items in F(2-1, 3-1) = F(1,2)

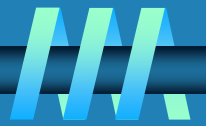
Knapsack Problem by DP (example)

- **Backtracking the Table to find out the items in the optimal set.**

		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**

- **item 1 is included in the optimal solution since the value changes from 0 to 12.**
- **Time Efficiency :**
  - **Building Table** : **θ(nW)** where n items, W capacity
  - **Composition:** **O(n)**

Memoization Solution



ALGORITHM *MFKnapsack*(i, j)

```
//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
//       items being considered and a nonnegative integer j indicating
//       the knapsack capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights[1..n], Values[1..n],
//and table F[0..n, 0..W] whose entries are initialized with −1's except for
//row 0 and column 0 initialized with 0's
if F[i, j] < 0
    if j < Weights[i]
        value ← MFKnapsack(i − 1, j)
    else
        value ← max(MFKnapsack(i − 1, j),
                    Values[i] + MFKnapsack(i − 1, j − Weights[i]))
    F[i, j] ← value
return F[i, j]
```

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

It is convenient to define the initial conditions as follows:

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

		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**