DP for the unbounded knapsack problem

Outline

Instances and feasible solutions
- Encoding of feasible solutions

A first DP algorithm
- Number of steps of the first DP algorithm

A second DP algorithm
- Number of steps of the second DP algorithm

Typical mistakes

In summary

Learning objectives

At the end of this chapter you will be able to:

- explain why the UKP differs from the 0/1 knapsack problem (KP), and write the recurrence $v(k)$ over the capacities only;
- run the DP algorithm on the capacities and on the pairs (item type, capacity), by hand;
- reconstruct an optimal solution, including the case in which part of the capacity is left unused;
- count the steps of the two DP algorithms and compare them.

Prerequisites

- From this module, the chapter «DP for the 0/1 knapsack problem»: the recurrence $v(j, k)$ over items and capacities, and the reconstruction of a solution.
- From this module, the chapter «Principles of dynamic programming»: optimal substructure, overlapping subproblems, base cases.
- From the module *Algorithms*: pseudocode conventions and the size of the input.

The problem of this chapter

The problem of this chapter is the unbounded knapsack problem (UKP): its definition and its assumptions are on its page.

Instances and feasible solutions

Introduction: instances and feasible solutions

- We introduce the UKP instance that we use throughout the chapter
- We bound the number of feasible solutions of a UKP instance, and see why enumerating them quickly becomes impractical
- We describe how a feasible solution is encoded, and the main difference with the KP
- This is the starting point of the DP algorithms of the next sections

Example: UKP instance

We consider a knapsack of capacity $c = 7$ and 4 item types, with the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
p_j	4	5	10	11
w_j	2	3	5	6

We refer to this UKP instance as the reference instance.

How many feasible solutions?

- For each item type j, a feasible solution can use at most $\lfloor c/w_j \rfloor$ copies of item type j, since using more copies would already exceed the capacity c with item type j alone.
- Accordingly, the number of feasible solutions is at most

$$\prod_{j=1}^{n} \left(\lfloor c/w_j \rfloor + 1 \right).$$

- The number of possible UKP solutions grows quickly with the number n of item types and the capacity c, since each variable x_j can independently take any of $\lfloor c/w_j \rfloor + 1$ values.
- Therefore, enumerating all possible solutions quickly becomes impractical as n or c increases.

Encoding of feasible solutions

A feasible solution is described by the integer vector

$$(x_1, x_2, \ldots, x_n),$$

where $x_j \in \mathbb{Z}_{\geq 0}$ is the number of copies of item type j selected.

The solution is feasible if

$$\sum_{j=1}^{n} w_j x_j \leq c.$$

Its total profit is

$$\sum_{j=1}^{n} p_j x_j.$$

The difference with the KP

- The main difference with the KP is that the variables are no longer binary.
- In the UKP, each variable x_j can take any non-negative integer value, provided that the capacity constraint is respected.

Summary

Instances and feasible solutions:

- A UKP instance: a capacity c and n item types, each with a profit p_j and a weight w_j, available in any number of copies
- A feasible solution uses at most $\lfloor c/w_j \rfloor$ copies of item type j: at most $\prod_{j=1}^{n} \left(\lfloor c/w_j \rfloor + 1 \right)$ feasible solutions, too many to enumerate

Encoding:

- An integer vector $(x_1, x_2, \ldots, x_n)$ with $\sum_{j=1}^{n} w_j x_j \leq c$ and profit $\sum_{j=1}^{n} p_j x_j$
- Unlike the KP, the variables are non-negative integers, not binary

A first DP algorithm

Introduction: a first DP algorithm

- In the UKP an item type can be used any number of times, so the optimal value depends only on the residual capacity
- We write a recurrence $v(k)$ over the capacities only, and verify optimal substructure and overlapping subproblems
- We compute the optimal value top-down and bottom-up, and then reconstruct an optimal solution
- Finally, we count the steps of the DP algorithm

Step 1: computation of the optimal value

For any $k \in \{0, 1, \ldots, c\}$, let $v(k)$ be the optimal value of the UKP subinstance given by:

1. a knapsack of capacity k,
2. the set $\{1, 2, \ldots, n\}$ of n item types,
3. for each item type $j \in \{1, 2, \ldots, n\}$, a profit p_j and a weight w_j.

The recurrence for $v(k)$

The recursive formula to compute $v(k)$ reads as follows:

$$v(k) = \begin{cases} 0, & \text{if } k = 0, \\ \max\limits_{j\,:\,w_j \le k} \{p_j + v(k - w_j)\}, & \text{if } k \ge 1 \text{ and } w_j \le k \text{ for some } j, \\ 0, & \text{if } k \ge 1 \text{ and } w_j > k \text{ for every } j. \end{cases}$$

Independent subproblems and base case

- The recursive decomposition process of the recursive formula for $v(k)$ generates at most n **independent subproblems**, each corresponding to a possible choice of an item type j, among those with $w_j \leq k$, to be inserted as one of the items of the knapsack of capacity k.

- The base case is $v(0) = 0$, since a knapsack of capacity 0 cannot contain any item and has profit 0.

- Moreover, if $w_j > k$ for every item type j, no item fits into the knapsack: the only feasible choice is to insert no item, and therefore $v(k) = 0$.

Optimal substructure

- The optimal value $v(k)$ is obtained by trying every item type j with $w_j \leq k$ and selecting the one that maximizes the sum $p_j + v(k - w_j)$ of the profit of the chosen item type and the optimal value of the residual subproblem.
- Since item type j is available in infinitely many copies, it may also be chosen again while solving the residual subproblem of capacity $k - w_j$.
- Hence, an optimal solution for a knapsack of capacity k is built by combining the chosen item type j with an optimal solution of the residual subproblem of capacity $k - w_j$.

$\rightsquigarrow$ **Optimal Substructure verified!**

Overlapping subproblems

- Different choices of the first item type, possibly made for different values of k, can lead to the same residual capacity.
- Therefore the same value $v(k)$ may need to be computed several times during the recursive evaluation of the recursive formula for $v(k)$.

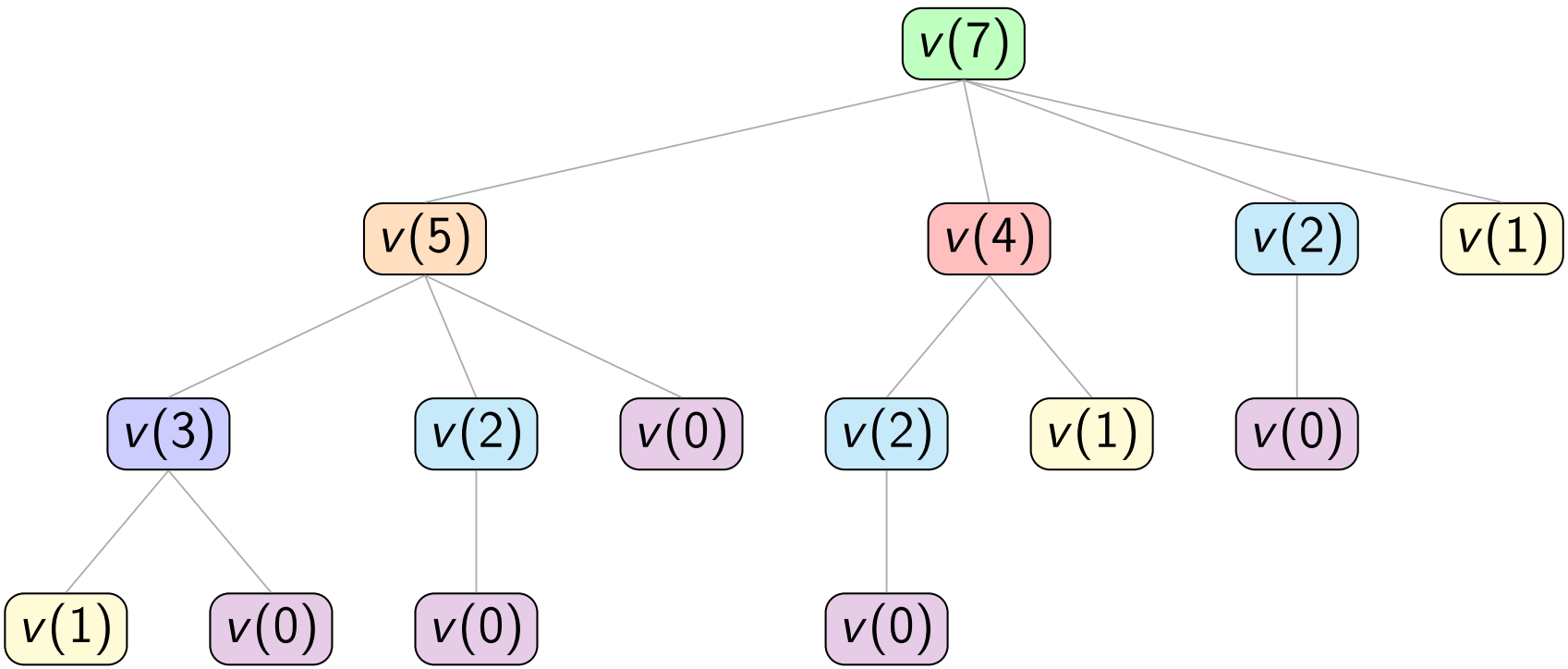
$\rightsquigarrow$ **Overlapping Subproblems verified!**

The top-down approach

- A **first method** to compute the optimal value $v(c)$ of the UKP instance is the **top-down approach**.
- The method starts from the call with $k = c$ and evaluates the recursive formula for $v(k)$ recursively: each call for a knapsack of capacity k triggers further calls for capacity $k - w_j$, for every item type j with $w_j \leq k$, until the base case $v(0)$ is reached or no item type fits into the residual capacity.
- The **recursion tree** of the function v has one node for each recursive call, labeled with the function call it represents. The root represents the initial call. The children of a node represent the recursive calls made by that call. The leaf nodes represent the base cases.

Example: recursion tree (1/2)

We consider the reference instance, with capacity $c = 7$ and item types with weights $w_1 = 2$, $w_2 = 3$, $w_3 = 5$, $w_4 = 6$. This is the recursion tree of the top-down approach to compute $v(7)$:



Example: recursion tree (2/2)

The function v is called 15 times, and the same subproblems are solved multiple times:

- $v(2)$ is called 3 times,
- $v(1)$ is called 3 times,
- $v(0)$ is called 5 times.

All the other calls are made only once.

Computing the values in increasing order of capacity

The value $v(k)$, for a generic capacity k, can be computed once the values $v(k')$ are known for all smaller capacities $k' < k$.

Example: computing the values of the v function (1/4)

We consider the reference instance with a knapsack of capacity $c = 7$ and the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
p_j	4	5	10	11
w_j	2	3	5	6

The value of the function v can be computed by starting from the base case $v(0) = 0$ and then computing the values of $v(1), v(2), \ldots, v(7)$ in order, as follows:

- Since no item type has $w_j \leq 1$, we have:

$$v(1) = 0.$$

Example: computing the values of the v function (2/4)

- Once we have computed $v(0)$, we can compute $v(2)$, as follows:

$$v(2) = p_1 + v(0) = 4 + 0 = 4.$$

- Once we have computed $v(1)$ and $v(0)$, we can compute $v(3)$, as follows:

$$v(3) = \max\{p_1 + v(1), p_2 + v(0)\} = \max\{4 + 0, 5 + 0\} = 5.$$

Example: computing the values of the v function (3/4)

- Once we have computed $v(2)$ and $v(1)$, we can compute $v(4)$, as follows:

$$v(4) = \max\{p_1 + v(2), p_2 + v(1)\} = \max\{4 + 4, 5 + 0\} = 8.$$

- Once we have computed $v(3)$, $v(2)$ and $v(0)$, we can compute $v(5)$, as follows:

$$\begin{aligned} v(5) &= \max\{p_1 + v(3), p_2 + v(2), p_3 + v(0)\} \\ &= \max\{4 + 5, 5 + 4, 10 + 0\} = 10. \end{aligned}$$

Example: computing the values of the v function (4/4)

- Once we have computed $v(4)$, $v(3)$, $v(1)$ and $v(0)$, we can compute $v(6)$, as follows:

$$\begin{aligned} v(6) &= \max\{p_1 + v(4), p_2 + v(3), p_3 + v(1), p_4 + v(0)\} \\ &= \max\{4 + 8, 5 + 5, 10 + 0, 11 + 0\} = 12. \end{aligned}$$

- Once we have computed $v(5)$, $v(4)$, $v(2)$ and $v(1)$, we can compute $v(7)$, as follows:

$$\begin{aligned} v(7) &= \max\{p_1 + v(5), p_2 + v(4), p_3 + v(2), p_4 + v(1)\} \\ &= \max\{4 + 10, 5 + 8, 10 + 4, 11 + 0\} = 14. \end{aligned}$$

The lookup table

- We can avoid solving the same subproblems repeatedly by storing the value of each subproblem as soon as it is computed. The values are stored in a **lookup table**.
- Since solving a subproblem with value $v(k)$ depends only on subproblems with values $v(k')$, with $k' < k$, we can solve the subproblems for increasing values of k, starting from the base case $v(0) = 0$ and building up to $v(c)$.
- This observation naturally leads to a **much more efficient method** to compute the optimal value $v(c)$ of the UKP instance.

The bottom-up approach

- A **second method** to compute the optimal value $v(c)$ of the UKP instance is the **bottom-up approach**.
- The method starts from the base case $v(0) = 0$ and evaluates the recursive formula for $v(k)$ for increasing capacities $k \in \{1, 2, \ldots, c\}$: each value $v(k)$ is computed using the already-computed values $v(0), v(1), \ldots, v(k-1)$, until the value $v(c)$ of the UKP instance is obtained.

The DP algorithm using the bottom-up approach (1/2)

This is the pseudo-code of the **DP algorithm using the bottom-up approach**:

Algorithm: UKP_DP1

Input : A UKP instance given by n, arrays $p[1..n]$, $w[1..n]$, and capacity c

Output : The optimal value of the UKP instance

```
1 function UKP_DP1(n, p, w, c)
2 |   v := new array[0..c]
3 |   v[0] ← 0
```

The DP algorithm using the bottom-up approach (2/2)

```
4     for k := 1 to c do
5         m := 0
6         for j := 1 to n do
7             if w[j] ≤ k then
8                 if p[j] + v[k − w[j]] > m then
9                     m ← p[j] + v[k − w[j]]
10                end
11            end
12        end
13        v[k] ← m
14    end
15    return v[c]
16 end
```

How the DP algorithm works

- The vector $v[0..c]$ is the **lookup table**: it stores the optimal values of all subproblems, with the base case $v[0] = 0$.
- The outer loop, over $k \in \{1, 2, \ldots, c\}$, computes $v[k]$ in increasing order of capacity.
- The inner loop, over $j \in \{1, 2, \ldots, n\}$, examines every item type; the condition $w[j] \leq k$ checks whether item type j fits into capacity k, and keeps in m the largest value $p[j] + v[k - w[j]]$ found so far. If no item type fits, m remains equal to its initial value 0.
- At the end of the inner loop, $v[k] \leftarrow m$ stores the optimal value $v[k]$.
- Finally, the optimal value $v[c]$ of the UKP instance is returned.

Example: execution of the DP algorithm (1/6)

We consider the reference instance with a knapsack of capacity $c = 7$ and the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
p_j	4	5	10	11
w_j	2	3	5	6

- For each step of the DP algorithm, the table shows the values of k and j, the current value of m, and the cell $v[k]$.
- The cell of m is highlighted in red when updated; the cell of $v[k]$ is highlighted in blue when assigned.

Example: execution of the DP algorithm (2/6)

k	j	$v[k]$	m	Action
–	–	0	–	$v[0] \leftarrow 0$
	–	–	0	$m := 0$
1	1	–	0	$w[1] = 2 > 1$: skip
	2	–	0	$w[2] = 3 > 1$: skip
	3	–	0	$w[3] = 5 > 1$: skip
	4	–	0	$w[4] = 6 > 1$: skip
	–	0	0	$v[1] \leftarrow m = 0$

Example: execution of the DP algorithm (3/6)

k	j	$v[k]$	m	Action
	–	–	0	$m := 0$
2	1	–	4	$p[1]+v[0] = 4+0 = 4 > m = 0$: $m \leftarrow 4$
	2	–	4	$w[2] = 3 > 2$: skip
	3	–	4	$w[3] = 5 > 2$: skip
	4	–	4	$w[4] = 6 > 2$: skip
	–	4	4	$v[2] \leftarrow m = 4$
	–	–	0	$m := 0$
3	1	–	4	$p[1]+v[1] = 4+0 = 4 > m = 0$: $m \leftarrow 4$
	2	–	5	$p[2]+v[0] = 5+0 = 5 > m = 4$: $m \leftarrow 5$
	3	–	5	$w[3] = 5 > 3$: skip
	4	–	5	$w[4] = 6 > 3$: skip
	–	5	5	$v[3] \leftarrow m = 5$

Example: execution of the DP algorithm (4/6)

k	j	$v[k]$	m	Action
	–	–	0	$m := 0$
4	1	–	8	$p[1]+v[2] = 4+4 = 8 > m = 0$: $m \leftarrow 8$
	2	–	8	$p[2]+v[1] = 5+0 = 5 \leq m = 8$: no update
	3	–	8	$w[3] = 5 > 4$: skip
	4	–	8	$w[4] = 6 > 4$: skip
	–	8	8	$v[4] \leftarrow m = 8$
	–	–	0	$m := 0$
5	1	–	9	$p[1]+v[3] = 4+5 = 9 > m = 0$: $m \leftarrow 9$
	2	–	9	$p[2]+v[2] = 5+4 = 9 = m$: no update
	3	–	10	$p[3]+v[0] = 10+0 = 10 > m = 9$: $m \leftarrow 10$
	4	–	10	$w[4] = 6 > 5$: skip
	–	10	10	$v[5] \leftarrow m = 10$

Example: execution of the DP algorithm (5/6)

k	j	$v[k]$	m	Action
	–	–	0	$m := 0$
6	1	–	12	$p[1]+v[4] = 4+8 = 12 > m = 0$: $m \leftarrow 12$
	2	–	12	$p[2]+v[3] = 5+5 = 10 \leq m = 12$: no update
	3	–	12	$p[3]+v[1] = 10+0 = 10 \leq m = 12$: no update
	4	–	12	$p[4]+v[0] = 11+0 = 11 \leq m = 12$: no update
	–	12	12	$v[6] \leftarrow m = 12$
	–	–	0	$m := 0$
7	1	–	14	$p[1]+v[5] = 4+10 = 14 > m = 0$: $m \leftarrow 14$
	2	–	14	$p[2]+v[4] = 5+8 = 13 \leq m = 14$: no update
	3	–	14	$p[3]+v[2] = 10+4 = 14 = m$: no update
	4	–	14	$p[4]+v[1] = 11+0 = 11 \leq m = 14$: no update
	–	14	14	$v[7] \leftarrow m = 14$

Example: execution of the DP algorithm (6/6)

This is the vector v after the execution of the DP algorithm:

	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
$v[k]$	0	0	4	5	8	10	12	14

After the execution of the DP algorithm, the vector v contains the optimal value $v[k]$ for every capacity.

Step 2: computation of an optimal solution

- To compute an optimal solution, it is necessary to store some additional information.
- For each capacity k, in addition to the optimal value $v[k]$, we store $s[k]$, the item type selected as first item in an optimal solution for a knapsack of capacity k; if no item type fits into capacity k, we set $s[k] = 0$.
- Starting from $k = c$, the value $s[k]$ gives the item type of the first selected item; the residual capacity becomes $k - w_{s[k]}$, and the same rule can be applied again, and so on, until the residual capacity reaches 0 or no item type fits into it ($s[k] = 0$), in which case the residual capacity is left unused.

Storing the information for an optimal solution (1/2)

This is the pseudo-code of the **DP algorithm using the bottom-up approach** with the storing of the information necessary to compute an optimal solution:

Algorithm: UKP_DP1_InfoSol

Input : A UKP instance given by n, arrays $p[1..n]$, $w[1..n]$, and capacity c

Output : The optimal value of the UKP instance and the information vector $s[0..c]$

```
1 function UKP_DP1_InfoSol(n, p, w, c)
2 |   v := new array[0..c]; s := new array[0..c]
3 |   v[0] ← 0; s[0] ← 0
```

Storing the information for an optimal solution (2/2)

```
4     for k := 1 to c do
5         m := 0; t := 0
6         for j := 1 to n do
7             if w[j] ≤ k then
8                 if p[j] + v[k − w[j]] > m then
9                     m ← p[j] + v[k − w[j]]; t ← j
10                end
11            end
12        end
13        v[k] ← m; s[k] ← t
14    end
15    return v[c], s
16 end
```

How the information is stored

- The DP algorithm uses the additional temporary variable t to keep track of the decision made when m is updated. If item type j is selected, $t \leftarrow j$ stores this decision; if no item type fits into capacity k, t keeps its initial value 0.

- At the end of the inner loop, $s[k] \leftarrow t$ stores the selected item type, together with the optimal value $v[k] \leftarrow m$.

Example: computation of the additional information (1/6)

We consider the reference instance with a knapsack of capacity $c = 7$ and the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
p_j	4	5	10	11
w_j	2	3	5	6

- For each step of the DP algorithm, the table shows the values of k and j, the cell $s[k]$, the current value of m, and the current value of t.
- The cell of m is highlighted in red when updated; the cell of t and the cell of $s[k]$ are highlighted in blue when updated.

Example: computation of the additional information (2/6)

k	j	$s[k]$	m	t	Action
–	–	–	–	–	Initial state: $v[0] \leftarrow 0$, $s[0] \leftarrow 0$
	–	–	0	0	$m := 0$, $t := 0$
1	1	–	0	0	$w[1] = 2 > 1$: skip
	2	–	0	0	$w[2] = 3 > 1$: skip
	3	–	0	0	$w[3] = 5 > 1$: skip
	4	–	0	0	$w[4] = 6 > 1$: skip
	–	0	0	0	$v[1] \leftarrow m = 0$, $s[1] \leftarrow t = 0$

Example: computation of the additional information (3/6)

k	j	$s[k]$	m	t	Action
	–	–	0	0	$m := 0,\ t := 0$
2	1	–	4	1	$4{+}0 = 4 > m = 0$: $m \leftarrow 4,\ t \leftarrow 1$
	2	–	4	1	$w[2] = 3 > 2$: skip
	3	–	4	1	$w[3] = 5 > 2$: skip
	4	–	4	1	$w[4] = 6 > 2$: skip
	–	1	4	1	$v[2] \leftarrow m = 4,\ s[2] \leftarrow t = 1$
	–	–	0	0	$m := 0,\ t := 0$
3	1	–	4	1	$4{+}0 = 4 > m = 0$: $m \leftarrow 4,\ t \leftarrow 1$
	2	–	5	2	$5{+}0 = 5 > m = 4$: $m \leftarrow 5,\ t \leftarrow 2$
	3	–	5	2	$w[3] = 5 > 3$: skip
	4	–	5	2	$w[4] = 6 > 3$: skip
	–	2	5	2	$v[3] \leftarrow m = 5,\ s[3] \leftarrow t = 2$

Example: computation of the additional information (4/6)

k	j	$s[k]$	m	t	Action
	–	–	0	0	$m := 0,\ t := 0$
4	1	–	8	1	$4{+}4 = 8 > m = 0$: $m \leftarrow 8,\ t \leftarrow 1$
	2	–	8	1	$5{+}0 = 5 \leq m = 8$: no update
	3	–	8	1	$w[3] = 5 > 4$: skip
	4	–	8	1	$w[4] = 6 > 4$: skip
	–	1	8	1	$v[4] \leftarrow m = 8,\ s[4] \leftarrow t = 1$
	–	–	0	0	$m := 0,\ t := 0$
5	1	–	9	1	$4{+}5 = 9 > m = 0$: $m \leftarrow 9,\ t \leftarrow 1$
	2	–	9	1	$5{+}4 = 9 = m$: no update
	3	–	10	3	$10{+}0 = 10 > m = 9$: $m \leftarrow 10,\ t \leftarrow 3$
	4	–	10	3	$w[4] = 6 > 5$: skip
	–	3	10	3	$v[5] \leftarrow m = 10,\ s[5] \leftarrow t = 3$

Example: computation of the additional information (5/6)

k	j	$s[k]$	m	t	Action
	–	–	0	0	$m := 0,\ t := 0$
6	1	–	12	1	$4{+}8 = 12 > m = 0$: $m \leftarrow 12,\ t \leftarrow 1$
	2	–	12	1	$5{+}5 = 10 \leq m = 12$: no update
	3	–	12	1	$10{+}0 = 10 \leq m = 12$: no update
	4	–	12	1	$11{+}0 = 11 \leq m = 12$: no update
	–	1	12	1	$v[6] \leftarrow m = 12,\ s[6] \leftarrow t = 1$
	–	–	0	0	$m := 0,\ t := 0$
7	1	–	14	1	$4{+}10 = 14 > m = 0$: $m \leftarrow 14,\ t \leftarrow 1$
	2	–	14	1	$5{+}8 = 13 \leq m = 14$: no update
	3	–	14	1	$10{+}4 = 14 = m$: no update
	4	–	14	1	$11{+}0 = 11 \leq m = 14$: no update
	–	1	14	1	$v[7] \leftarrow m = 14,\ s[7] \leftarrow t = 1$

Example: computation of the additional information (6/6)

This is the vector s after the execution of the DP algorithm:

	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
$s[k]$	0	0	1	2	1	3	1	1

Computing an optimal solution from the vector s

- Once the vector s has been computed, an optimal solution of the UKP instance can be computed by repeatedly applying the following rule, starting from $k = c$ and $x_j = 0$ for all $j \in \{1, 2, \ldots, n\}$:

$$x_{s[k]} \leftarrow x_{s[k]} + 1, \qquad k \leftarrow k - w_{s[k]},$$

  until $k = 0$ or $s[k] = 0$.

- The second case occurs when no item type fits into the residual capacity k, which is then left unused.
- This yields the vector $(x_1, x_2, \ldots, x_n)$, where each entry x_j is the number of copies of item type j in an optimal solution.
- The following pseudo-code implements the computation of an optimal solution, given the vector s.

Algorithm: UKP_DP1_SolComputation

Input : The number n of item types, the capacity c, the array $w[1..n]$, and the vector $s[0..c]$

Output : An optimal solution of the UKP instance

```
1  function UKP_DP1_SolComputation(n, w, c, s)
2      x := new array[1..n]
3      for j := 1 to n do
4          x[j] ← 0
5      end
6      k := c
7      while k > 0 and s[k] > 0 do
8          x[s[k]] ← x[s[k]] + 1
9          k ← k − w[s[k]]
10     end
11     return x
12 end
```

Example: computing an optimal solution (1/2)

After the execution of the DP algorithm on the reference instance, we have the following vector s:

	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
$s[k]$	0	0	1	2	1	3	1	1

We can compute an optimal solution for a knapsack of capacity $c = 7$ as follows:

Iteration	k	$s[k]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$k \leftarrow k - w_{s[k]}$
–	–	–	0	0	0	0	$k = 7$
1	7	1	1	0	0	0	$k = 7 - 2 = 5$
2	5	3	1	0	1	0	$k = 5 - 5 = 0$

Example: computing an optimal solution (2/2)

An optimal solution is the vector:

$$(1, 0, 1, 0)$$

- Its total weight is $w_1 + w_3 = 2 + 5 = 7 = c$.
- Its total profit is $p_1 + p_3 = 4 + 10 = 14$, which matches the optimal value $v(7) = 14$.

Number of steps

- Let $S(n, c)$ denote the number of steps executed by `UKP_DP1` in the worst case, where n is the number of item types and c is the knapsack capacity.
- We compute $S(n, c)$ by counting how many times the lines of the pseudo-code of `UKP_DP1` are executed.
- In the worst case, the conditions at lines 7 and 8 are true at every iteration of the inner loop, and therefore line 9 is executed as often as line 8.

We recall the pseudo-code of UKP_DP1 (1/2)

Algorithm: UKP_DP1

Input : A UKP instance given by n, arrays $p[1..n]$, $w[1..n]$, and capacity c

Output : The optimal value of the UKP instance

```
1 function UKP_DP1(n, p, w, c)
2 |   v := new array[0..c]
3 |   v[0] ← 0
```

We recall the pseudo-code of UKP_DP1 (2/2)

```
4   |   for k := 1 to c do
5   |   |   m := 0
6   |   |   for j := 1 to n do
7   |   |   |   if w[j] ≤ k then
8   |   |   |   |   if p[j] + v[k − w[j]] > m then
9   |   |   |   |   |   m ← p[j] + v[k − w[j]]
10  |   |   |   |   end
11  |   |   |   end
12  |   |   end
13  |   |   v[k] ← m
14  |   end
15  |   return v[c]
16  end
```

Number of executions of the lines

The number of executions of the relevant lines is:

- Line 2 is executed 1 time.
- Line 3 is executed 1 time.
- Line 4 is executed c times.
- Line 5 is executed c times.
- Line 6 is executed cn times.
- Line 7 is executed cn times.
- Line 8 is executed cn times.
- Line 9 is executed cn times.
- Line 13 is executed c times.
- Line 15 is executed 1 time.

The total number of steps

Therefore, the total number of steps in the worst case is

$$1 + 1 + c + c + cn + cn + cn + cn + c + 1.$$

Thus,

$$S(n, c) = 3 + 3c + 4cn.$$

Hence,

$$S(n, c) = \Theta(nc).$$

Indeed,

$$\lim_{n,c\to\infty} \frac{4cn + 3c + 3}{nc} = \lim_{n,c\to\infty} \left(4 + \frac{3}{n} + \frac{3}{nc}\right) = 4 \in \mathbb{R}_{>0}.$$

The number of steps of the first DP algorithm

The DP algorithm executes

$$S(n, c) = \Theta(nc)$$

steps in the worst case.

- The version `UKP_DP1_InfoSol` stores the additional vector s and performs only a constant number of extra operations inside the same loops.
- Therefore, its number of steps remains $\Theta(nc)$.

Summary (1/2)

The recurrence over the capacities:

- $v(k)$ is the optimal value for a knapsack of capacity k with all the n item types
- $v(0) = 0$; $v(k) = \max_{j:\, w_j \leq k}\{p_j + v(k - w_j)\}$, or 0 if no item type fits
- At most n independent subproblems; optimal substructure and overlapping subproblems verified

Top-down and bottom-up:

- The recursion tree for $v(7)$ has 15 calls, with repeated subproblems
- `UKP_DP1` fills the lookup table $v[0..c]$ for increasing k; on the reference instance $v[7] = 14$

Summary (2/2)

An optimal solution:

- $s[k]$ is the item type selected first for capacity k, or 0 if no item type fits
- From $k = c$: one more copy of item type $s[k]$, then $k \leftarrow k - w_{s[k]}$, until $k = 0$ or $s[k] = 0$
- On the reference instance: $(1, 0, 1, 0)$, with profit 14

Number of steps:

- $S(n, c) = 3 + 3c + 4cn = \Theta(nc)$, also with the vector s

A second DP algorithm

Introduction: a second DP algorithm

- We write a second recurrence, $v(j, k)$, over the pairs (item type, capacity), as for the KP
- The difference with the KP: after inserting one copy of item type j, the recurrence stays in the same row j, because further copies are still available
- We follow the same path: top-down, bottom-up, reconstruction of an optimal solution
- Finally, we count the steps and compare them with those of the first DP algorithm

Step 1: computation of the optimal value

For any $j \in \{0, 1, \ldots, n\}$ and $k \in \{0, 1, \ldots, c\}$, let $v(j, k)$ be the optimal value of the UKP subinstance given by:

1. a knapsack of capacity k,
2. the set $\{1, 2, \ldots, j\}$ of j item types,
3. for each item type $h \in \{1, 2, \ldots, j\}$, a profit p_h and a weight w_h.

The recurrence for $v(j, k)$

The recursive formula to compute $v(j, k)$ reads as follows:

$$v(j,k) = \begin{cases} 0, & \text{if } j = 0, \\ v(j-1,k), & \text{if } j \geq 1 \text{ and } k < w_j, \\ \max\{v(j-1,k),\ p_j + v(j,k-w_j)\}, & \text{if } j \geq 1 \text{ and } k \geq w_j. \end{cases}$$

Independent subproblems and base case

- The recursive decomposition process of the recursive formula for $v(j, k)$ generates at most 2 **independent subproblems**, corresponding to whether or not to insert one copy of item type j into a knapsack of capacity k, using only item types $1, 2, \ldots, j$.

- If $w_j > k$, item type j cannot be inserted, and therefore the only feasible option is not to use it.

- The base case is $v(0, k) = 0$ for every k, since with no item type available the only feasible choice is to insert no item, which always has profit 0.

The two options

- If $j \geq 1$ and $k \geq w_j$, the optimal value $v(j, k)$ is obtained by trying the two possible options, not inserting item type j or inserting one further copy of it, and selecting the one that maximizes the profit. The corresponding values are $v(j-1, k)$ and $p_j + v(j, k - w_j)$, respectively.

- If $j \geq 1$ and $k < w_j$, item type j cannot be used, and therefore $v(j, k) = v(j-1, k)$.

- The term $p_j + v(j, k - w_j)$ refers to the same row j, because after inserting one copy of item type j, additional copies of the same item type are still available.

Optimal substructure and overlapping subproblems

- Hence, an optimal solution for a knapsack of capacity k using only item types $1, 2, \ldots, j$ is built by combining the chosen decision on item type j with an optimal solution of the corresponding residual subproblem.

$$\rightsquigarrow \textbf{Optimal Substructure verified!}$$

- Different sequences of decisions on the item types, possibly made for different values of j and k, can lead to the same subproblem, and therefore the same value $v(j, k)$ may need to be computed several times during the recursive evaluation of the recursive formula for $v(j, k)$.

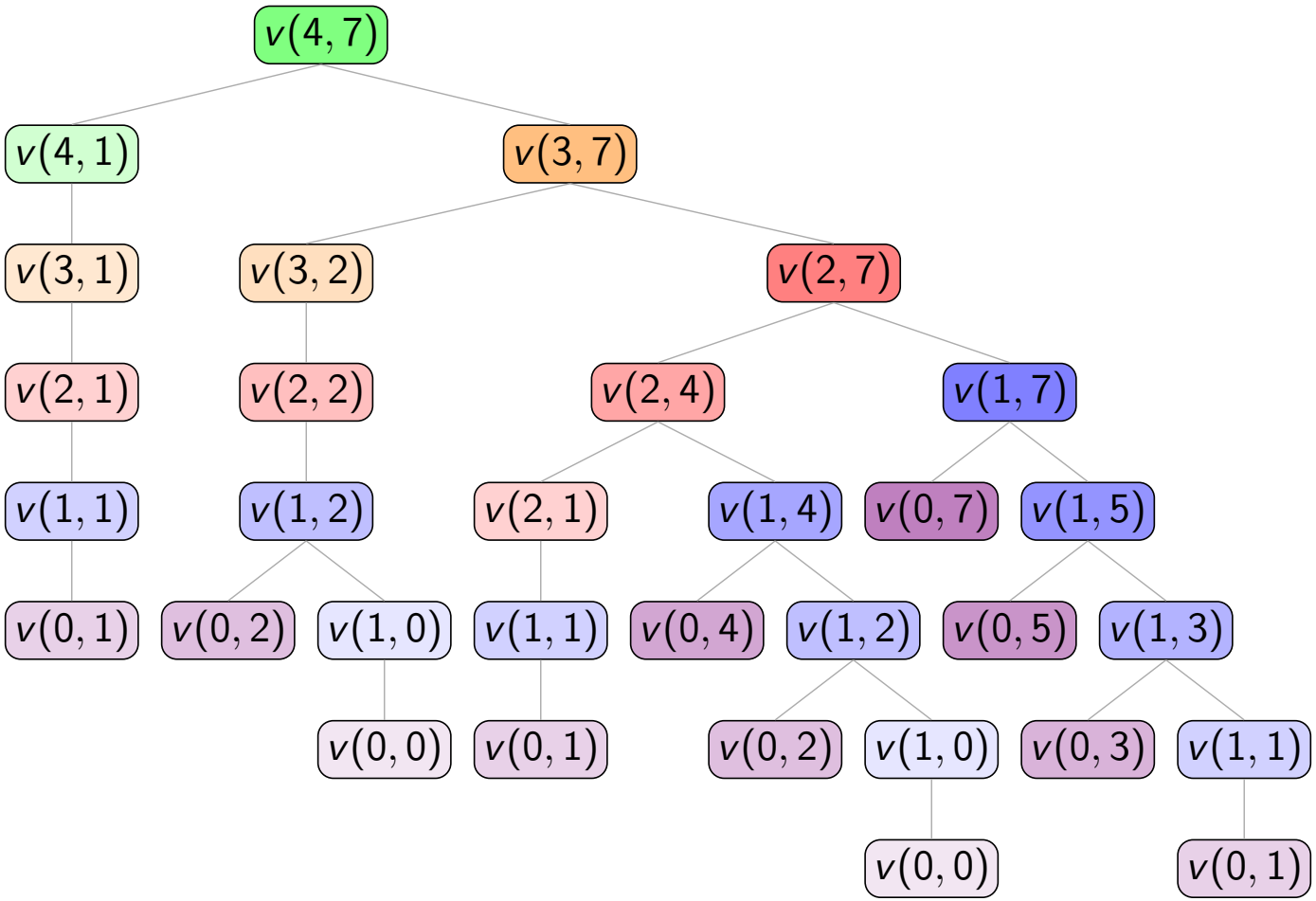
$$\rightsquigarrow \textbf{Overlapping Subproblems verified!}$$

The top-down approach

- A **first method** to compute the optimal value $v(n, c)$ of the UKP instance is the **top-down approach**.
- The method starts from the call with $(j, k) = (n, c)$ and evaluates the recursive formula for $v(j, k)$ recursively: each call for capacity k using only item types $1, 2, \ldots, j$ triggers further calls corresponding to the feasible decisions on item type j, until one of the base cases $v(0, k)$ is reached.
- The **recursion tree** of the function v has one node for each recursive call, labeled with the function call it represents. The root represents the initial call. The children of a node represent the recursive calls made by that call. The leaf nodes represent the base cases.

Example: recursion tree (1/2)

We consider the reference instance, with capacity $c = 7$ and item types with weights $w_1 = 2$, $w_2 = 3$, $w_3 = 5$, $w_4 = 6$. This is the recursion tree of the top-down approach to compute $v(4, 7)$:



Example: recursion tree (2/2)

The function v is called 32 times, and the same subproblems are solved multiple times:

- $v(1,1)$ is called 3 times,
- $v(0,1)$ is called 3 times,
- $v(1,2)$ is called 2 times,
- $v(0,2)$ is called 2 times,
- $v(1,0)$ is called 2 times,
- $v(0,0)$ is called 2 times,
- $v(2,1)$ is called 2 times.

All the other calls are made only once.

Computing the values row by row

- The value $v(j, k)$ can be computed once $v(j-1, k)$ is known and, when $k \geq w_j$, $v(j, k-w_j)$ is also known.
- Therefore, for a fixed value of j, all values $v(j, k)$ can be computed by considering the values of k in increasing order.
- Starting from the base case $v(0, k) = 0$ for every k, the values of the function v can be computed row by row, moving to the next value of j only after all values $v(j, k)$ for the current value of j have been computed.

Example: computing the values of the v function (1/5)

We consider the reference instance with a knapsack of capacity $c = 7$ and the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
p_j	4	5	10	11
w_j	2	3	5	6

The values of the function v are computed as follows, one value of j at a time, starting from the base case $v(0, k) = 0$ for every k.

Example: computing the values of the v function (2/5)

For $j = 1$, only item type 1 ($w_1 = 2$) is available. Therefore:

$$v(1,0) = 0, \qquad v(1,1) = v(0,1) = 0,$$
$$v(1,2) = \max\{v(0,2), p_1 + v(1,0)\} = \max\{0, 4+0\} = 4,$$
$$v(1,3) = \max\{v(0,3), p_1 + v(1,1)\} = \max\{0, 4+0\} = 4,$$
$$v(1,4) = \max\{v(0,4), p_1 + v(1,2)\} = \max\{0, 4+4\} = 8,$$
$$v(1,5) = \max\{v(0,5), p_1 + v(1,3)\} = \max\{0, 4+4\} = 8,$$
$$v(1,6) = \max\{v(0,6), p_1 + v(1,4)\} = \max\{0, 4+8\} = 12,$$
$$v(1,7) = \max\{v(0,7), p_1 + v(1,5)\} = \max\{0, 4+8\} = 12.$$

Example: computing the values of the v function (3/5)

For $j = 2$, item types 1 and 2 are available. We obtain:

$$v(2,0) = 0, \qquad v(2,1) = v(1,1) = 0, \qquad v(2,2) = v(1,2) = 4,$$
$$v(2,3) = \max\{v(1,3), p_2 + v(2,0)\} = \max\{4, 5+0\} = 5,$$
$$v(2,4) = \max\{v(1,4), p_2 + v(2,1)\} = \max\{8, 5+0\} = 8,$$
$$v(2,5) = \max\{v(1,5), p_2 + v(2,2)\} = \max\{8, 5+4\} = 9,$$
$$v(2,6) = \max\{v(1,6), p_2 + v(2,3)\} = \max\{12, 5+5\} = 12,$$
$$v(2,7) = \max\{v(1,7), p_2 + v(2,4)\} = \max\{12, 5+8\} = 13.$$

Example: computing the values of the v function (4/5)

For $j = 3$, item types 1, 2 and 3 are available. We obtain:

$$v(3,0) = 0, \qquad v(3,1) = v(2,1) = 0, \qquad v(3,2) = v(2,2) = 4,$$
$$v(3,3) = v(2,3) = 5, \qquad v(3,4) = v(2,4) = 8,$$
$$v(3,5) = \max\{v(2,5), p_3 + v(3,0)\} = \max\{9, 10+0\} = 10,$$
$$v(3,6) = \max\{v(2,6), p_3 + v(3,1)\} = \max\{12, 10+0\} = 12,$$
$$v(3,7) = \max\{v(2,7), p_3 + v(3,2)\} = \max\{13, 10+4\} = 14.$$

Example: computing the values of the v function (5/5)

For $j = 4$, all item types are available. We obtain:

$$v(4,0) = 0, \quad v(4,1) = v(3,1) = 0, \quad v(4,2) = v(3,2) = 4,$$
$$v(4,3) = v(3,3) = 5, \quad v(4,4) = v(3,4) = 8,$$
$$v(4,5) = v(3,5) = 10,$$
$$v(4,6) = \max\{v(3,6), p_4 + v(4,0)\} = \max\{12, 11 + 0\} = 12,$$
$$v(4,7) = \max\{v(3,7), p_4 + v(4,1)\} = \max\{14, 11 + 0\} = 14.$$

The lookup table

- We can avoid solving the same subproblems repeatedly by storing the value of each subproblem as soon as it is computed. The values are stored in a **lookup table**.
- Since solving any particular subproblem depends only on subproblems with a smaller value of j, or with the same value of j and a smaller capacity, we can solve the subproblems in increasing order of j and, for each fixed value of j, in increasing order of capacity, starting from the base cases $v(0, k) = 0$ for every $k \in \{0, 1, \ldots, c\}$ and building up to $v(n, c)$.
- This observation naturally leads to a **much more efficient method** to compute the optimal value $v(n, c)$ of the UKP instance.

The bottom-up approach

- A **second method** to compute the optimal value $v(n, c)$ of the UKP instance is the **bottom-up approach**.
- The method starts from the base cases $v(0, k) = 0$ for every $k \in \{0, 1, \ldots, c\}$ and evaluates the recursive formula for $v(j, k)$ for increasing values of $j \in \{1, 2, \ldots, n\}$ and $k \in \{0, 1, \ldots, c\}$: each value $v(j, k)$ is computed using already-computed values, until the value $v(n, c)$ of the UKP instance is obtained.

The DP algorithm using the bottom-up approach (1/2)

This is the pseudo-code of the **DP algorithm using the bottom-up approach**:

Algorithm: UKP_DP2

Input : A UKP instance given by n, arrays $p[1..n]$, $w[1..n]$, and capacity c

Output : The optimal value of the UKP instance

```
1 function UKP_DP2(n, p, w, c)
2 |   v := new array[0..n][0..c]
3 |   for k := 0 to c do
4 |   |   v[0][k] ← 0
5 |   end
```

The DP algorithm using the bottom-up approach (2/2)

```
6   |  for j := 1 to n do
7   |  |  for k := 0 to c do
8   |  |  |  if k < w[j] then
9   |  |  |  |  v[j][k] ← v[j − 1][k]
10  |  |  |  else
11  |  |  |  |  if p[j] + v[j][k − w[j]] > v[j − 1][k] then
12  |  |  |  |  |  v[j][k] ← p[j] + v[j][k − w[j]]
13  |  |  |  |  else
14  |  |  |  |  |  v[j][k] ← v[j − 1][k]
15  |  |  |  |  end
16  |  |  |  end
17  |  |  end
18  |  end
19  |  return v[n][c]
20  end
```

How the DP algorithm works (1/2)

- The matrix $v[0..n][0..c]$ is the **lookup table**: it stores the optimal values of all subproblems.
- The initialization sets $v[0][k] = 0$ for every capacity $k \in \{0, 1, \ldots, c\}$, since selecting no item is always feasible with profit 0.
- The outer loop, over $j \in \{1, 2, \ldots, n\}$, considers the available item types in increasing order.
- The inner loop, over $k \in \{0, 1, \ldots, c\}$, computes the best value for each capacity using only item types $1, 2, \ldots, j$.

How the DP algorithm works (2/2)

- If $k < w[j]$, item type j cannot be used, and therefore $v[j][k] \leftarrow v[j-1][k]$.
- Otherwise, the DP algorithm compares the value obtained by not using item type j, namely $v[j-1][k]$, with the value obtained by inserting one further copy of item type j, namely $p[j] + v[j][k-w[j]]$.
- Finally, the optimal value of the UKP instance is returned as $v[n][c]$.

Example: execution of the DP algorithm (1/7)

We consider the reference instance with a knapsack of capacity $c = 7$ and the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
p_j	4	5	10	11
w_j	2	3	5	6

For each step of the DP algorithm, the table shows the values of j and k, the cell $v[j][k]$, and the operation performed.

Example: execution of the DP algorithm (2/7)

j	k	$v[j][k]$	Action
0	0	0	$v[0][0] \leftarrow 0$
0	1	0	$v[0][1] \leftarrow 0$
0	2	0	$v[0][2] \leftarrow 0$
0	3	0	$v[0][3] \leftarrow 0$
0	4	0	$v[0][4] \leftarrow 0$
0	5	0	$v[0][5] \leftarrow 0$
0	6	0	$v[0][6] \leftarrow 0$
0	7	0	$v[0][7] \leftarrow 0$

Example: execution of the DP algorithm (3/7)

j	k	$v[j][k]$	Action
1	0	0	$k = 0 < w[1] = 2$: $v[1][0] \leftarrow v[0][0] = 0$
1	1	0	$k = 1 < w[1] = 2$: $v[1][1] \leftarrow v[0][1] = 0$
1	2	4	$\max\{v[0][2], p[1] + v[1][0]\} = \max\{0, 4 + 0\} = 4$
1	3	4	$\max\{v[0][3], p[1] + v[1][1]\} = \max\{0, 4 + 0\} = 4$
1	4	8	$\max\{v[0][4], p[1] + v[1][2]\} = \max\{0, 4 + 4\} = 8$
1	5	8	$\max\{v[0][5], p[1] + v[1][3]\} = \max\{0, 4 + 4\} = 8$
1	6	12	$\max\{v[0][6], p[1] + v[1][4]\} = \max\{0, 4 + 8\} = 12$
1	7	12	$\max\{v[0][7], p[1] + v[1][5]\} = \max\{0, 4 + 8\} = 12$

Example: execution of the DP algorithm (4/7)

j	k	$v[j][k]$	Action
2	0	0	$k = 0 < w[2] = 3$: $v[2][0] \leftarrow v[1][0] = 0$
2	1	0	$k = 1 < w[2] = 3$: $v[2][1] \leftarrow v[1][1] = 0$
2	2	4	$k = 2 < w[2] = 3$: $v[2][2] \leftarrow v[1][2] = 4$
2	3	5	$\max\{v[1][3], p[2] + v[2][0]\} = \max\{4, 5 + 0\} = 5$
2	4	8	$\max\{v[1][4], p[2] + v[2][1]\} = \max\{8, 5 + 0\} = 8$
2	5	9	$\max\{v[1][5], p[2] + v[2][2]\} = \max\{8, 5 + 4\} = 9$
2	6	12	$\max\{v[1][6], p[2] + v[2][3]\} = \max\{12, 5 + 5\} = 12$
2	7	13	$\max\{v[1][7], p[2] + v[2][4]\} = \max\{12, 5 + 8\} = 13$

Example: execution of the DP algorithm (5/7)

j	k	$v[j][k]$	Action
3	0	0	$k = 0 < w[3] = 5$: $v[3][0] \leftarrow v[2][0] = 0$
3	1	0	$k = 1 < w[3] = 5$: $v[3][1] \leftarrow v[2][1] = 0$
3	2	4	$k = 2 < w[3] = 5$: $v[3][2] \leftarrow v[2][2] = 4$
3	3	5	$k = 3 < w[3] = 5$: $v[3][3] \leftarrow v[2][3] = 5$
3	4	8	$k = 4 < w[3] = 5$: $v[3][4] \leftarrow v[2][4] = 8$
3	5	10	$\max\{v[2][5], p[3] + v[3][0]\} = \max\{9, 10 + 0\} = 10$
3	6	12	$\max\{v[2][6], p[3] + v[3][1]\} = \max\{12, 10 + 0\} = 12$
3	7	14	$\max\{v[2][7], p[3] + v[3][2]\} = \max\{13, 10 + 4\} = 14$

Example: execution of the DP algorithm (6/7)

j	k	$v[j][k]$	Action
4	0	0	$k = 0 < w[4] = 6$: $v[4][0] \leftarrow v[3][0] = 0$
4	1	0	$k = 1 < w[4] = 6$: $v[4][1] \leftarrow v[3][1] = 0$
4	2	4	$k = 2 < w[4] = 6$: $v[4][2] \leftarrow v[3][2] = 4$
4	3	5	$k = 3 < w[4] = 6$: $v[4][3] \leftarrow v[3][3] = 5$
4	4	8	$k = 4 < w[4] = 6$: $v[4][4] \leftarrow v[3][4] = 8$
4	5	10	$k = 5 < w[4] = 6$: $v[4][5] \leftarrow v[3][5] = 10$
4	6	12	$\max\{v[3][6], p[4] + v[4][0]\} = \max\{12, 11 + 0\} = 12$
4	7	14	$\max\{v[3][7], p[4] + v[4][1]\} = \max\{14, 11 + 0\} = 14$

Example: execution of the DP algorithm (7/7)

This is the matrix v after the execution of the DP algorithm:

$v[j][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
$j=0$	0	0	0	0	0	0	0	0
$j=1$	0	0	4	4	8	8	12	12
$j=2$	0	0	4	5	8	9	12	13
$j=3$	0	0	4	5	8	10	12	14
$j=4$	0	0	4	5	8	10	12	14

After the execution of the DP algorithm, the matrix v contains the optimal value $v[j][k]$ for every subproblem (j, k).

Step 2: computation of an optimal solution

- To compute an optimal solution, it is necessary to store some additional information.
- For each item type j and each capacity k, in addition to the optimal value $v[j][k]$, we store $s[j][k]$, which is equal to 1 if one copy of item type j is used when computing $v[j][k]$, and to 0 otherwise.
- Starting from $j = n$ and $k = c$, if $s[j][k] = 0$, then item type j is not used and j is decreased by 1. If $s[j][k] = 1$, then one copy of item type j is used and the residual capacity becomes $k - w_j$.
- The procedure stops when the residual capacity reaches 0 or when $j = 0$, in which case the residual capacity is left unused.

Storing the information for an optimal solution (1/2)

This is the pseudo-code of the **DP algorithm using the bottom-up approach** with the storing of the information necessary to compute an optimal solution:

Algorithm: UKP_DP2_InfoSol

Input : A UKP instance given by n, arrays $p[1..n]$, $w[1..n]$, and capacity c

Output : The optimal value of the UKP instance and the matrix of binary information $s[0..n][0..c]$

```
1 function UKP_DP2_InfoSol(n, p, w, c)
2 |   v := new array[0..n][0..c]; s := new array[0..n][0..c]
3 |   for k := 0 to c do
4 |   |   v[0][k] ← 0; s[0][k] ← 0
5 |   end
```

Storing the information for an optimal solution (2/2)

```
6     for j := 1 to n do
7         for k := 0 to c do
8             if k < w[j] then
9                 v[j][k] ← v[j − 1][k]; s[j][k] ← 0
10            else
11                if p[j] + v[j][k − w[j]] > v[j − 1][k] then
12                    v[j][k] ← p[j] + v[j][k − w[j]]; s[j][k] ← 1
13                else
14                    v[j][k] ← v[j − 1][k]; s[j][k] ← 0
15                end
16            end
17        end
18    end
19    return v[n][c], s
20 end
```

How the information is stored

- The DP algorithm uses the additional matrix s, which keeps track of the decision made when $v[j][k]$ is computed.
- If one copy of item type j is selected, $s[j][k] \leftarrow 1$ stores this decision; otherwise, $s[j][k] \leftarrow 0$ stores that item type j is not used.

Example: computation of the additional information (1/7)

We consider the reference instance with a knapsack of capacity $c = 7$ and the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
p_j	4	5	10	11
w_j	2	3	5	6

For each step of the DP algorithm, the table shows the values of j and k, the cell $s[j][k]$, and the operation performed.

Example: computation of the additional information (2/7)

j	k	$s[j][k]$	Action
0	0	0	$s[0][0] \leftarrow 0$
0	1	0	$s[0][1] \leftarrow 0$
0	2	0	$s[0][2] \leftarrow 0$
0	3	0	$s[0][3] \leftarrow 0$
0	4	0	$s[0][4] \leftarrow 0$
0	5	0	$s[0][5] \leftarrow 0$
0	6	0	$s[0][6] \leftarrow 0$
0	7	0	$s[0][7] \leftarrow 0$

Example: computation of the additional information (3/7)

j	k	$s[j][k]$	Action
1	0	0	$k < w[1]$: $s[1][0] \leftarrow 0$
1	1	0	$k < w[1]$: $s[1][1] \leftarrow 0$
1	2	1	$p[1] + v[1][0] = 4 + 0 > v[0][2] = 0$: $s[1][2] \leftarrow 1$
1	3	1	$p[1] + v[1][1] = 4 + 0 > v[0][3] = 0$: $s[1][3] \leftarrow 1$
1	4	1	$p[1] + v[1][2] = 4 + 4 > v[0][4] = 0$: $s[1][4] \leftarrow 1$
1	5	1	$p[1] + v[1][3] = 4 + 4 > v[0][5] = 0$: $s[1][5] \leftarrow 1$
1	6	1	$p[1] + v[1][4] = 4 + 8 > v[0][6] = 0$: $s[1][6] \leftarrow 1$
1	7	1	$p[1] + v[1][5] = 4 + 8 > v[0][7] = 0$: $s[1][7] \leftarrow 1$

Example: computation of the additional information (4/7)

j	k	$s[j][k]$	Action
2	0	0	$k < w[2]$: $s[2][0] \leftarrow 0$
2	1	0	$k < w[2]$: $s[2][1] \leftarrow 0$
2	2	0	$k < w[2]$: $s[2][2] \leftarrow 0$
2	3	1	$p[2] + v[2][0] = 5 + 0 > v[1][3] = 4$: $s[2][3] \leftarrow 1$
2	4	0	$p[2] + v[2][1] = 5 + 0 \leq v[1][4] = 8$: $s[2][4] \leftarrow 0$
2	5	1	$p[2] + v[2][2] = 5 + 4 > v[1][5] = 8$: $s[2][5] \leftarrow 1$
2	6	0	$p[2] + v[2][3] = 5 + 5 \leq v[1][6] = 12$: $s[2][6] \leftarrow 0$
2	7	1	$p[2] + v[2][4] = 5 + 8 > v[1][7] = 12$: $s[2][7] \leftarrow 1$

Example: computation of the additional information (5/7)

j	k	$s[j][k]$	Action
3	0	0	$k < w[3]$: $s[3][0] \leftarrow 0$
3	1	0	$k < w[3]$: $s[3][1] \leftarrow 0$
3	2	0	$k < w[3]$: $s[3][2] \leftarrow 0$
3	3	0	$k < w[3]$: $s[3][3] \leftarrow 0$
3	4	0	$k < w[3]$: $s[3][4] \leftarrow 0$
3	5	1	$p[3] + v[3][0] = 10 + 0 > v[2][5] = 9$: $s[3][5] \leftarrow 1$
3	6	0	$p[3] + v[3][1] = 10 + 0 \leq v[2][6] = 12$: $s[3][6] \leftarrow 0$
3	7	1	$p[3] + v[3][2] = 10 + 4 > v[2][7] = 13$: $s[3][7] \leftarrow 1$

Example: computation of the additional information (6/7)

j	k	$s[j][k]$	Action
4	0	0	$k < w[4]$: $s[4][0] \leftarrow 0$
4	1	0	$k < w[4]$: $s[4][1] \leftarrow 0$
4	2	0	$k < w[4]$: $s[4][2] \leftarrow 0$
4	3	0	$k < w[4]$: $s[4][3] \leftarrow 0$
4	4	0	$k < w[4]$: $s[4][4] \leftarrow 0$
4	5	0	$k < w[4]$: $s[4][5] \leftarrow 0$
4	6	0	$p[4] + v[4][0] = 11 + 0 \leq v[3][6] = 12$: $s[4][6] \leftarrow 0$
4	7	0	$p[4] + v[4][1] = 11 + 0 \leq v[3][7] = 14$: $s[4][7] \leftarrow 0$

Example: computation of the additional information (7/7)

This is the matrix s after the execution of the DP algorithm:

$s[j][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
$j=0$	0	0	0	0	0	0	0	0
$j=1$	0	0	1	1	1	1	1	1
$j=2$	0	0	0	1	0	1	0	1
$j=3$	0	0	0	0	0	1	0	1
$j=4$	0	0	0	0	0	0	0	0

Computing an optimal solution from the matrix s

- Once the matrix s has been computed, an optimal solution for a knapsack of capacity c can be computed by repeatedly applying the following rule, starting from $(j, k) = (n, c)$ and $x_h = 0$ for all $h \in \{1, 2, \ldots, n\}$:

$$\begin{aligned} & j \leftarrow j - 1 && \text{if } s[j][k] = 0, \\ & x_j \leftarrow x_j + 1, \quad k \leftarrow k - w_j && \text{if } s[j][k] = 1. \end{aligned}$$

- The procedure stops when $k = 0$ or $j = 0$; the second case occurs when the residual capacity is left unused.
- This yields the vector $(x_1, x_2, \ldots, x_n)$, where each entry x_h is the number of copies of item type h in an optimal solution.
- The following pseudo-code implements the computation of an optimal solution, given the matrix s.

Computing an optimal solution: the pseudo-code (1/2)

Algorithm: UKP_DP2_SolComputation

Input : The number of item types n, the array $w[1..n]$, the capacity c, and the matrix $s[0..n][0..c]$

Output : An optimal solution of the UKP instance

```
1 function UKP_DP2_SolComputation(n, w, c, s)
2     x := new array[1..n]
3     for h := 1 to n do
4         x[h] ← 0
5     end
6     j := n
7     k := c
```

Computing an optimal solution: the pseudo-code (2/2)

```
8     while k > 0 and j > 0 do
9         if s[j][k] = 0 then
10            j ← j − 1
11        else
12            x[j] ← x[j] + 1
13            k ← k − w[j]
14        end
15    end
16    return x
17 end
```

Example: computing an optimal solution (1/2)

After the execution of the DP algorithm on the reference instance, we have the following matrix s:

$s[j][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
$j=0$	0	0	0	0	0	0	0	0
$j=1$	0	0	1	1	1	1	1	1
$j=2$	0	0	0	1	0	1	0	1
$j=3$	0	0	0	0	0	1	0	1
$j=4$	0	0	0	0	0	0	0	0

Example: computing an optimal solution (2/2)

We can compute an optimal solution for a knapsack of capacity $c = 7$ as follows:

Iteration	j	k	$s[j][k]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	Action
1	4	7	0	0	0	0	0	$j \leftarrow 3$
2	3	7	1	0	0	1	0	$x[3] \leftarrow 1,\ k \leftarrow 7 - 5 = 2$
3	3	2	0	0	0	1	0	$j \leftarrow 2$
4	2	2	0	0	0	1	0	$j \leftarrow 1$
5	1	2	1	1	0	1	0	$x[1] \leftarrow 1,\ k \leftarrow 2 - 2 = 0$

An optimal solution is the vector:

$$(1, 0, 1, 0)$$

Its total weight is $w_1 + w_3 = 2 + 5 = 7 = c$, and its total profit is $p_1 + p_3 = 4 + 10 = 14$, which matches the optimal value $v(4, 7) = 14$.

Number of steps

- Let $S(n, c)$ denote the number of steps executed by `UKP_DP2` in the worst case, where n is the number of item types and c is the knapsack capacity.
- We compute $S(n, c)$ by counting how many times the lines of the pseudo-code of `UKP_DP2` are executed.
- In the worst case, line 11 is checked whenever the item type fits. Since the item weights are positive integers, this happens at most nc times.

We recall the pseudo-code of UKP_DP2 (1/2)

Algorithm: UKP_DP2

Input : A UKP instance given by n, arrays $p[1..n]$, $w[1..n]$, and capacity c

Output : The optimal value of the UKP instance

```
1 function UKP_DP2(n, p, w, c)
2 |   v := new array[0..n][0..c]
3 |   for k := 0 to c do
4 |   |   v[0][k] ← 0
5 |   end
```

We recall the pseudo-code of UKP_DP2 (2/2)

```
6   |   for j := 1 to n do
7   |   |   for k := 0 to c do
8   |   |   |   if k < w[j] then
9   |   |   |   |   v[j][k] ← v[j − 1][k]
10  |   |   |   else
11  |   |   |   |   if p[j] + v[j][k − w[j]] > v[j − 1][k] then
12  |   |   |   |   |   v[j][k] ← p[j] + v[j][k − w[j]]
13  |   |   |   |   else
14  |   |   |   |   |   v[j][k] ← v[j − 1][k]
15  |   |   |   |   end
16  |   |   |   end
17  |   |   end
18  |   end
19  |   return v[n][c]
20  end
```

Number of executions of the lines

The number of executions of the relevant lines is:

- Line 2 is executed 1 time.
- Line 3 is executed $c + 1$ times.
- Line 4 is executed $c + 1$ times.
- Line 6 is executed n times.
- Line 7 is executed $n(c + 1)$ times.
- Line 8 is executed $n(c + 1)$ times.
- Line 11 is executed at most nc times.
- One among Line 9, Line 12, and Line 14 is executed for each pair (j, k), and therefore these assignment lines are executed $n(c + 1)$ times in total.
- Line 19 is executed 1 time.

The total number of steps

Therefore, the total number of steps in the worst case is

$$1 + (c+1) + (c+1) + n + n(c+1) + n(c+1) + nc + n(c+1) + 1.$$

Thus,

$$S(n,c) = 4nc + 4n + 2c + 4.$$

Hence,

$$S(n,c) = \Theta(nc).$$

Indeed,

$$\lim_{n,c\to\infty} \frac{4nc + 4n + 2c + 4}{nc} = \lim_{n,c\to\infty} \left(4 + \frac{4}{c} + \frac{2}{n} + \frac{4}{nc}\right)$$
$$= 4 \in \mathbb{R}_{>0}.$$

The number of steps of the second DP algorithm

The DP algorithm executes

$$S(n, c) = \Theta(nc)$$

steps in the worst case.

- The version `UKP_DP2_InfoSol` stores the additional matrix s and performs only a constant number of extra operations inside the same loops.
- Therefore, its number of steps remains $\Theta(nc)$.

Summary (1/2)

The recurrence over the pairs (item type, capacity):

- $v(j, k)$ is the optimal value for a knapsack of capacity k with the item types $1, 2, \ldots, j$
- $v(0, k) = 0$; $v(j, k) = v(j-1, k)$ if $k < w_j$; otherwise $\max\{v(j-1, k),\ p_j + v(j, k - w_j)\}$
- The term $p_j + v(j, k - w_j)$ stays in the same row j: further copies of item type j are still available

Top-down and bottom-up:

- The recursion tree for $v(4, 7)$ has 32 calls, with repeated subproblems
- `UKP_DP2` fills the matrix $v[0..n][0..c]$ row by row; on the reference instance $v[4][7] = 14$

Summary (2/2)

An optimal solution:

- $s[j][k] = 1$ if one copy of item type j is used when computing $v[j][k]$, and 0 otherwise
- From $(j, k) = (n, c)$: if $s[j][k] = 0$ then $j \leftarrow j - 1$, otherwise one more copy of item type j and $k \leftarrow k - w_j$, until $k = 0$ or $j = 0$
- On the reference instance: $(1, 0, 1, 0)$, with profit 14, as with the first DP algorithm

Number of steps:

- $S(n, c) = 4nc + 4n + 2c + 4 = \Theta(nc)$, also with the matrix s: the same order as the first DP algorithm

Typical mistakes

Introduction: typical mistakes

- We list the mistakes that are made most often with the DP algorithms for the UKP
- They concern the second recurrence, the unused capacity, the reconstruction of a solution and the base case
- Knowing them helps to run the DP algorithms correctly by hand

Observation 1: Typical mistakes (1/2)

- ***Using row $j-1$ in the second recurrence.*** *The term $p_j + v(j, k - w_j)$ refers to the same row j: this is what allows several copies of item type j.*
- ***Assuming that the capacity is filled.*** *Part of the capacity may stay unused: when no item type fits, $v(k) = 0$, not $-\infty$; in the reference instance, $v(1) = 0$.*

Observation: Typical mistakes (2/2)

- ***Reconstructing past the end.*** *The reconstruction stops when the residual capacity is* 0*, or when* $s[k] = 0$ *in the first DP algorithm and* $j = 0$ *in the second one; otherwise it reads outside the arrays.*
- ***Forgetting the base case.*** $v(0) = 0$*, and* $v[0..c]$ *has* $c + 1$ *cells, not* c*.*

Summary

Typical mistakes:

- In the second recurrence, further copies of item type j come from the same row j, not from row $j-1$
- The capacity need not be filled: if no item type fits, the value is 0
- The reconstruction stops at residual capacity 0, or at $s[k] = 0$ (first DP algorithm) or $j = 0$ (second one)
- The base case $v(0) = 0$: the vector $v[0..c]$ has $c + 1$ cells

In summary

Introduction: in summary

- We collect the key ideas of the chapter
- We close with the dashboard of the reference instance

Summary (1/2)

- In the UKP an item type can be used any number of times, so the optimal value depends only on the residual capacity: $v(k) = \max_{j:\, w_j \leq k}\{p_j + v(k - w_j)\}$.
- The first DP algorithm fills a vector of $c + 1$ cells in $\Theta(nc)$ steps; the second one fills the matrix $v(j, k)$ and reuses the same row for further copies.
- An optimal solution is read backwards from the information stored, stopping when the residual capacity is zero or unused.
- The running time is pseudo-polynomial, as for the KP.

Summary (2/2)

Dashboard of the UKP example. The reference instance ($c = 7$, four item types):

Method	Lower bound	Upper bound	Origin
DP algorithm	14	14	optimal value, solution $(1, 0, 1, 0)$