Dynamic Programming I

Dynamic Programming

- **Dynamic 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 Computer Science
- "Programming" here means "planning"

Dynamic Programming

Bellman explains the reasoning behind the term **dynamic programming** in his autobiography, **Eye of the Hurricane: An Autobiography** (1984). He explains:

“I spent the Fall quarter (of 1950) at RAND. My first task was to find a name for multistage decision processes. An interesting question is, where did the name, **dynamic programming**, come from? The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was Secretary of Defense, and he actually had a pathological fear and hatred of the word <u>research</u>. I’m not using the term lightly; I’m using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term <u>research</u> in his presence. You can imagine how he felt, then, about the term <u>mathematical</u>. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially.

Dynamic Programming

Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose? In the first place I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word "**programming**". I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying I thought, lets kill two birds with one stone. Lets take a word that has an absolutely precise meaning, namely dynamic, in the classical physical sense. It also has a very interesting property as an adjective, and that is its impossible to use the word dynamic in a pejorative sense. Try thinking of some combination that will possibly give it a pejorative meaning. It's impossible. Thus, I thought **dynamic programming** was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities."

Example: Fibonacci Numbers

Recall the definition of **Fibonacci numbers**:

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

$$F(0) = 0$$

$$F(1) = 1$$

Computing the nth Fibonacci number **recursively** (**top-down**):

$$F(n)$$

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

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

Dynamic Programming

Main idea:

a) set up a **recurrence** relating a solution to a larger instance to solutions of some smaller instances

b) solve smaller instances **once**

c) record solutions in a **table**

d) extract **solution** to the initial instance from that table

Example: Fibonacci Numbers

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

$$F(0) = 0$$

$$F(1) = 1$$

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

$$\ldots$$

$$F(n-2) =$$

$$F(n-1) =$$

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

Efficiency:

- Time:______
- Space:_____

0	1	1	$\cdots$	$F(n-2)$	$F(n-1)$	$F(n)$

Example: Fibonacci Numbers

Computing the n^{th} Fibonacci number iteratively

Algorithm Fib(n)

// Input: A nonnegative integer n

//Output: The nth Fibonacci number

$F[0] \leftarrow 0$

$F[1] \leftarrow 1$

for $i \leftarrow 2$ **to** n **do**

$F[i] \leftarrow F[i-1] + F[i-2]$

return $F[n]$

Example: Coin-Row Problem

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**.

Example: $5, \; 1, \; 2, \; 10, \; 6, \; 2.$ What is the best selection?

Example: Coin-Row Problem

DP solution: 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:

1) those **without last coin** – the max amount is?
2) those **with the last coin** – the max amount is?

Thus we have the following **recurrence**

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

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

Example: Coin-Row Problem

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

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

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

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

Example: Coin-Row Problem

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

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

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

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

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

Example: Coin-Row Problem

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

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

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

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

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

Example: Coin-Row Problem

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

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

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

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

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

Example: Coin-Row Problem

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

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

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

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

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

Example: Coin-Row Problem

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

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

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

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

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

Example: Coin-Row Problem

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

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

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

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

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

Example: Coin-Row Problem

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

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

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

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

- Coins of optimal solution: ________________?
- Time/Space efficiency: ________________?

Example: Coin-Row Problem

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

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

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

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

- Coins of optimal solution: 2, 10, 5
- Time/Space efficiency: **linear**

Knapsack Problem

Given n items of known:

- **items**: $1, 2, \dots, n$
- **weights**: $w_1, w_2, \dots, w_n$
- **values**: $v_1, v_2, \dots, v_n$
- a **knapsack** of capacity W

Problem: Find the **most valuable subset of the items** that **fit into the knapsack**

Knapsack Problem

Example: Knapsack capacity $W = 16$

ITEM	WEIGHT	VALUE
1	2	\$20
2	5	\$30
3	10	\$50
4	5	\$10

The exhaustive search approach: (1) generate **all subsets** of the set of n items; **(2)** compute the total weight of each subset; **(3)** find a subset of largest value among them.

The total # of all subsets: 2^n **Efficiency:** $\Omega(2^n)$

SUBSETS	TOTAL WEIGHT	TOTAL VALUE
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	not feasible
{1,2,4}	12	$60
{1,3,4}	17	not feasible
{2,3,4}	20	not feasible
{1,2,3,4}	22	not feasible

SUBSETS	TOTAL WEIGHT	TOTAL VALUE
{1}	2	$20
{2}	5	$30
{3}	10	$50
{4}	5	$10
{1,2}	7	$50
{1,3}	12	$70
{1,4}	7	$30
{2,3}	15	$80
{2,4}	10	$40
{3,4}	15	$60
{1,2,3}	17	**not feasible**
{1,2,4}	12	$60
{1,3,4}	17	**not feasible**
{2,3,4}	20	**not feasible**
{1,2,3,4}	22	**not feasible**

Knapsack Problem – DP Approach

Drive a **recurrence relation** expressing a solution of the knapsack problem in terms of its smaller instance:

- consider an instance defined by the first i **items**, $1 \leq \mathrm{i} \leq \mathrm{n}$, with
  - weights $\mathrm{w}_1, \mathrm{w}_2, \ldots, \mathrm{w}_\mathrm{i}$,
  - values $\mathrm{v}_1, \mathrm{v}_2, \ldots, \mathrm{v}_\mathrm{i}$,
  - knapsack capacity j, $1 \leq \mathrm{j} \leq \mathrm{W}$.

- $\mathrm{F}(\mathrm{i}, \mathrm{j})$: the value of the **optimal solution** to this instance.

Knapsack Problem – DP Approach

- $F(i, j)$: the value of the **optimal solution** to this instance – the value of the **most valuable subset** of the first i items that fit the knapsack of capacity j.

Knapsack Problem – DP Approach

- $F(i, j)$: the value of the **optimal solution** to this instance – the value of the **<u>most valuable subset</u>** of the first i items that fit the knapsack of capacity j.

All the subsets of the first **i items** that fit the knapsack of **capacity j**:

The subsets that **do not include** item i:

The subsets that **include** item i:

Knapsack Problem – DP Approach

- $F(i, j)$: the value of the **optimal solution** to this instance – the value of the **<u>most valuable subset</u>** of the first i items that fit the knapsack of capacity j.

All the subsets of the first **i items** that fit the knapsack of **capacity j**:

$F(i, j)$

The subsets that **do not include** item i:

The subsets that **include** item i:

Knapsack Problem – DP Approach

- $\mathrm{F(i,j)}$: the value of the **optimal solution** to this instance – the value of the **most valuable subset** of the first i items that fit the knapsack of capacity j.

All the subsets of the first **i items** that fit the knapsack of **capacity j**:

$$\mathrm{F(i,j)}$$

The subsets that **do not include** item i:

$$\mathrm{F(i-1,j)}$$

The subsets that **include** item i:

Knapsack Problem – DP Approach

- $F(i, j)$: the value of the **optimal solution** to this instance – the value of the **most valuable subset** of the first i items that fit the knapsack of capacity j.

All the subsets of the first **i items** that fit the knapsack of **capacity j**:

$$F(i, j)$$

The subsets that **do not include** item i:

$$F(i-1, j)$$

The subsets that **include** item i:

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

Knapsack Problem – DP Approach

$$F(i, j) =$$

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

- $F(0, j) = 0$ for $j \geq 0$ and $F(i, 0) = 0$ for $i \geq 0$

- **Goal**: to find $F(n, W)$

Knapsack Problem – DP Table

			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**

Example: Knapsack Problem

ITEM	WEIGHT	VALUE	CAPACITY
1	2	\$12	$W = 5$
2	1	\$10	
3	3	\$20	
4	2	\$15	

Dynamic Programming Table

		capacity j				
i	0	1	2	3	4	5
0						
1						
2						
3						
4						

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

Dynamic Programming Table

capacity j

i	0	1	2	3	4	5
0	0					
1						
2						
3						
4						

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

$$F(0,0) = 0$$

Dynamic Programming Table

capacity j

i	0	1	2	3	4	5
0	0	0				
1						
2						
3						
4						

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

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

Dynamic Programming Table

capacity j

i	0	1	2	3	4	5
0	0	0	0			
1						
2						
3						
4						

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

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

Dynamic Programming Table

capacity j

i	0	1	2	3	4	5
0	0	0	0	0		
1						
2						
3						
4						

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

$$F(0,3) = 0$$

Dynamic Programming Table

capacity j

i	0	1	2	3	4	5
0	0	0	0	0	0	
1						
2						
3						
4						

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

$$F(0,4) = 0$$

Dynamic Programming Table

capacity j

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

$$F(0,5) = 0$$

Dynamic Programming Table

capacity j

$w_1 = 2, v_1 = 12$

$w_2 = 1, v_2 = 10$

$w_3 = 3, v_3 = 20$

$w_4 = 2, v_4 = 15$

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2						
3						
4						

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

Dynamic Programming Table

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				
$w_2 = 1, v_2 = 10$	2						
$w_3 = 3, v_3 = 20$	3						
$w_4 = 2, v_4 = 15$	4						

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

Dynamic Programming Table

capacity j

$w_1 = 2, v_1 = 12$
$w_2 = 1, v_2 = 10$
$w_3 = 3, v_3 = 20$
$w_4 = 2, v_4 = 15$

i	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12			
2						
3						
4						

$$F(1,2) = 12$$

Dynamic Programming Table

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		
$w_2 = 1, v_2 = 10$	2						
$w_3 = 3, v_3 = 20$	3						
$w_4 = 2, v_4 = 15$	4						

$$F(1,3) = 12$$

Dynamic Programming Table

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	
$w_2 = 1, v_2 = 10$	2						
$w_3 = 3, v_3 = 20$	3						
$w_4 = 2, v_4 = 15$	4						

$$F(1,4) = 12$$

Dynamic Programming Table

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						
$w_3 = 3, v_3 = 20$	3						
$w_4 = 2, v_4 = 15$	4						

$$F(1,5) = 12$$

Dynamic Programming Table

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					
$w_3 = 3, v_3 = 20$	3						
$w_4 = 2, v_4 = 15$	4						

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

Dynamic Programming Table

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				
$w_3 = 3, v_3 = 20$	3						
$w_4 = 2, v_4 = 15$	4						

$$F(2,1) = 10$$

Dynamic Programming Table

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			
$w_3 = 3, v_3 = 20$	3						
$w_4 = 2, v_4 = 15$	4						

$$F(2,2) = 12$$

Dynamic Programming Table

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		
$w_3 = 3, v_3 = 20$ 3						
$w_4 = 2, v_4 = 15$ 4						

$$F(2,3) = 22$$

Dynamic Programming Table

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	
$w_3 = 3, v_3 = 20$	3						
$w_4 = 2, v_4 = 15$	4						

$$F(2,4) = 22$$

Dynamic Programming Table

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						
$w_4 = 2, v_4 = 15$	4						

$$F(2,5) = 22$$

Dynamic Programming Table

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					
$w_4 = 2, v_4 = 15$	4						

$$F(3,0) = 0$$

Dynamic Programming Table

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				
$w_4 = 2, v_4 = 15$	4						

$$F(3,1) = 10$$

Dynamic Programming Table

	i	capacity j: 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			
$w_4 = 2, v_4 = 15$	4						

$$F(3,2) = 12$$

Dynamic Programming Table

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		
$w_4 = 2, v_4 = 15$	4						

$$F(3,3) = 22$$

Dynamic Programming Table

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	
$w_4 = 2, v_4 = 15$	4						

$$F(3,4) = 30$$

Dynamic Programming Table

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						

$$F(3,5) = 32$$

Dynamic Programming Table

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					

$$F(4,0) = 0$$

Dynamic Programming Table

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				

$$F(4,1) = 10$$

Dynamic Programming Table

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			

$$F(4,2) = 15$$

Dynamic Programming Table

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		

$$F(4,3) = 25$$

Dynamic Programming Table

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	

$$F(4,4) = 30$$

Dynamic Programming Table

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**

$$F(4,5) = 37$$

Dynamic Programming Table

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**

Optimal subset:

Dynamic Programming Table

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**

Optimal subset: item 4, item 2, item 1. //backtracking

Knapsack Problem

- **Time efficiency:**

$$\Theta(nW)$$

- **Space efficiency:**

$$\Theta(nW)$$

- Can we do better?

  Combine the strengths of the top-down and bottom-up approaches: **solve only subproblems that are necessary & does so only once** (memory functions).

Exercises

Exercise 1: Solve the instance 7, 2, 1, 12, 5 of the coin-row problem.

Exercise 2: Find the most valuable subset of the items that fit into the knapsack of capacity 6.

ITEM	WEIGHT	VALUE
1	3	$13
2	1	$12
3	1	$10
4	2	$17
5	2	$15

Exercises

Exercise 3: Write pseudocode of the **bottom-up** dynamic programming algorithm for coin-row problem.

Exercise 4: Write pseudocode of the **bottom-up** dynamic programming algorithm for knapsack problem.

Exercise 5: Write pseudocode of the **top-down** dynamic programming algorithm for knapsack problem.