DP for the 0/1 knapsack problem (1/2)

Outline

Instances and feasible solutions
- Encoding of feasible solutions

A first DP algorithm
- Number of steps
- An implementation with lists
- Number of steps with lists

Learning objectives

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

- write the recurrence $v(j, k)$ of the KP, explain why it is correct, and compute the table by hand with the top-down approach, the bottom-up approach and a single vector;
- reconstruct an optimal solution from the matrix s of the decisions;
- count the steps of the DP algorithm and explain why $\Theta(nc)$ is pseudo-polynomial, not polynomial;
- describe the states (w, p), discard the dominated ones, and run the DP algorithm on lists of states;
- exchange the roles of profits and weights, and compute the optimal value from the table $u(j, q)$.

Prerequisites

- From this module, the chapter «Principles of dynamic programming»: optimal substructure, overlapping subproblems, recurrence relations, base cases, and the top-down and bottom-up approaches.
- From the module *Combinatorial Optimization Problems*, the chapter «Knapsack problems»: the 0/1 knapsack problem (KP), the binary encoding of a solution, and the 2^n subsets of items.
- From the module *Algorithms*: how to read pseudocode, and the size of the input of an algorithm (needed to explain why $\Theta(nc)$ is pseudo-polynomial).

Where we were

- **As a problem**: the KP, its instances and the 2^n subsets (*Combinatorial Optimization Problems*, chapter «Knapsack problems»).
- **As a model**: the ILP model, its LP relaxation and the dual bounds (*Combinatorial Optimization Problems*, chapter «Formulations of knapsack problems»).
- **Now, as a table**: the DP algorithm computes the optimal value exactly, with no model and no solver.

Instances and feasible solutions

Introduction: instances and feasible solutions

- The problem of this chapter is the 0/1 knapsack problem (KP): its definition and its assumptions are on its page
- We fix a small KP instance, used in the examples of the first DP algorithm
- We recall how a feasible solution is encoded by a binary vector
- We count the subsets of items: enumerating them all quickly becomes impractical, and this is why we need a better method

Example: KP instance

We consider a knapsack of capacity $c = 5$ and a set $\{1, 2, 3\}$ of 3 items ($n = 3$), with the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	7	8
w_j	2	3	3

We refer to this KP instance as the first instance of the chapter.

The number of subsets

- The number of possible subsets of items is 2^n since, for each item, there is an independent option of inserting it or not into the knapsack.
- Some subsets may not respect the knapsack capacity. Accordingly, the number of feasible solutions is at most 2^n.
- The number of possible KP solutions grows exponentially with the number n of items.
- Therefore, enumerating all possible solutions quickly becomes impractical as n increases.

Encoding of feasible solutions (1/2)

A *feasible solution* of the KP is a subset of items whose total weight does not exceed the knapsack capacity.

A KP feasible solution can be described by the binary vector

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

where each component has the following meaning:

$$x_j = \begin{cases} 1 & \text{if item } j \text{ is selected,} \\ 0 & \text{otherwise,} \end{cases} \qquad \forall j \in \{1, 2, \ldots, n\}.$$

Encoding of feasible solutions (2/2)

Since the solution is feasible, the total weight of the selected items must not exceed the knapsack capacity:

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

The total profit of a feasible solution is

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

We now list all the subsets of items of the first instance, first with a binary tree, then with a drawing and a table.

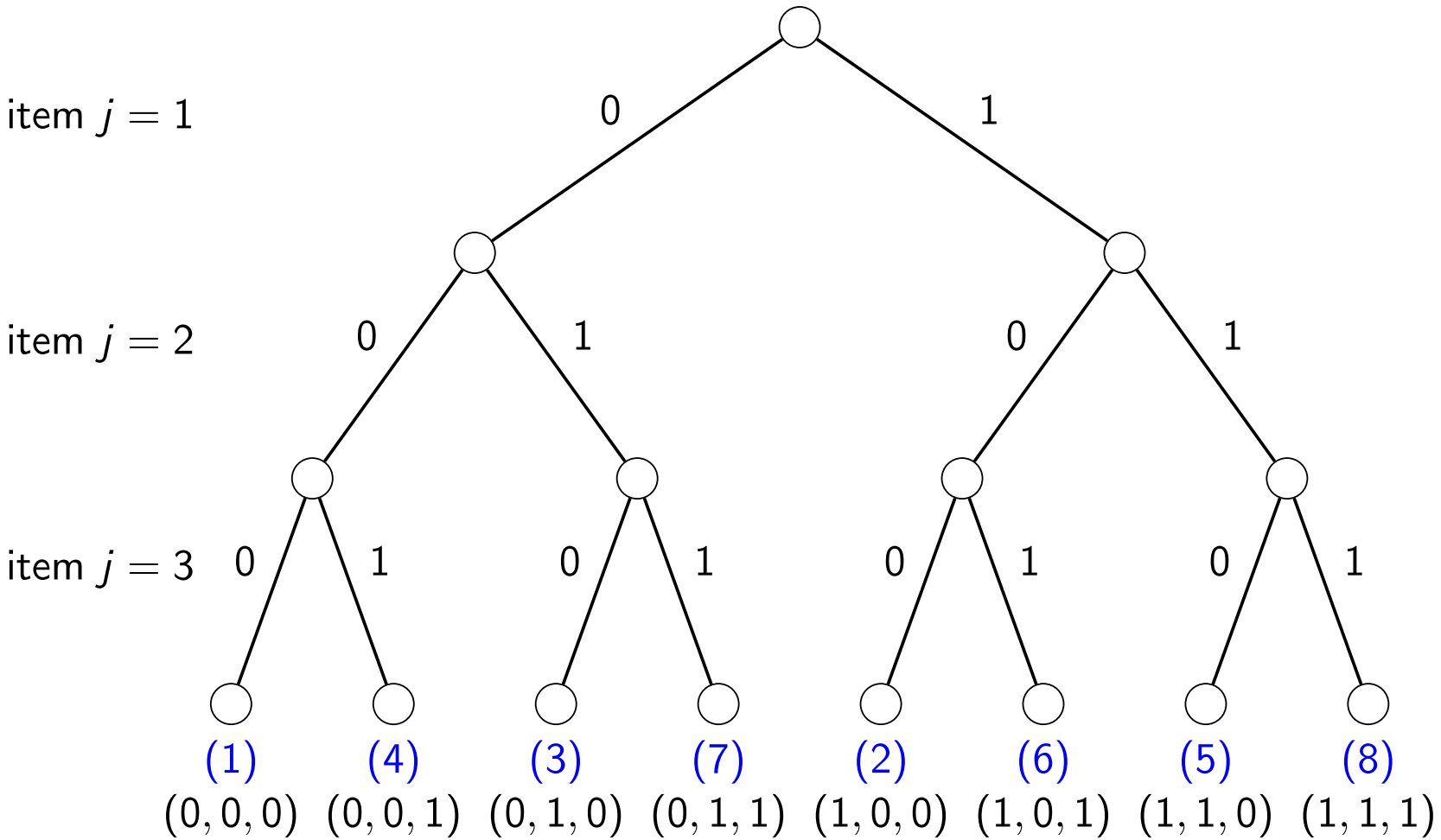
Example: enumeration of all possible KP solutions (1/6)

We consider the first instance, with a knapsack of capacity $c = 5$ and the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	7	8
w_j	2	3	3

- There are $2^3 = 8$ possible subsets of items, corresponding to the different ways of selecting or not selecting the items.
- In the binary tree of the next slide, each level corresponds to an item: an edge labeled 1 means that the item is selected, while an edge labeled 0 means that it is not selected.
- Each leaf represents one complete subset of items and reports the corresponding vector.

Example: enumeration of all possible KP solutions (2/6)

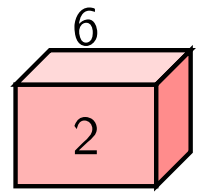


Example: enumeration of all possible KP solutions (3/6)

- The next two slides show the subsets associated with the leaves of the tree, through the labels $(1), (2), \ldots, (8)$.
- For each selected item, the number written inside the parallelepiped is its weight, while the number above it is its profit.
- The width of each parallelepiped is proportional to the item weight.
- Subsets whose total weight exceeds the knapsack capacity are marked as not feasible.

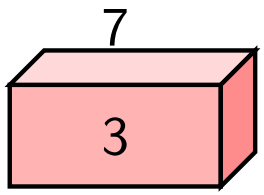
Example: enumeration of all possible KP solutions (4/6)

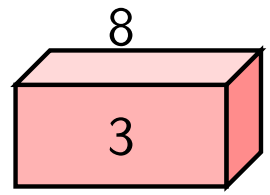
empty subset



(1)
total weight = 0
total profit = 0

(2)
total weight = 2
total profit = 6

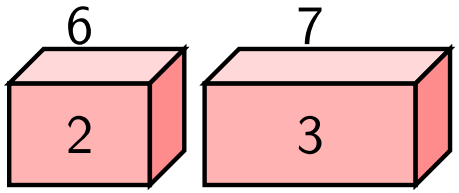




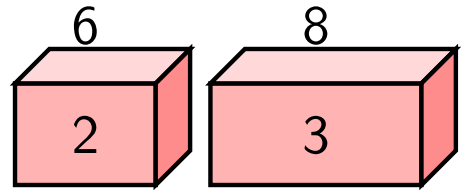
(3)
total weight = 3
total profit = 7

(4)
total weight = 3
total profit = 8

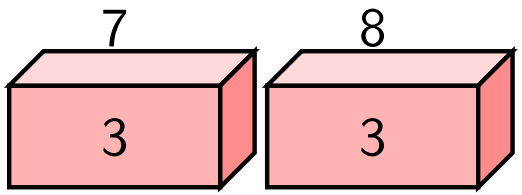
Example: enumeration of all possible KP solutions (5/6)



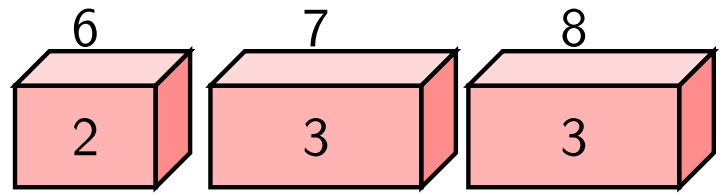
(5)
total weight = 5
total profit = 13



(6)
total weight = 5
total profit = 14



(7)
total weight = 6
total profit = 15
not feasible



(8)
total weight = 8
total profit = 21
not feasible

Example: enumeration of all possible KP solutions (6/6)

Label	Solution vector	Total weight	Total profit	Feasible?
(1)	$(0,0,0)$	0	0	yes
(2)	$(1,0,0)$	2	6	yes
(3)	$(0,1,0)$	3	7	yes
(4)	$(0,0,1)$	3	8	yes
(5)	$(1,1,0)$	$2+3=5$	$6+7=13$	yes
(6)	$(1,0,1)$	$2+3=5$	$6+8=14$	yes
(7)	$(0,1,1)$	$3+3=6$	$7+8=15$	no
(8)	$(1,1,1)$	$2+3+3=8$	$6+7+8=21$	no

Summary

The first instance:

- A knapsack of capacity $c = 5$ and $n = 3$ items, with profits $6, 7, 8$ and weights $2, 3, 3$
- It is used in the examples of the first DP algorithm

Feasible solutions:

- A feasible solution is a binary vector $(x_1, x_2, \ldots, x_n)$ with $\sum_{j=1}^{n} w_j x_j \leq c$
- There are 2^n subsets of items: on the first instance, 6 of the 8 subsets are feasible
- Enumerating all subsets quickly becomes impractical as n increases

A first DP algorithm

Introduction: a first DP algorithm

- We define the subproblems $v(j, k)$: the first j items and a knapsack of capacity k
- We write their recurrence and verify the optimal substructure and the overlapping subproblems
- We compute the optimal value top-down, bottom-up and with a single vector, and then an optimal solution
- We count the steps of the DP algorithm, and we see a different implementation with lists of states
- This DP algorithm computes the optimal value exactly, with no model and no solver

Step 1: computation of the optimal value

The subproblems $v(j, k)$

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 KP subinstance given by:

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

The recurrence of $v(j,k)$

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

$$v(j,k) = \begin{cases} 0 & \text{if } j = 0 \text{ or } k = 0, \\ v(j-1,k) & \text{if } 1 \leq j \leq n, \\ & 1 \leq k \leq c, \text{ and } w_j > k, \\ \max\left\{ v(j-1,k), \right. & \text{if } 1 \leq j \leq n, \\ \left. \quad p_j + v(j-1,k-w_j) \right\} & 1 \leq k \leq c,\ w_j \leq k. \end{cases} \tag{1}$$

Reading the recurrence (1/2)

- The recursive decomposition process of formula (1) generates at most 2 **independent subproblems**, corresponding to whether or not to select item j for a knapsack of capacity k.
- If $w_j > k$, item j cannot be selected, and therefore the only feasible option is not to select it.
- Here, $v(0, k) = 0$ since no item is available, and $v(j, 0) = 0$ since no capacity is available.

Reading the recurrence (2/2)

- If $w_j \leq k$, the optimal value $v(j, k)$ is obtained by trying the two possible options, selecting or not selecting item j, and selecting the one that maximizes the profit.
- The corresponding values are $p_j + v(j-1, k-w_j)$ and $v(j-1, k)$, respectively. If $w_j > k$, item j cannot be selected, and therefore $v(j, k) = v(j-1, k)$.
- Hence, an optimal solution for the subinstance with capacity k and available items $\{1, 2, \ldots, j\}$ is built by combining the chosen decision on item j with an optimal solution of the corresponding residual subproblem.

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

Overlapping subproblems

- Different sequences of decisions on the items, possibly made for different values of j and k, can lead to the same subproblem.
- Therefore the same value $v(j, k)$ may need to be computed several times during the recursive evaluation of formula (1).

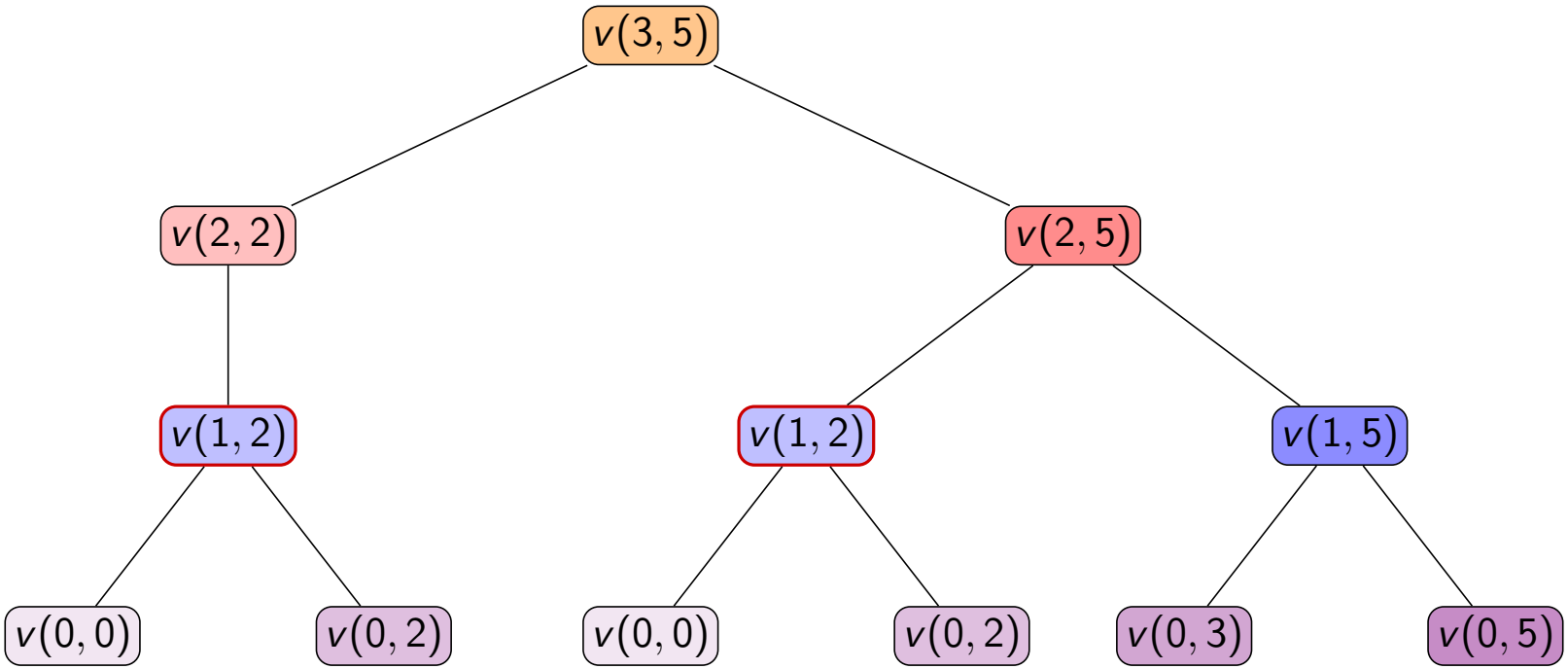
$\rightsquigarrow$ **Overlapping Subproblems verified!**

The top-down approach

- A **first method** to compute the optimal value $v(n, c)$ of the KP instance is the **top-down approach**.
- The method starts from the call with $j = n$ and $k = c$ and evaluates formula (1) recursively: each call considers item j and either does not select it, or also considers selecting it when it fits.
- The recursion stops when no items are left or when the residual capacity is equal to 0.
- 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

We consider the first instance. This is the recursion tree of the top-down approach to compute $v(3, 5)$:



The function is called 12 times, and the same subproblem is solved more than once: $v(1, 2)$ is called 2 times (shown with a red border), and so are $v(0, 2)$ and $v(0, 0)$.

Order of computation

- The value $v(j, k)$ can be computed once the values required by the right-hand side of formula (1) are known.
- In particular, $v(j-1, k)$ is needed when item j is not selected and, if $w_j \leq k$, $v(j-1, k-w_j)$ is needed when item j is selected.
- Therefore, for a fixed value of j, all values $v(j, k)$ can be computed by considering the capacity values k in increasing order.

Example: computing the values of the v function (1/2)

We consider the first instance, with a knapsack of capacity $c = 5$ and the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	7	8
w_j	2	3	3

The values of the function v are computed as follows:

- The base cases give $v(0, k) = 0$ for all $k \geq 0$.
- Once we have computed the base cases, we can compute $v(1, 2)$:

$$v(1, 2) = \max\{v(0, 2),\ v(0, 0) + p_1\} = \max\{0,\ 0 + 6\} = 6$$

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

- We can also compute $v(1,5)$:

$$v(1,5) = \max\{v(0,5),\ v(0,3) + p_1\} = \max\{0,\ 0+6\} = 6$$

- Since $w_2 = 3 > k = 2$, item 2 cannot fit, so:

$$v(2,2) = v(1,2) = 6$$

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

$$v(2,5) = \max\{v(1,5),\ v(1,2) + p_2\} = \max\{6,\ 6+7\} = 13$$

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

$$v(3,5) = \max\{v(2,5),\ v(2,2) + p_3\} = \max\{13,\ 6+8\} = 14$$

Lookup table and bottom-up approach (1/2)

- 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, we can solve the subproblems in increasing order of j and, for each fixed value of j, in increasing order of capacity.
- We start from the base cases $v(0, k) = 0$ and $v(j, 0) = 0$ and build 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 KP instance.

Lookup table and bottom-up approach (2/2)

- A **second method** to compute the optimal value $v(n, c)$ of the KP instance is the **bottom-up approach**.
- The method starts from the base cases and evaluates formula (1) 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 KP instance is obtained.

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

Algorithm: DP

Input : A KP instance given by the number n of items, arrays $p[1..n]$ and $w[1..n]$ of item profits and weights, and the knapsack capacity c

Output : The optimal value of the KP instance

```
1 function DP(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
```

Algorithm: DP (continued)

```
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 v[j − 1][k − w[j]] + p[j] > v[j − 1][k] then
12                    v[j][k] ← v[j − 1][k − w[j]] + p[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 first loop initializes the values $v[0][k] = 0$ for all capacities $k \in \{0, 1, \ldots, c\}$, corresponding to the case in which no item is available.
- The outer loop, over items $j \in \{1, 2, \ldots, n\}$, computes the rows of the matrix in increasing order of j.
- The inner loop, over $k \in \{0, 1, \ldots, c\}$, computes the optimal value $v[j][k]$ for each capacity k.

How the DP algorithm works (2/2)

- If item j does not fit, then $v[j][k] \leftarrow v[j-1][k]$.
- Otherwise, the DP algorithm compares the value obtained by not selecting item j, namely $v[j-1][k]$, with the value obtained by selecting item j, namely $p[j] + v[j-1][k-w[j]]$.
- Finally, the optimal value of the KP instance is returned as $v[n][c]$.

Example: execution of the DP algorithm (1/4)

We consider the first instance, with a knapsack of capacity $c = 5$ and the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	7	8
w_j	2	3	3

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.

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	$v[0][5] \leftarrow 0$

Example: execution of the DP algorithm (2/4)

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

Example: execution of the DP algorithm (3/4)

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

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

After the execution of the DP algorithm, this is the matrix v:

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

From the matrix to a single vector

- After the execution of the DP algorithm, the matrix v contains the optimal value $v[j][k] = v(j, k)$ for every item $j \in \{0, 1, \ldots, n\}$ and every capacity value $k \in \{0, 1, \ldots, c\}$.
- The DP algorithm computes the row j of the matrix v using only the row $j - 1$.
- Therefore, when only the optimal value of the KP instance is required, it is not necessary to store the whole matrix: a single vector $v[0..c]$ is sufficient, which uses $c + 1$ cells instead of the $(n + 1)(c + 1)$ cells of the matrix.

A single vector (1/2)

- The vector is initialized with the base cases $v[k] = 0$ for all $k \in \{0, 1, \ldots, c\}$, which correspond to the row $j = 0$ of the matrix.
- Then, for each item $j \in \{1, 2, \ldots, n\}$, the vector is updated so that it contains the row j of the matrix.
- The order in which the cells are updated is essential. When the cell $v[k]$ is updated for item j, the value $v[k - w_j]$ must still be the one of the row $j - 1$, that is $v(j - 1, k - w_j)$.

A single vector (2/2)

- Since $k - w_j < k$, this is guaranteed if the capacity values are considered in **decreasing** order $k = c, c-1, \ldots, w_j$: when the cell k is updated, the cell $k - w_j$ has not been updated yet during the current pass.
- If the capacity values were considered in increasing order, the value $v[k - w_j]$ would already be the one of the row j, that is $v(j, k - w_j)$, and item j could be selected more than once, which is not allowed in the KP.
- For every capacity value $k < w_j$, item j does not fit and therefore $v[k]$ must keep the value $v(j-1, k)$: no update is performed, and the inner loop can stop at $k = w_j$.

This is the pseudo-code of the **DP algorithm using a single vector**:

Algorithm: DP_Vector

Input : A KP instance given by the number n of items, arrays $p[1..n]$ and $w[1..n]$ of item profits and weights, and the knapsack capacity c

Output : The optimal value of the KP instance

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

Algorithm: DP_Vector (continued)

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

How DP_Vector works (1/2)

- The vector $v[0..c]$ is the **lookup table**: at the end of the pass for item j, it stores the row j of the matrix v of the DP algorithm.
- The first loop initializes the values $v[k] = 0$ for all capacities $k \in \{0, 1, \ldots, c\}$, corresponding to the case in which no item is available.
- The outer loop, over items $j \in \{1, 2, \ldots, n\}$, performs one pass of the vector for each item.

How DP_Vector works (2/2)

- The inner loop, over $k \in \{w[j], w[j]+1, \ldots, c\}$, considers the capacity values in decreasing order, so that the cells that are still needed with their previous value are not yet modified.
- The condition compares the value obtained by not selecting item j, namely the current value $v[k]$, with the value obtained by selecting item j, namely $p[j] + v[k - w[j]]$. The cell is updated only when selecting item j is strictly better.
- Finally, the optimal value of the KP instance is returned as $v[c]$.

Example: execution of DP using a single vector (1/3)

We consider the first instance, with a knapsack of capacity $c = 5$ and the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	7	8
w_j	2	3	3

The first loop initializes $v[k] = 0$ for all $k \in \{0, 1, \ldots, 5\}$. For each step of the two nested loops, the table shows the values of j and k, the cell $v[k]$ after the step, and the operation performed.

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

Example: execution of DP using a single vector (2/3)

j	k	$v[k]$	Action
2	5	13	$p[2]+v[2]=7+6>v[5]=6$: $v[5]\leftarrow 13$
2	4	7	$p[2]+v[1]=7+0>v[4]=6$: $v[4]\leftarrow 7$
2	3	7	$p[2]+v[0]=7+0>v[3]=6$: $v[3]\leftarrow 7$
3	5	14	$p[3]+v[2]=8+6>v[5]=13$: $v[5]\leftarrow 14$
3	4	8	$p[3]+v[1]=8+0>v[4]=7$: $v[4]\leftarrow 8$
3	3	8	$p[3]+v[0]=8+0>v[3]=7$: $v[3]\leftarrow 8$

Example: execution of DP using a single vector (3/3)

This is the content of the vector v after the initialization and after each pass of the outer loop:

$v[k]$	0	1	2	3	4	5
initialization	0	0	0	0	0	0
pass $j = 1$	0	0	6	6	6	6
pass $j = 2$	0	0	6	7	7	13
pass $j = 3$	0	0	6	8	8	14

Each pass transforms the vector v from the row $j - 1$ into the row j of the matrix v computed by the DP algorithm on the same instance. The optimal value of the KP instance is $v[5] = 14$.

The vector computes only the optimal value

- The vector version computes only the optimal value of the KP instance.
- The information needed to construct an optimal solution, which is described in Step 2, records the decision taken for every item j and every capacity value k, and therefore cannot be stored in a single vector.

Step 2: computation of an optimal solution

The matrix s of the decisions

- To compute an optimal solution, i.e., the binary vector $(x_1, x_2, \ldots, x_n)$, it is necessary to store some additional information.
- For each item j and capacity value k, in addition to the optimal value $v[j][k]$, we store $s[j][k]$, which tells whether item j is selected in an optimal solution for the subproblem restricted to items $\{1, 2, \ldots, j\}$ and capacity k.
- More precisely, $s[j][k]$ is equal to 1 if item j is selected when computing the value $v[j][k]$, and it is equal to 0 otherwise.

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

Algorithm: DP_InfoSol

Input : A KP instance given by the number n of items, arrays $p[1..n]$ and $w[1..n]$ of item profits and weights, and the knapsack capacity c

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

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

Algorithm: DP_InfoSol (continued)

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

Algorithm: DP_InfoSol (continued)

```
24      return v[n][c], s
25  end
```

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

Example: computation of the additional information (1/4)

We consider the first instance, with a knapsack of capacity $c = 5$ and the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	7	8
w_j	2	3	3

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.

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	$s[0][5] \leftarrow 0$

Example: computation of the additional information (2/4)

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

Example: computation of the additional information (3/4)

j	k	$s[j][k]$	Action
3	0	0	$0 < 3$: $s[3][0] \leftarrow 0$
3	1	0	$1 < 3$: $s[3][1] \leftarrow 0$
3	2	0	$2 < 3$: $s[3][2] \leftarrow 0$
3	3	1	$p[3] + v[2][0] = 8 + 0 > v[2][3] = 7$: $s[3][3] \leftarrow 1$
3	4	1	$p[3] + v[2][1] = 8 + 0 > v[2][4] = 7$: $s[3][4] \leftarrow 1$
3	5	1	$p[3] + v[2][2] = 8 + 6 > v[2][5] = 13$: $s[3][5] \leftarrow 1$

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

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

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

Reconstruction of an optimal solution

- Once the matrix s has been computed, an optimal solution can be reconstructed by scanning the items from $j = n$ down to $j = 1$, starting from capacity $k = c$ and $x_j = 0$ for all $j \in \{1, 2, \ldots, n\}$:

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

- This yields the vector $(x_1, x_2, \ldots, x_n)$, where x_j is equal to 1 if item j is selected in an optimal solution, and 0 otherwise.

Algorithm: DP_Reconstruct

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

Output : An optimal solution of the KP instance

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

Algorithm: DP_Reconstruct (continued)

```
7     for j := n downto 1 do
8         if s[j][k] = 1 then
9             x[j] ← 1
10            k ← k − w[j]
11        end
12    end
13    return x
14 end
```

Example: computing an optimal solution (1/2)

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

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

Example: computing an optimal solution (2/2)

We can reconstruct an optimal solution by starting from $j = 3$ and $k = 5$ as follows:

Iteration	j	k	$s[j][k]$	$x[1]$	$x[2]$	$x[3]$	Action
–	–	–	–	0	0	0	$k = 5$
1	3	5	1	0	0	1	$k \leftarrow 5 - 3 = 2$
2	2	2	0	0	0	1	no update
3	1	2	1	1	0	1	$k \leftarrow 2 - 2 = 0$

Thus, the vector

$$(1, 0, 1)$$

is an optimal solution. The corresponding optimal value is $6 + 8 = 14$.

How many steps does the DP algorithm execute?

We count how many times each line of the pseudo-code of `DP` is executed.

Number of steps of DP (1/3)

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

Number of steps of DP (2/3)

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.

Number of steps of DP (3/3)

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 three versions

The DP algorithm executes

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

steps in the worst case.

- The version `DP_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 $\Theta(nc)$.
- The version `DP_Vector` uses a single vector instead of the matrix and executes the same two nested loops, whose inner loop performs at most $c + 1$ iterations for each item. Therefore, its number of steps remains $\Theta(nc)$, while the number of cells of the lookup table decreases from $(n + 1)(c + 1)$ to $c + 1$.

An implementation with lists

Step 1: computation of the optimal value

Why lists?

- The DP algorithm of this section stores one value for every pair given by an item and a capacity value, that is $(n+1)(c+1)$ values in total.
- Each row of the matrix v is non-decreasing in the capacity value and changes only at a few capacity values: every other cell repeats the value of the cell on its left.
- We now see a **different implementation of the same DP algorithm**. The recursive decomposition and the optimal substructure are the ones of formula (1): what changes is the data structure.
- Instead of a whole row of the matrix, only the pairs at which the row changes are stored, that is only the pairs that are actually obtained by a subset of items. These pairs are kept in a list, one list for each item.

States

A **state** is a pair (w, p), where w is the total weight and p is the total profit of a subset of items whose total weight does not exceed the knapsack capacity c.

- The empty subset gives the state $(0, 0)$. A subset of the items $\{1, 2, \ldots, j\}$ gives a state obtained by summing the weights and the profits of its items.
- Two different subsets can give the same state. In this case the two subsets are interchangeable: both occupy the same weight in the knapsack and both give the same profit, and therefore it is enough to keep one state.
- More generally, a state can be discarded whenever another state is at least as good in both components.

Dominated states

Given two states (w, p) and (w', p') with $(w, p) \neq (w', p')$, the state (w, p) **dominates** the state (w', p') if

$$w \leq w' \qquad \text{and} \qquad p \geq p'.$$

A state is **non-dominated** if no other state dominates it.

- A dominated state can be discarded. Indeed, let (w, p) dominate (w', p') and let S' be a subset of the remaining items that can be added to (w', p') without exceeding the capacity.
- Since $w \leq w'$, the same subset S' can also be added to (w, p), and since $p \geq p'$ the resulting total profit is not smaller. Therefore no optimal solution is lost.

The lists L_j

For any $j \in \{0, 1, \ldots, n\}$, let L_j be the list of the non-dominated states of the KP subinstance given by:

1. a knapsack of capacity c,
2. a set $\{1, 2, \ldots, j\}$ of available items,
3. for each item $h \in \{1, 2, \ldots, j\}$, a profit p_h and a weight w_h.

The states of L_j are sorted by strictly increasing weight.

The recurrence of L_j

The recursive formula to compute L_j reads as follows:

$$L_j = \begin{cases} \{(0,0)\} & \text{if } j = 0, \\ \mathrm{nd}\big(L_{j-1} \cup M_j\big) & \text{if } 1 \leq j \leq n, \end{cases} \tag{2}$$

where $\mathrm{nd}(S)$ denotes the list of the non-dominated states of a set S of states, and

$$M_j = \big\{ (w + w_j,\ p + p_j) \ :\ (w,p) \in L_{j-1},\ w + w_j \leq c \big\}.$$

Reading the recurrence of L_j (1/2)

- The recursive decomposition process of formula (2) generates at most 2 **independent subproblems** for each state of L_{j-1}, corresponding to whether or not to select item j.
- If item j is not selected, the state is unchanged and belongs to L_{j-1}.
- If item j is selected, the state becomes $(w + w_j, p + p_j)$ and belongs to M_j; this option is available only when $w + w_j \leq c$.
- Here, $L_0 = \{(0, 0)\}$ since the only subset of an empty set of items is the empty subset.

Reading the recurrence of L_j (2/2)

- Hence, every non-dominated state reachable with the items $\{1, 2, \ldots, j\}$ is built by combining the chosen decision on item j with a non-dominated state reachable with the items $\{1, 2, \ldots, j-1\}$.

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

- Different subsets of the items $\{1, 2, \ldots, j\}$ can give the same state, and a state can also be dominated by another one. In both cases a single state is kept in L_j, and all the subsets that lead to it are treated at once in the following iterations.

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

The optimal value from the last list

The optimal value of the KP instance is the largest profit stored in the last list:

$$\max \left\{ p \ : \ (w, p) \in L_n \right\}. \tag{3}$$

- In a list of non-dominated states sorted by strictly increasing weight, the profits are also strictly increasing: if a state had a weight larger than another one and a profit not larger, it would be dominated.
- Therefore the maximum of formula (3) is the profit of the last state of L_n.

The lists and the rows of the matrix v

The list L_j holds the information of the row j of the matrix v in compressed form:

$$v[j][k] = \max\left\{ p \ : \ (w,p) \in L_j, \ w \leq k \right\} \qquad \forall k \in \{0,1,\ldots,c\}.$$

Every state (w,p) of L_j is a capacity value at which the row increases, and p is the new value.

Example: the lists of states (1/3)

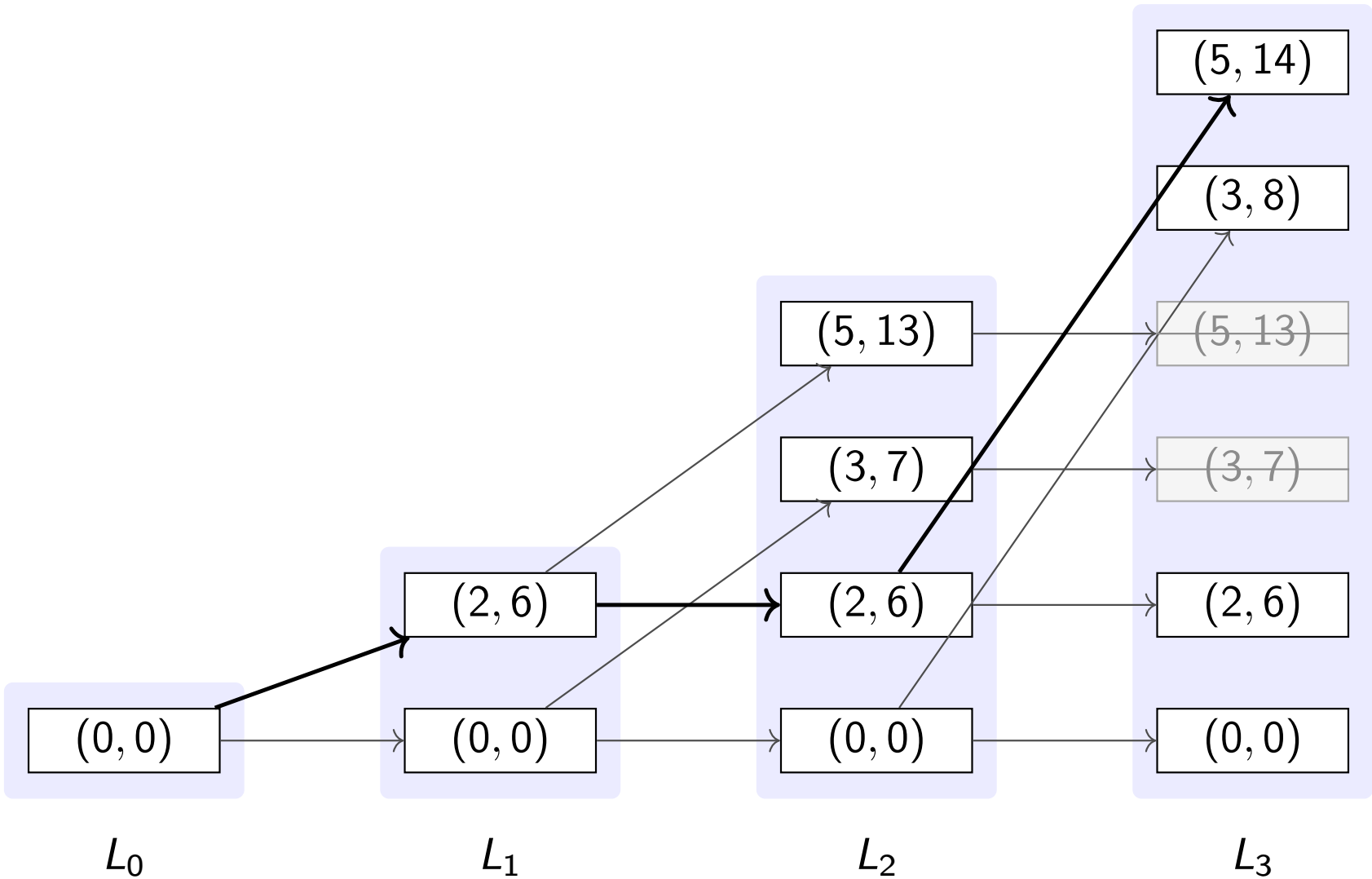
We consider the first instance, with a knapsack of capacity $c = 5$ and the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	7	8
w_j	2	3	3

In the figure of the next slide:

- each shaded band collects the states of one list;
- an arc entering the band of L_j corresponds to the decision taken on item j: a horizontal arc means that item j is not selected, while a slanted arc means that item j is selected;
- the thick arcs give the states of an optimal solution.

Example: the lists of states (2/3)



Example: the lists of states (3/3)

- The two crossed states of the last band are dominated and are therefore discarded: $(3, 7)$ is dominated by $(3, 8)$ and $(5, 13)$ is dominated by $(5, 14)$.
- Selecting item 3 from the states $(3, 7)$ and $(5, 13)$ of L_2 would give the total weights 6 and 8, which exceed the knapsack capacity $c = 5$.
- Therefore no arc leaves these two states towards the states of M_3.

Computing the lists

- The lists are computed for increasing values of $j \in \{1, 2, \ldots, n\}$, starting from $L_0 = \{(0, 0)\}$: each list L_j is computed from the already-computed list L_{j-1}, until the list L_n is obtained.
- The list L_{j-1} and the list M_j are both sorted by increasing weight, because M_j is obtained from L_{j-1} by adding the same weight w_j to every state.
- Therefore the union of formula (2) is computed by scanning the two lists at the same time, as in the merge of two sorted lists.

The scanning order

To describe the scanning order we use the following convention.

The state (w, p) **precedes** the state (w', p') if

$$w < w' \qquad \text{or} \qquad (w = w' \text{ and } p > p').$$

- If the states are scanned in this order, a state is non-dominated if and only if its profit is strictly larger than the largest profit of the states already scanned.
- Indeed, every state already scanned has a weight not larger, and therefore it dominates the current state as soon as its profit is not smaller.

This is the pseudo-code of the **DP algorithm using lists**:

Algorithm: DP_Lists

Input : A KP instance given by the number n of items, arrays $p[1..n]$ and $w[1..n]$ of item profits and weights, and the knapsack capacity c

Output : The optimal value of the KP instance

```
1 function DP_Lists(n, p, w, c)
2 |   L := new list containing the single state (0, 0)
```

Algorithm: DP_Lists (continued)

```
    for j := 1 to n do
        M := new empty list
        for h := 1 to |L| do
            if L[h].w + w[j] ≤ c then
                append (L[h].w + w[j], L[h].p + p[j]) to M
            end
        end
        L ← Merge(L, M)
    end
    return L[|L|].p
end
```

This is the pseudo-code of the procedure that merges two lists and discards the dominated states:

Algorithm: Merge

Input : Two lists L and M of states, both sorted by increasing weight

Output : The list of the non-dominated states of $L \cup M$, sorted by increasing weight

```
1 function Merge(L, M)
2     R := new empty list
3     h := 1
4     k := 1
5     π := −1
```

Algorithm: Merge (continued)

```
6     while h ≤ |L| or k ≤ |M| do
7         if k > |M| or (h ≤ |L| and L[h] precedes M[k]) then
8             (a, b) ← L[h]
9             h ← h + 1
10        else
11            (a, b) ← M[k]
12            k ← k + 1
13        end
14        if b > π then
15            append (a, b) to R
16            π ← b
17        end
18    end
19    return R
20 end
```

How DP_Lists works (1/2)

- The list L is the **lookup table**: at the end of the iteration for item j, it stores the list L_j.
- The list L is initialized with the single state $(0, 0)$, which is the list L_0.
- The outer loop, over items $j \in \{1, 2, \ldots, n\}$, computes one list for each item.
- The inner loop builds the list M of the states obtained by selecting item j. A state is added to M only when the resulting total weight does not exceed the knapsack capacity.

How DP_Lists works (2/2)

- The procedure `Merge` scans the two lists at the same time and keeps a state only when its profit is strictly larger than the largest profit already appended, which is stored in the auxiliary variable π.
- The initial value $\pi = -1$ guarantees that the first scanned state is always appended.
- Since the states of L are sorted by strictly increasing weight and strictly increasing profit, the largest profit is the one of the last state. Therefore the optimal value of the KP instance is returned as $L[|L|].p$.
- After the execution of the DP algorithm, the list L contains the non-dominated states L_n of the KP instance, sorted by strictly increasing weight and strictly increasing profit.

Example: execution of the DP algorithm with lists (1/3)

We consider the first instance, with a knapsack of capacity $c = 5$ and the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	7	8
w_j	2	3	3

For each item j, the table shows the list M_j of the states obtained by selecting item j and the resulting list L_j.

j	M_j	L_j
0	–	$\{(0,0)\}$
1	$\{(2,6)\}$	$\{(0,0),(2,6)\}$
2	$\{(3,7),(5,13)\}$	$\{(0,0),(2,6),(3,7),(5,13)\}$
3	$\{(3,8),(5,14)\}$	$\{(0,0),(2,6),(3,8),(5,14)\}$

Example: execution of the DP algorithm with lists (2/3)

For each state scanned by `Merge`, the table shows the value of j, the state, the list it is taken from, and the operation performed.

j	state	from	Action
1	$(0,0)$	L_0	$0 > \pi = -1$: append
1	$(2,6)$	M_1	$6 > \pi = 0$: append
2	$(0,0)$	L_1	$0 > \pi = -1$: append
2	$(2,6)$	L_1	$6 > \pi = 0$: append
2	$(3,7)$	M_2	$7 > \pi = 6$: append
2	$(5,13)$	M_2	$13 > \pi = 7$: append

Example: execution of the DP algorithm with lists (3/3)

j	state	from	Action
3	$(0, 0)$	L_2	$0 > \pi = -1$: append
3	$(2, 6)$	L_2	$6 > \pi = 0$: append
3	$(3, 8)$	M_3	$8 > \pi = 6$: append
3	$(3, 7)$	L_2	$7 \leq \pi = 8$: discard (dominated)
3	$(5, 14)$	M_3	$14 > \pi = 8$: append
3	$(5, 13)$	L_2	$13 \leq \pi = 14$: discard (dominated)

The last state of L_3 is $(5, 14)$, and therefore the optimal value of the KP instance is 14.

An implementation with lists

Step 2: computation of an optimal solution

The additional information for the lists (1/2)

- To compute an optimal solution, i.e., the binary vector $(x_1, x_2, \ldots, x_n)$, it is necessary to store some additional information.
- The list L_n alone gives the optimal value, but it does not say which items produce the state that attains it.
- For each item j and each position $h \in \{1, 2, \ldots, |L_j|\}$, in addition to the state $L_j[h]$, we store two values:
  - $\sigma_j[h]$, which is equal to 1 if item j is selected in the state $L_j[h]$, and is equal to 0 otherwise;
  - $\rho_j[h]$, which is the position in L_{j-1} of the state from which $L_j[h]$ is obtained.

The additional information for the lists (2/2)

- A state taken from L_{j-1} gives $\sigma_j[h] = 0$, while a state taken from M_j gives $\sigma_j[h] = 1$.
- In both cases $\rho_j[h]$ is the position of the state of L_{j-1} that generated it.
- The DP algorithm must also keep all the lists $L_0, L_1, \ldots, L_n$, and not only the last one, because the reconstruction goes backwards through them.

Example: computation of the additional information (1/2)

We consider the first instance. For each item j and each position h, the table shows the state $L_j[h]$ and the stored values $\sigma_j[h]$ and $\rho_j[h]$.

j	h	$L_j[h]$	$\sigma_j[h]$	$\rho_j[h]$
1	1	$(0,0)$	0	1
1	2	$(2,6)$	1	1
2	1	$(0,0)$	0	1
2	2	$(2,6)$	0	2
2	3	$(3,7)$	1	1
2	4	$(5,13)$	1	2

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

j	h	$L_j[h]$	$\sigma_j[h]$	$\rho_j[h]$
3	1	$(0,0)$	0	1
3	2	$(2,6)$	0	2
3	3	$(3,8)$	1	1
3	4	$(5,14)$	1	2

For instance, the state $L_3[4] = (5,14)$ has $\sigma_3[4] = 1$ and $\rho_3[4] = 2$: it is obtained by selecting item 3 from the state $L_2[2] = (2,6)$, since $2 + w_3 = 2 + 3 = 5$ and $6 + p_3 = 6 + 8 = 14$.

Reconstruction with the lists

- Once the lists and the stored values have been computed, an optimal solution is reconstructed by scanning the items from $j = n$ down to $j = 1$, starting from the last position $h = |L_n|$, which holds the state of largest profit:

$$x_j \leftarrow \sigma_j[h], \quad h \leftarrow \rho_j[h] \quad \text{for } j = n, n-1, \ldots, 1.$$

- This yields the vector $(x_1, x_2, \ldots, x_n)$, where x_j is equal to 1 if item j is selected in an optimal solution, and 0 otherwise.

Algorithm: DP_Lists_Reconstruct

Input : The number n of items, the lists $L_0, L_1, \ldots, L_n$, and the stored values σ and ρ

Output : An optimal solution of the KP instance

```
1 function DP_Lists_Reconstruct(n, L, σ, ρ)
2 |   x := new array[1..n]
3 |   for j := 1 to n do
4 |   |   x[j] ← 0
5 |   end
6 |   h := |L_n|
```

Algorithm: DP_Lists_Reconstruct (continued)

```
7     for j := n downto 1 do
8         x[j] ← σ_j[h]
9         h ← ρ_j[h]
10    end
11    return x
12 end
```

Example: computing an optimal solution with the lists (1/2)

After the execution of the DP algorithm on the first instance, the last list is

$$L_3 = \{(0,0),(2,6),(3,8),(5,14)\}$$

and the optimal value is the profit of its last state, that is 14.

Example: computing an optimal solution with the lists (2/2)

We can reconstruct an optimal solution by starting from $j = 3$ and $h = |L_3| = 4$ as follows:

Iteration	j	h	$L_j[h]$	$\sigma_j[h]$	$\rho_j[h]$	$x[1]$	$x[2]$	$x[3]$	Action
–	–	–	–	–	–	0	0	0	$h = 4$
1	3	4	$(5, 14)$	1	2	0	0	1	$h \leftarrow 2$
2	2	2	$(2, 6)$	0	2	0	0	1	$h \leftarrow 2$
3	1	2	$(2, 6)$	1	1	1	0	1	$h \leftarrow 1$

Thus, the vector

$$(1, 0, 1)$$

is an optimal solution. The corresponding optimal value is $6 + 8 = 14$.

Number of steps of DP_Lists (1/2)

- Let $S(n, c)$ denote the number of steps executed by `DP_Lists` in the worst case, where n is the number of items and c is the knapsack capacity.
- The states of a list have strictly increasing weights, and every weight belongs to $\{0, 1, \ldots, c\}$: therefore $|L_j| \leq c + 1$ for all $j \in \{0, 1, \ldots, n\}$, and $|M_j| \leq |L_{j-1}|$.
- For each item, `DP_Lists` scans L_{j-1} once to build M_j and then calls `Merge`, which scans $|L_{j-1}| + |M_j| \leq 2(c + 1)$ states and performs a constant number of operations for each of them.
- Therefore the number of steps executed for each item is $\Theta(c)$ in the worst case, and

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

Number of steps of DP_Lists (2/2)

- The three implementations have the same order of growth, but they do not store the same amount of information.
- The matrix v always has $(n+1)(c+1)$ cells, while the lists contain only the states that are actually reachable: a list reaches the bound $c+1$ only when every capacity value is the total weight of a subset of the items.
- On the first instance the lists contain 1, 2, 4, and 4 states, against the 24 cells of the matrix v.
- The list L_j also satisfies $|L_j| \leq 2^j$, since the items $\{1, 2, \ldots, j\}$ have 2^j subsets. Therefore

$$|L_j| \leq \min\left\{2^j,\ c+1\right\},$$

and the number of states stays small while j is small, whatever the knapsack capacity.

Summary (1/2)

The recurrence $v(j, k)$:

- $v(j, k)$ is the optimal value with the items $\{1, 2, \ldots, j\}$ and capacity k
- It chooses between leaving item j out and taking it; it has the optimal substructure and overlapping subproblems

Computing the optimal value:

- Top-down, the same subproblems are solved more than once
- Bottom-up, the matrix v is filled row by row in $\Theta(nc)$ steps
- A single vector, scanned with decreasing capacities, gives the optimal value with $c + 1$ cells

Summary (2/2)

An optimal solution:

- The matrix s stores the decisions; it is read backwards from $j = n$ and $k = c$

Lists of states:

- A state (w, p) is the total weight and the total profit of a subset; dominated states are discarded
- The list L_j is the row j of v in compressed form, computed with `Merge`
- The number of steps is still $\Theta(nc)$, but $|L_j| \leq \min\{2^j, c + 1\}$

Summary of the first part

- A feasible solution of the KP is a binary vector; the 2^n subsets cannot be enumerated when n grows.
- The recurrence $v(j, k)$ has the optimal substructure and overlapping subproblems; the matrix v is filled bottom-up in $\Theta(nc)$ steps, and a single vector with $c + 1$ cells gives the optimal value.
- The matrix s of the decisions gives an optimal solution, read backwards from (n, c).
- The lists of non-dominated states store only the states that can be reached.
- In the second part: a second DP algorithm, which exchanges the roles of profits and weights.