

SAPIENZA
UNIVERSITÀ DI ROMA

Dynamic Programming

Volume 1 — Dynamic programming

The principles of dynamic programming and DP algorithms for knapsack problems, lot sizing, coin change and rod cutting: recurrences, pseudocode, complexity and worked tables.

Fabio Furini

fabiofurini.github.io

Department of Computer, Control and Management Engineering — Sapienza University of Rome

Online version and downloadable material:
fabiofurini.github.io/dynamic-programming/
github.com/fabiofurini/dynamic-programming

Academic year 2026–2027

Contents

Principles **4**

1 Principles of dynamic programming **5**
1.1 Divide, conquer and combine 6
1.2 Problem properties required by DP algorithms 7
1.3 Steps for designing a DP algorithm 8
1.4 The DP algorithms of this module 8
1.5 Typical mistakes 9
1.6 In summary 9

Knapsack problems **10**

2 DP for the 0/1 knapsack problem **11**
2.1 Instances and feasible solutions 12
2.2 A first DP algorithm 14
2.3 A second DP algorithm 36
2.4 Typical mistakes 62
2.5 In summary 63

3 DP for the unbounded knapsack problem **64**
3.1 Instances and feasible solutions 65
3.2 A first DP algorithm 66
3.3 A second DP algorithm 76
3.4 Typical mistakes 87
3.5 In summary 88

4 DP for the bounded knapsack problem **89**
4.1 Instances and feasible solutions 90
4.2 A DP algorithm 91
4.3 Typical mistakes 102
4.4 In summary 102

5 DP for the subset sum problem **104**
5.1 A DP algorithm . . . 105
5.2 Typical mistakes . . . 111
5.3 In summary . . . 111

6 DP for the multiple-choice knapsack problem **112**
6.1 A DP algorithm . . . 113
6.2 Typical mistakes . . . 119
6.3 In summary . . . 119

7 DP for the multidimensional knapsack problem **120**
7.1 A DP algorithm . . . 121
7.2 Typical mistakes . . . 128
7.3 In summary . . . 128

8 DP for the knapsack problem with setups **129**
8.1 A DP algorithm . . . 130
8.2 Typical mistakes . . . 137
8.3 In summary . . . 137

9 DP for the multiple knapsack problem **138**
9.1 A DP algorithm . . . 139
9.2 Typical mistakes . . . 146
9.3 In summary . . . 146

Lot sizing problems **147**

10 DP for the uncapacitated lot sizing problem **148**
10.1 Notation, model and feasible solutions . . . 149
10.2 A DP algorithm for the ULS . . . 154
10.3 Typical mistakes . . . 167
10.4 In summary . . . 167

11 DP for the capacitated lot sizing problem **168**
11.1 Notation, model and feasible solutions . . . 169
11.2 A DP algorithm for the CLS . . . 170
11.3 Typical mistakes . . . 181
11.4 In summary . . . 182

Miscellaneous problems **183**

12 DP for the coin change problem **184**
12.1 Instances and feasible solutions . . . 185

12.2 A DP algorithm 186
12.3 Typical mistakes 196
12.4 In summary 196

13 DP for the rod cutting problem 197
13.1 Instances and feasible solutions 198
13.2 A first DP algorithm 200
13.3 A second DP algorithm 209
13.4 Typical mistakes 221
13.5 In summary 221

Principles

Chapter 1

Principles of dynamic programming

Prerequisites

- From the module *Combinatorial Optimization Problems*, the chapter «Combinatorial optimization: definitions»: the combinatorial optimization problems, their feasible solutions, and the fact that their number grows exponentially.
- From the module *Combinatorial Optimization Problems*, the chapter «Knapsack problems»: the 0/1 knapsack problem (KP), the running example of the DP algorithms.
- From the module *Algorithms*: pseudocode (Chapter 2), recursion (Chapter 14), and the growth of the running time of an algorithm (Chapter 10 «Asymptotic growth»).

Objectives

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

- explain why enumerating all the feasible solutions fails, and what a DP algorithm does instead;
- state the divide, conquer and combine principles, and the difference between independent subproblems and overlapping subproblems;
- check the optimal substructure and the overlapping subproblems on a problem;
- list the steps of a DP algorithm: recurrence, base cases, lookup table, and reconstruction of an optimal solution.

1.1 Divide, conquer and combine

Definition 1.1: DP algorithm

A ***Dynamic Programming (DP) algorithm*** is an exact algorithm to find an optimal solution of a class of optimization problems with specific properties (Section 1.2). It decomposes the original problem into smaller subproblems of the same type, and it reuses the information computed when solving them.

- A brute-force algorithm lists all the feasible solutions, and their number grows exponentially (2^n subsets of items for the KP). A DP algorithm avoids this enumeration by reusing the optimal values of the subproblems.
- A DP algorithm exploits the ***divide, conquer, and combine paradigm***, whose main principles are:
  1. **Dividing principle** $\rightsquigarrow$ decompose the problem into a number of subproblems that are smaller problems of the same type;
  2. **Conquering principle** $\rightsquigarrow$ solve the subproblems in a recursive manner. If the subproblem sizes are small enough just solve the subproblems in a straightforward manner;
  3. **Combining principle** $\rightsquigarrow$ combine optimal solutions of the subproblems to obtain an optimal solution for the original problem.
- DP algorithms divide the problem into ***independent subproblems***, solve them recursively, and then combine their solutions to solve the original problem. Here, independent means that each subproblem generated by the division can be solved without affecting the others.
- In a DP algorithm, the same subproblem may be generated several times during the recursive solution process. In order to avoid redundant computations, the optimal value of each subproblem is stored in a ***lookup table***.
- Whenever the same subproblem is generated again, its optimal value is retrieved from the lookup table instead of being recomputed.

1.2 Problem properties required by DP algorithms

- DP algorithms work only for combinatorial optimization problems that allow a recursive decomposition process respecting the following properties:

  1. ***Optimal Substructure*** $\rightsquigarrow$ the idea is that, when a problem is decomposed into smaller subproblems, an optimal solution of the problem can be built from optimal solutions of the subproblems.

     In practice, to verify this property one usually checks the following points:

     - the problem can be decomposed into smaller subproblems of the same type;
     - subproblems can be solved independently;
     - optimal solutions of the subproblems can be combined to obtain an optimal solution of the original problem;
     - a **recurrence relation** to compute the optimal value of the problem can be established.

  2. ***Overlapping Subproblems*** $\rightsquigarrow$ the idea is that, when a problem is recursively decomposed into smaller subproblems, the same subproblems may be generated several times.

     In practice, to verify this property one usually checks the following points:

     - the recursive decomposition process generates the same subproblem more than once;
     - it is possible to store the optimal values of the subproblems in a lookup table; then whenever the same subproblem is required again, its optimal value can be retrieved from the lookup table instead of being recomputed.

DP algorithms can be used when a problem has both the Optimal Substructure and Overlapping Subproblems.

1.3 Steps for designing a DP algorithm

- Identify the ***recursive decomposition process*** that, starting from a given problem, generates its subproblems, which are smaller problems of the same type, and verify the following two properties:

  1. the Optimal Substructure property, by checking that an optimal solution of the problem can be obtained by combining the optimal solutions of its subproblems;

  2. the Overlapping Subproblems property, by checking whether the recursive decomposition process generates the same subproblems several times;

- The construction of a DP algorithm usually follows two main steps:

  1. computation of the optimal value

     - define a ***recurrence relation*** that expresses the optimal value of a problem in terms of the optimal values of its subproblems;
     - specify the ***base cases***, i.e., the subproblems whose optimal solution is immediate (the optimal value then follows directly from it);
     - determine a DP algorithm that computes the optimal values of all relevant subproblems, typically in a ***bottom-up fashion***, storing each value in a lookup table so that it is computed only once and retrieved whenever needed again;

  2. construction of an optimal solution

     - specify the information that must be additionally stored in order to reconstruct an optimal solution of the problem.
     - identify the way to construct an optimal solution from the information stored.

1.4 The DP algorithms of this module

- Every DP algorithm of this module follows the two steps above: a recurrence with its base cases gives the optimal value, an additional table gives an optimal solution.

Problem	State	Recurrence over	Steps
KP	$v(j,k)$	items and capacities	$\Theta(nc)$
UKP	$v(k)$	capacities	$\Theta(nc)$
BKP	$v(j,k)$	items and capacities	$\Theta(nc\,b_{\max})$
SSP	$r(j,k)$	items and target values	$\Theta(nc)$
MCKP	$v(h,k)$	classes and capacities	$\Theta(nc)$
MDKP	$v(j,k_1,\ldots,k_m)$	items and capacity vectors	$\Theta(m\,n\prod_d(c_d+1))$
KPS	$v(h,k)$, $u(h,i,k)$	classes, items and capacities	$\Theta(nc)$
MKP	$v(j,k_1,\ldots,k_m)$	items and residual capacities	$\Theta(m\,n\prod_i(c_i+1))$
RCP	$v(\ell)$	rod lengths	$\Theta(n^2)$
CCP	$v(k)$	amounts of change	$\Theta(nc)$
ULS	$v(t)$	periods	$\Theta(m^2)$
CLS	$v(t,k)$	periods and inventories	$O(mDC)$

- In all the numerical examples the data are integers: the number of states is finite, and each state is a table cell. The running times $\Theta(nc)$, $\Theta(nc\,b_{\max})$ and $O(mDC)$ contain numbers of the input (a capacity, a bound, a demand), not only its size: they are **pseudo-polynomial**.
- With a vector of capacities (MDKP, MKP) the table has one dimension for each capacity: the number of steps is pseudo-polynomial only for a fixed number m of dimensions or knapsacks.

1.5 Typical mistakes

Warning

- **Using DP without the optimal substructure.** If an optimal solution of the problem cannot be built from optimal solutions of the subproblems, the recurrence is wrong.
- **Using DP without overlapping subproblems.** If every subproblem appears once, storing the values in a lookup table gives nothing: the DP algorithm is a plain recursion.
- **Forgetting the base cases.** The recurrence needs the subproblems whose optimal value is immediate, otherwise the recursion does not stop.
- **Confusing the optimal value with an optimal solution.** The table of the optimal values does not give the solution: an additional table of the decisions is needed.

1.6 In summary

Summary

- A DP algorithm is an exact algorithm that divides a problem into subproblems of the same type, solves them recursively and combines the optimal solutions.
- It works when the problem has the optimal substructure and overlapping subproblems; the optimal values are stored in a lookup table.
- The design has two steps: the optimal value (recurrence, base cases, bottom-up table) and an optimal solution (decisions stored and read backwards).
- Brute force takes exponential time; DP takes a time that is polynomial in the number of states, and pseudo-polynomial when a number of the input is a table dimension.

Knapsack problems

Chapter 2

DP for the 0/1 knapsack problem

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 (Chapter 2), and the size of the input of an algorithm (Chapter 11) (needed to explain why $\Theta(nc)$ is pseudo-polynomial).

Where we were

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

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 $u(j, q)$.

The problem of this chapter is the 0/1 knapsack problem (KP): its definition and its assumptions are on its page.

2.1 Instances and feasible solutions

Example 2.1: 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 Example 2.1.

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.

2.1.1 Encoding of feasible solutions

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:

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

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

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

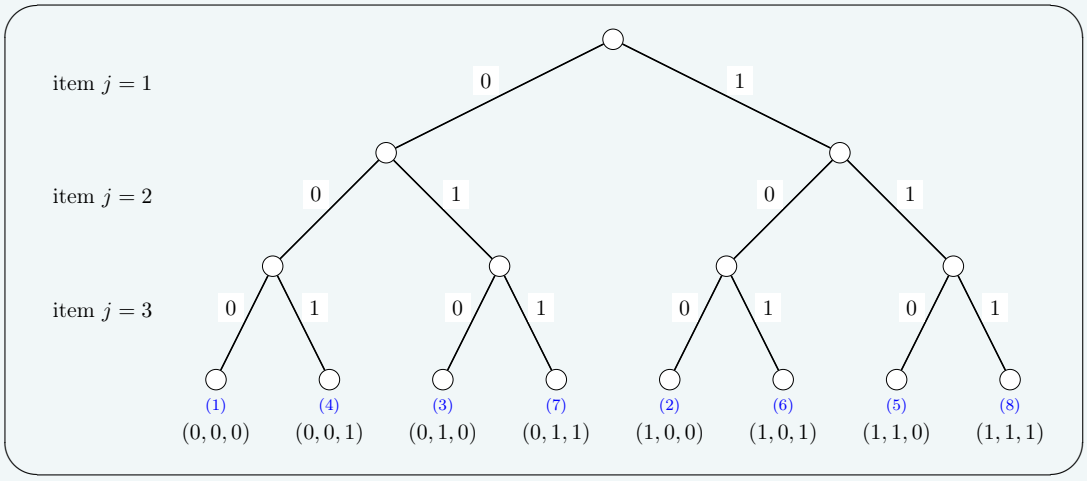
Example 2.2: Enumeration of all possible KP solutions

We consider the KP instance of Example 2.1 with a knapsack of capacity $c = 5$ and the following profits and weights:

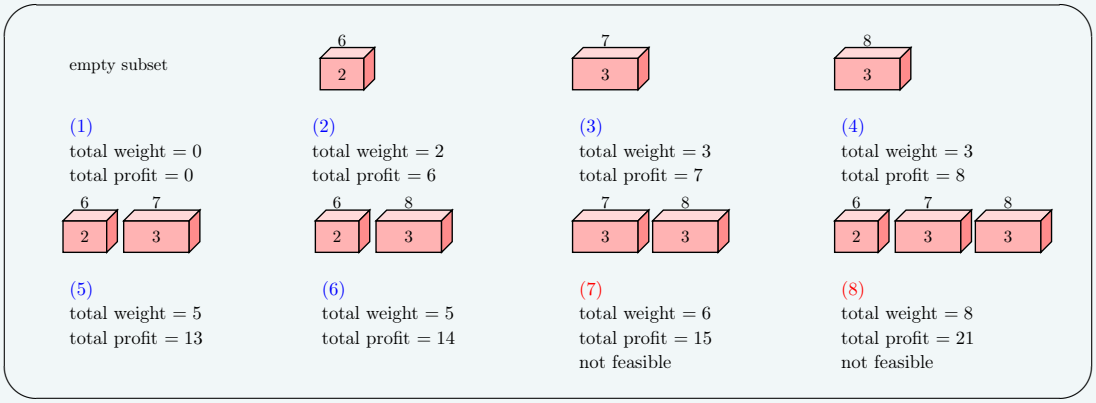
	$j=1$	$j=2$	$j=3$
p_j	6	7	8
w_j	2	3	3

There are $2^3 = 8$ possible subsets of items, corresponding to the different ways of selecting or not selecting the items.

In the following binary tree, 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.



The figure below shows 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.



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

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

$$v(j,k) = \begin{cases} 0 & \text{if } j = 0 \text{ or } k = 0, \\ v(j-1,k) & \text{if } 1 \le j \le n,\ 1 \le k \le c, \text{ and } w_j > k, \\ \max\left\{ v(j-1,k),\ p_j + v(j-1,k-w_j) \right\} & \text{if } 1 \le j \le n,\ 1 \le k \le c,\ w_j \le k. \end{cases} \tag{2.1}$$

- The recursive decomposition process of formula (2.1) 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.

- 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!**

- Different sequences of decisions on the items, 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 formula (2.1).

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

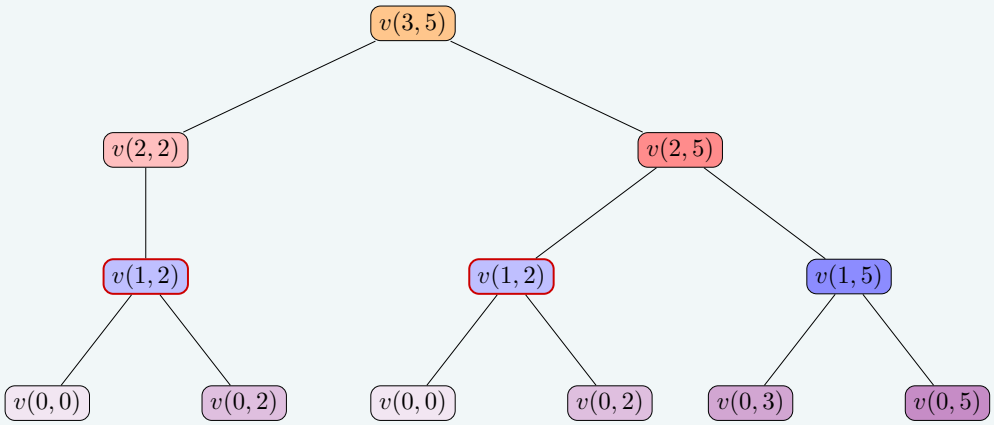
- A **first method** to compute the optimal value $v(n, c)$ of the KP instance is the **top-down approach**.

- The method starts from the call with $j = n$ and $k = c$ and evaluates formula (2.1) 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 2.3: Recursion tree

We consider the KP instance of Example 2.1. 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 $v(0,0)$.

– The value $v(j,k)$ can be computed once the values required by the right-hand side of formula (2.1) 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 2.4: Computing the values of the v function

We consider the KP instance of Example 2.1 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 $v(1,2)$:

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

* 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 $v(3,5)$:

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

– 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, starting from the base cases $v(0,k) = 0$ and $v(j,0) = 0$ 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 KP instance.

– 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 (2.1) 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

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
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]
```

* 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.
* 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 2.5: Execution of the DP algorithm

We consider the KP instance of Example 2.1 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	$v[0][5] \leftarrow 0$
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	$\max\{v[1][5], p[2] + v[1][2]\} = \max\{6, 7+6\} = 13$
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	$\max\{v[2][5], p[3] + v[2][2]\} = \max\{13, 8+6\} = 14$

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

– 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.

- 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 $v(j-1,k-w_j)$.

- 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

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
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]
```

* 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.
* 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 2.6: Execution of the DP algorithm using a single vector

We consider the KP instance of Example 2.1 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	$p[1]+v[0]=6+0>v[2]=0$: $v[2]\leftarrow 6$
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	$p[3]+v[0]=8+0>v[3]=7$: $v[3]\leftarrow 8$

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 reported in Example 2.5. The optimal value of the KP instance is $v[5]=14$.

  - 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
  - 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

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

∗ 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 2.7: Computation of the additional information

We consider the KP instance of Example 2.1 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	$s[0][5] \leftarrow 0$
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	$p[2] + v[1][2] = 7 + 6 > v[1][5] = 6$: $s[2][5] \leftarrow 1$
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	$p[3] + v[2][2] = 8 + 6 > v[2][5] = 13$: $s[3][5] \leftarrow 1$

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

– 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 $j \in \{1, 2, \ldots, n\}$:

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

– 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

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
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
```

Example 2.8: Computing an optimal solution

After the execution of the DP algorithm on the instance of Example 2.1, 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

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 $6 + 8 = 14$.

2.2.1 Number of steps

- 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. 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.

  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 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 $c + 1$.

2.2.2 An implementation with lists

- Step 1: computation of the optimal value
  - 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.
  - This subsection describes a **different implementation of the same DP algorithm**. The recursive decomposition and the optimal substructure are the ones of formula (2.1): 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.

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.

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.

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 recursive formula to compute L_j reads as follows:

$$L_j = \begin{cases} \{(0,0)\} & \text{if } j = 0, \\ \mathrm{nd}\big(L_{j-1} \cup M_j\big) & \text{if } 1 \leq j \leq n, \end{cases} \tag{2.2}$$

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

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

– The recursive decomposition process of formula (2.2) 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.

– 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 $\{1, 2, \ldots, j-1\}$.

$\rightsquigarrow$ **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.

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

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

The optimal value of the KP instance is

$$\max\{\, p \;:\; (w,p) \in L_n \,\}. \tag{2.3}$$

– 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 (2.3) is the profit of the last state of L_n.

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

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

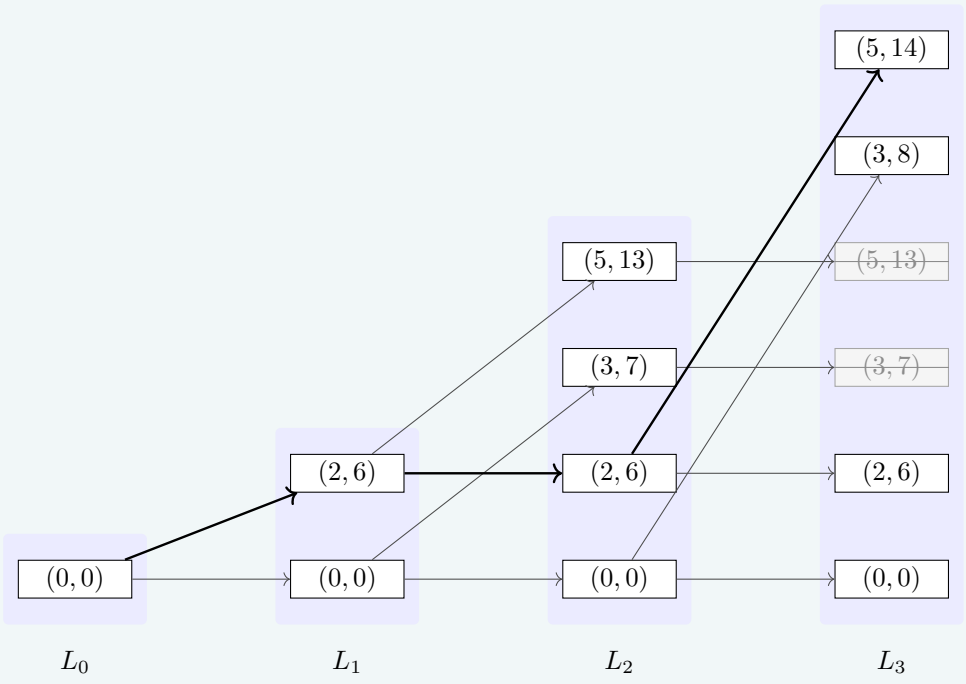
Every state (w,p) of L_j is a capacity value at which the row increases, and p is the new value.

Example 2.9: The lists of states

We consider the KP instance of Example 2.1 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

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.



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 $c = 5$, and therefore no arc leaves these two states towards the states of M_3.

– 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 (2.2) is computed by scanning the two lists at the same time, as in the merge of two sorted lists.

– 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

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)
3      for j := 1 to n do
4          M := new empty list
5          for h := 1 to |L| do
6              if L[h].w + w[j] ≤ c then
7                  append (L[h].w + w[j], L[h].p + p[j]) to M
8              end
9          end
10         L ← Merge(L, M)
11     end
12     return L[|L|].p
```

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

Algorithm

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
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
```

* 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.
* 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 $L[|L|].p$.

Example 2.10: Execution of the DP algorithm

We consider the KP instance of Example 2.1 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	$\{(3,8),(5,14)\}$	$\{(0,0),(2,6),(3,8),(5,14)\}$

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
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.

  - 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.

- Step 2: computation of an optimal solution
  - 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.
- 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 2.11: Computation of the additional information

We consider the KP instance of Example 2.1. 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
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 $6 + p_3 = 6 + 8 = 14$.

- 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

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

Example 2.12: Computing an optimal solution

After the execution of the DP algorithm on the instance of Example 2.1, 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.

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 $6 + 8 = 14$.

2.2.3 Number of steps

- 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
$$S(n, c) = \Theta(nc).$$
- 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 KP instance of Example 2.1 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.

2.3 A second DP algorithm

- Step 1: computation of the optimal value
  - In the DP algorithm of the previous section, a subproblem is identified by a capacity value, and its optimal value is the maximum total profit that can be obtained without exceeding that capacity.
  - In the DP algorithm of this section, the roles of the profits and of the weights are exchanged: a subproblem is identified by a profit value, and its optimal value is the minimum total weight needed to obtain exactly that profit.
  - Since each profit p_j is a positive integer and each item can be selected at most once, the total profit of a feasible solution is an integer between 0 and the total profit of all the items
$$P = \sum_{j=1}^{n} p_j.$$

– Therefore, it is enough to consider the profit values $q \in \{0, 1, \ldots, P\}$.

– The number of subproblems of this DP algorithm grows with the total profit P. To keep the number of subproblems small, the DP algorithm is illustrated on the following KP instance, which has the same item weights and the same knapsack capacity as the KP instance of Example 2.1, and smaller item profits.

Example 2.13: 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	3	3	4
w_j	2	3	3

We refer to this KP instance as Example 2.13.

For any $j \in \{0, 1, \ldots, n\}$ and $q \in \{0, 1, \ldots, P\}$, let $u(j, q)$ be the optimal value of the minimum-weight subinstance given by:

1. a target profit q,
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,

which asks to select a subset of the available items whose total profit is equal to q, so that the total weight of the selected items is minimized.

The recursive formula to compute $u(j, q)$ reads as follows:

$$u(j,q) = \begin{cases} 0 & \text{if } q = 0, \\ +\infty & \text{if } j = 0 \text{ and } 1 \le q \le P, \\ u(j-1,q) & \text{if } 1 \le j \le n,\ 1 \le q \le P, \text{ and } p_j > q, \\ \min\left\{ u(j-1,q),\ w_j + u(j-1,q-p_j) \right\} & \text{if } 1 \le j \le n,\ 1 \le q \le P,\ p_j \le q. \end{cases} \tag{2.4}$$

– The recursive decomposition process of formula (2.4) generates at most 2 **independent subproblems**, corresponding to whether or not to select item j to reach the

target profit q. If $p_j > q$, selecting item j would exceed the target profit, and therefore the only feasible option is not to select it.

- Here, $u(j, 0) = 0$ since the empty subset has total profit 0 and total weight 0. Moreover, $u(0, q) = +\infty$ for $q \geq 1$: in this case no item is available, so no subset has total profit equal to q. The value $+\infty$ is used to rule out this infeasible case.
- If $p_j \leq q$, the optimal value $u(j, q)$ is obtained by trying the two possible options, selecting or not selecting item j, and selecting the one that minimizes the total weight. The corresponding values are $w_j + u(j-1, q-p_j)$ and $u(j-1, q)$, respectively. If $p_j > q$, item j cannot be selected, and therefore $u(j, q) = u(j-1, q)$.
- Hence, a minimum-weight subset with total profit q and available items $\{1, 2, \ldots, j\}$ is built by combining the chosen decision on item j with a minimum-weight subset of the corresponding residual subproblem.

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

- Different sequences of decisions on the items, possibly made for different values of j and q, can lead to the same subproblem, and therefore the same value $u(j, q)$ may need to be computed several times during the recursive evaluation of formula (2.4).

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

- The values $u(n, q)$ do not give the optimal value of the KP instance directly: they give, for each target profit q, the minimum total weight needed to obtain it. A subset with total profit q fits into the knapsack if and only if its minimum total weight does not exceed the capacity, that is, if and only if $u(n, q) \leq c$.

  The optimal value of the KP instance is

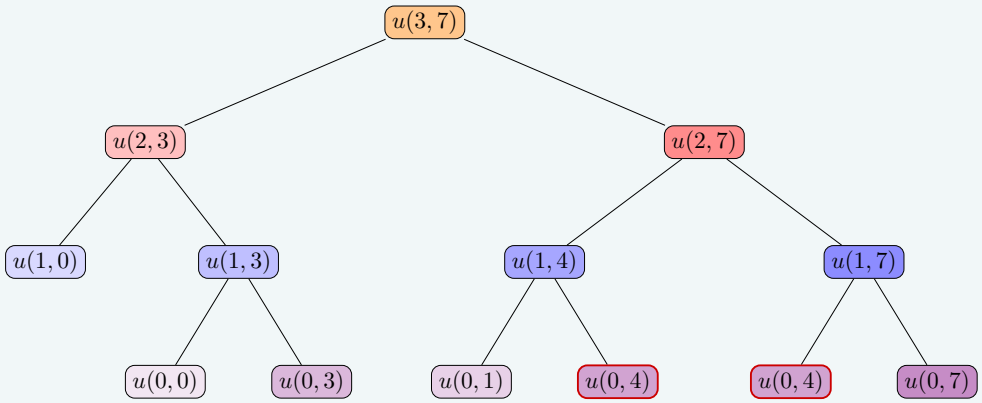
  $$\max \Big\{ q \in \{0, 1, \ldots, P\} \ : \ u(n, q) \leq c \Big\}. \tag{2.5}$$

  * The set in formula (2.5) is never empty, since $u(n, 0) = 0 \leq c$, and therefore the maximum is always well defined.
- A **first method** to compute the optimal value of the KP instance is the **top-down approach**.
- The method evaluates formula (2.4) recursively, starting from the call with $j = n$, for each target profit $q \in \{0, 1, \ldots, P\}$: each call considers item j and either does not select it, or also considers selecting it when its profit does not exceed the target profit. The recursion stops when no items are left or when the target profit is equal to 0. Finally, formula (2.5) is applied to the computed values.

- The **recursion tree** of the function u 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 2.14: Recursion tree

We consider the KP instance of Example 2.13. This is the recursion tree of the top-down approach to compute $u(3, 7)$:



The function is called 13 times, and the same subproblem is solved more than once: $u(0, 4)$ is called 2 times (shown with a red border). Moreover, the top-down approach must be applied to every target profit $q \in \{0, 1, \ldots, 10\}$, and the same subproblem is generated by different initial calls: for instance, the initial call $u(3, 3)$ generates the calls $u(2, 3)$, $u(1, 3)$, $u(1, 0)$, $u(0, 3)$, and $u(0, 0)$, which are already generated by the initial call $u(3, 7)$ shown above.

– The value $u(j, q)$ can be computed once the values required by the right-hand side of formula (2.4) are known. In particular, $u(j - 1, q)$ is needed when item j is not selected and, if $p_j \leq q$, $u(j - 1, q - p_j)$ is needed when item j is selected.

– Therefore, for a fixed value of j, all values $u(j, q)$ can be computed by considering the profit values q in increasing order.

Example 2.15: Computing the values of the u function

We consider the KP instance of Example 2.13 with a knapsack of capacity $c = 5$ and the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$
p_j	3	3	4
w_j	2	3	3

The total profit of all the items is $P = 3+3+4 = 10$. The values of the function u are computed as follows:

* The base cases give $u(0, 0) = 0$ and $u(0, q) = +\infty$ for all $q \geq 1$.

* The first case of formula (2.4) ($q = 0$) gives the base case:

$$u(1, 0) = 0$$

* Once we have computed the base cases, we can compute $u(1,3)$:

$$u(1,3) = \min\{u(0,3),\ w_1 + u(0,0)\} = \min\{+\infty,\ 2+0\} = 2$$

* We can also compute $u(1,4)$ and $u(1,7)$:

$$u(1,4) = \min\{u(0,4),\ w_1 + u(0,1)\} = \min\{+\infty,\ +\infty\} = +\infty$$

$$u(1,7) = \min\{u(0,7),\ w_1 + u(0,4)\} = \min\{+\infty,\ +\infty\} = +\infty$$

* Once we have computed $u(1,3)$ and $u(1,0)$, we can compute $u(2,3)$:

$$u(2,3) = \min\{u(1,3),\ w_2 + u(1,0)\} = \min\{2,\ 3+0\} = 2$$

* Once we have computed $u(1,7)$ and $u(1,4)$, we can compute $u(2,7)$:

$$u(2,7) = \min\{u(1,7),\ w_2 + u(1,4)\} = \min\{+\infty,\ +\infty\} = +\infty$$

* Once we have computed $u(2,7)$ and $u(2,3)$, we can compute $u(3,7)$:

$$u(3,7) = \min\{u(2,7),\ w_3 + u(2,3)\} = \min\{+\infty,\ 3+2\} = 5$$

* Therefore, a total profit of 7 can be obtained with a total weight of 5, which does not exceed the knapsack capacity $c = 5$.

– 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 target profit, starting from the base cases $u(0,0) = 0$, $u(0,q) = +\infty$ for all $q \in \{1,2,\ldots,P\}$, and $u(j,0) = 0$, and building up to the values $u(n,q)$ for all $q \in \{0,1,\ldots,P\}$.

– This observation naturally leads to a **much more efficient method** to compute the optimal value of the KP instance.

– A **second method** to compute the optimal value of the KP instance is the **bottom-up approach**.

– The method starts from the base cases and evaluates formula (2.4) for increasing values of $j \in \{1,2,\ldots,n\}$ and $q \in \{0,1,\ldots,P\}$: each value $u(j,q)$ is computed using already-computed values, until all the values $u(n,q)$ are obtained. The optimal value of the KP instance is then obtained by applying formula (2.5).

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

Algorithm

Algorithm: DP2

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 DP2(n, p, w, c)
2      P := 0
3      for j := 1 to n do
4          P ← P + p[j]
5      end
6      u := new array[0..n][0..P]
7      u[0][0] ← 0
8      for q := 1 to P do
9          u[0][q] ← +∞
10     end
11     for j := 1 to n do
12         for q := 0 to P do
13             if q < p[j] then
14                 u[j][q] ← u[j − 1][q]
15             else
16                 if w[j] + u[j − 1][q − p[j]] < u[j − 1][q] then
17                     u[j][q] ← w[j] + u[j − 1][q − p[j]]
18                 else
19                     u[j][q] ← u[j − 1][q]
20                 end
21             end
22         end
23     end
24     z := 0
25     for q := 0 to P do
26         if u[n][q] ≤ c then
27             z ← q
28         end
29     end
30     return z
```

* The first loop computes the total profit P of all the items, which is the largest target profit that has to be considered.
* The matrix $u[0..n][0..P]$ is the **lookup table**: it stores the optimal values of all subproblems.
* The second loop initializes the values $u[0][0] = 0$ and $u[0][q] = +\infty$ for all target profits $q \in \{1, 2, \ldots, P\}$, 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 $q \in \{0, 1, \ldots, P\}$, computes the optimal value $u[j][q]$ for each target profit q.
* If the profit of item j exceeds the target profit, then $u[j][q] \leftarrow u[j-1][q]$.
* Otherwise, the DP algorithm compares the value obtained by not selecting item j, namely $u[j-1][q]$, with the value obtained by selecting item j, namely $w[j] + u[j-1][q-p[j]]$.
* The auxiliary variable z stores the largest target profit whose minimum total weight does not exceed the knapsack capacity. It is initialized to 0, which is always a feasible target profit since $u[n][0] = 0 \leq c$.
* The last loop scans the target profits in increasing order and updates z whenever $u[n][q] \leq c$, so that at the end z is the maximum of formula (2.5).
* Finally, the optimal value of the KP instance is returned as z.

Example 2.16: Execution of the DP algorithm

We consider the KP instance of Example 2.13 with a knapsack of capacity $c = 5$ and the following profits and weights:

	$j=1$	$j=2$	$j=3$
p_j	3	3	4
w_j	2	3	3

The total profit of all the items is $P = 3 + 3 + 4 = 10$. For each step of the DP algorithm, the table shows the values of j and q, the cell $u[j][q]$, and the operation performed.

j	q	$u[j][q]$	Action
0	0	0	$u[0][0] \leftarrow 0$
0	1	$+\infty$	$u[0][1] \leftarrow +\infty$
0	2	$+\infty$	$u[0][2] \leftarrow +\infty$
0	3	$+\infty$	$u[0][3] \leftarrow +\infty$
0	4	$+\infty$	$u[0][4] \leftarrow +\infty$
0	5	$+\infty$	$u[0][5] \leftarrow +\infty$
0	6	$+\infty$	$u[0][6] \leftarrow +\infty$
0	7	$+\infty$	$u[0][7] \leftarrow +\infty$
0	8	$+\infty$	$u[0][8] \leftarrow +\infty$
0	9	$+\infty$	$u[0][9] \leftarrow +\infty$
0	10	$+\infty$	$u[0][10] \leftarrow +\infty$
1	0	0	$u[1][0] \leftarrow u[0][0] = 0$
1	1	$+\infty$	$u[1][1] \leftarrow u[0][1] = +\infty$
1	2	$+\infty$	$u[1][2] \leftarrow u[0][2] = +\infty$

1	3	2	$\min\{u[0][3], w[1] + u[0][0]\} = \min\{+\infty, 2 + 0\} = 2$
1	4	$+\infty$	$\min\{u[0][4], w[1] + u[0][1]\} = \min\{+\infty, +\infty\} = +\infty$
1	5	$+\infty$	$\min\{u[0][5], w[1] + u[0][2]\} = \min\{+\infty, +\infty\} = +\infty$
1	6	$+\infty$	$\min\{u[0][6], w[1] + u[0][3]\} = \min\{+\infty, +\infty\} = +\infty$
1	7	$+\infty$	$\min\{u[0][7], w[1] + u[0][4]\} = \min\{+\infty, +\infty\} = +\infty$
1	8	$+\infty$	$\min\{u[0][8], w[1] + u[0][5]\} = \min\{+\infty, +\infty\} = +\infty$
1	9	$+\infty$	$\min\{u[0][9], w[1] + u[0][6]\} = \min\{+\infty, +\infty\} = +\infty$
1	10	$+\infty$	$\min\{u[0][10], w[1] + u[0][7]\} = \min\{+\infty, +\infty\} = +\infty$
2	0	0	$u[2][0] \leftarrow u[1][0] = 0$
2	1	$+\infty$	$u[2][1] \leftarrow u[1][1] = +\infty$
2	2	$+\infty$	$u[2][2] \leftarrow u[1][2] = +\infty$
2	3	2	$\min\{u[1][3], w[2] + u[1][0]\} = \min\{2, 3 + 0\} = 2$
2	4	$+\infty$	$\min\{u[1][4], w[2] + u[1][1]\} = \min\{+\infty, +\infty\} = +\infty$
2	5	$+\infty$	$\min\{u[1][5], w[2] + u[1][2]\} = \min\{+\infty, +\infty\} = +\infty$
2	6	5	$\min\{u[1][6], w[2] + u[1][3]\} = \min\{+\infty, 3 + 2\} = 5$
2	7	$+\infty$	$\min\{u[1][7], w[2] + u[1][4]\} = \min\{+\infty, +\infty\} = +\infty$
2	8	$+\infty$	$\min\{u[1][8], w[2] + u[1][5]\} = \min\{+\infty, +\infty\} = +\infty$
2	9	$+\infty$	$\min\{u[1][9], w[2] + u[1][6]\} = \min\{+\infty, +\infty\} = +\infty$
2	10	$+\infty$	$\min\{u[1][10], w[2] + u[1][7]\} = \min\{+\infty, +\infty\} = +\infty$
3	0	0	$u[3][0] \leftarrow u[2][0] = 0$
3	1	$+\infty$	$u[3][1] \leftarrow u[2][1] = +\infty$
3	2	$+\infty$	$u[3][2] \leftarrow u[2][2] = +\infty$
3	3	2	$u[3][3] \leftarrow u[2][3] = 2$
3	4	3	$\min\{u[2][4], w[3] + u[2][0]\} = \min\{+\infty, 3 + 0\} = 3$
3	5	$+\infty$	$\min\{u[2][5], w[3] + u[2][1]\} = \min\{+\infty, +\infty\} = +\infty$
3	6	5	$\min\{u[2][6], w[3] + u[2][2]\} = \min\{5, +\infty\} = 5$
3	7	5	$\min\{u[2][7], w[3] + u[2][3]\} = \min\{+\infty, 3 + 2\} = 5$
3	8	$+\infty$	$\min\{u[2][8], w[3] + u[2][4]\} = \min\{+\infty, +\infty\} = +\infty$
3	9	$+\infty$	$\min\{u[2][9], w[3] + u[2][5]\} = \min\{+\infty, +\infty\} = +\infty$
3	10	8	$\min\{u[2][10], w[3] + u[2][6]\} = \min\{+\infty, 3 + 5\} = 8$

After the execution of the two nested loops of the DP algorithm, this is the matrix u:

$u[j][q]$	0	1	2	3	4	5	6	7	8	9	10
$j=0$	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$j=1$	0	$+\infty$	$+\infty$	2	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$j=2$	0	$+\infty$	$+\infty$	2	$+\infty$	$+\infty$	5	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$j=3$	0	$+\infty$	$+\infty$	2	3	$+\infty$	5	5	$+\infty$	$+\infty$	8

The last loop of the DP algorithm scans the last row of the matrix u and keeps the largest target profit whose minimum total weight does not exceed the knapsack capacity $c = 5$:

q	$u[3][q]$	Action
0	0	$u[3][0] = 0 \leq 5$: $z \leftarrow 0$
1	$+\infty$	$u[3][1] = +\infty > 5$: no update
2	$+\infty$	$u[3][2] = +\infty > 5$: no update
3	2	$u[3][3] = 2 \leq 5$: $z \leftarrow 3$
4	3	$u[3][4] = 3 \leq 5$: $z \leftarrow 4$
5	$+\infty$	$u[3][5] = +\infty > 5$: no update
6	5	$u[3][6] = 5 \leq 5$: $z \leftarrow 6$
7	5	$u[3][7] = 5 \leq 5$: $z \leftarrow 7$
8	$+\infty$	$u[3][8] = +\infty > 5$: no update
9	$+\infty$	$u[3][9] = +\infty > 5$: no update
10	8	$u[3][10] = 8 > 5$: no update

Thus, the optimal value of the KP instance is $z = 7$.

- After the execution of the DP algorithm, the matrix u contains the optimal value $u[j][q] = u(j, q)$ for every item $j \in \{0, 1, \ldots, n\}$ and every target profit $q \in \{0, 1, \ldots, P\}$.

- The DP algorithm computes the row j of the matrix u 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 $u[0..P]$ is sufficient, which uses $P + 1$ cells instead of the $(n + 1)(P + 1)$ cells of the matrix.

- The vector is initialized with the base cases $u[0] = 0$ and $u[q] = +\infty$ for all $q \in \{1, 2, \ldots, P\}$, 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 $u[q]$ is updated for item j, the value $u[q-p_j]$ must still be the one of the row $j-1$, that is $u(j-1, q-p_j)$.

- Since $q-p_j < q$, this is guaranteed if the target profits are considered in **decreasing** order $q = P, P-1, \ldots, p_j$: when the cell q is updated, the cell $q - p_j$ has not been updated yet during the current pass.

– If the target profits were considered in increasing order, the value $u[q - p_j]$ would already be the one of the row j, that is $u(j, q - p_j)$, and item j could be selected more than once, which is not allowed in the KP.

– For every target profit $q < p_j$, the profit of item j exceeds the target profit and therefore $u[q]$ must keep the value $u(j-1, q)$: no update is performed, and the inner loop can stop at $q = p_j$.

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

Algorithm

Algorithm: DP2_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 DP2_Vector(n, p, w, c)
2      P := 0
3      for j := 1 to n do
4          P ← P + p[j]
5      end
6      u := new array[0..P]
7      u[0] ← 0
8      for q := 1 to P do
9          u[q] ← +∞
10     end
11     for j := 1 to n do
12         for q := P downto p[j] do
13             if w[j] + u[q − p[j]] < u[q] then
14                 u[q] ← w[j] + u[q − p[j]]
15             end
16         end
17     end
18     z := 0
19     for q := 0 to P do
20         if u[q] ≤ c then
21             z ← q
22         end
23     end
24     return z
```

∗ The vector $u[0..P]$ is the **lookup table**: at the end of the pass for item j, it stores the row j of the matrix u of the DP algorithm.

* The second loop initializes the values $u[0] = 0$ and $u[q] = +\infty$ for all target profits $q \in \{1, 2, \ldots, P\}$, 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.

* The inner loop, over $q \in \{p[j], p[j] + 1, \ldots, P\}$, considers the target profits 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 $u[q]$, with the value obtained by selecting item j, namely $w[j] + u[q - p[j]]$. The cell is updated only when selecting item j is strictly better.

* The last loop scans the vector and returns the largest target profit whose minimum total weight does not exceed the knapsack capacity, as in formula (2.5).

Example 2.17: Execution of the DP algorithm using a single vector

We consider the KP instance of Example 2.13 with a knapsack of capacity $c = 5$ and the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$
p_j	3	3	4
w_j	2	3	3

The total profit of all the items is $P = 3 + 3 + 4 = 10$ and the second loop initializes $u[0] = 0$ and $u[q] = +\infty$ for all $q \in \{1, 2, \ldots, 10\}$. For each step of the two nested loops, the table shows the values of j and q, the cell $u[q]$ after the step, and the operation performed.

j	q	$u[q]$	Action
1	10	$+\infty$	$w[1] + u[7] = +\infty \geq u[10] = +\infty$: no update
1	9	$+\infty$	$w[1] + u[6] = +\infty \geq u[9] = +\infty$: no update
1	8	$+\infty$	$w[1] + u[5] = +\infty \geq u[8] = +\infty$: no update
1	7	$+\infty$	$w[1] + u[4] = +\infty \geq u[7] = +\infty$: no update
1	6	$+\infty$	$w[1] + u[3] = +\infty \geq u[6] = +\infty$: no update
1	5	$+\infty$	$w[1] + u[2] = +\infty \geq u[5] = +\infty$: no update
1	4	$+\infty$	$w[1] + u[1] = +\infty \geq u[4] = +\infty$: no update
1	3	2	$w[1] + u[0] = 2 < u[3] = +\infty$: $u[3] \leftarrow 2$
2	10	$+\infty$	$w[2] + u[7] = +\infty \geq u[10] = +\infty$: no update
2	9	$+\infty$	$w[2] + u[6] = +\infty \geq u[9] = +\infty$: no update
2	8	$+\infty$	$w[2] + u[5] = +\infty \geq u[8] = +\infty$: no update
2	7	$+\infty$	$w[2] + u[4] = +\infty \geq u[7] = +\infty$: no update
2	6	5	$w[2] + u[3] = 5 < u[6] = +\infty$: $u[6] \leftarrow 5$
2	5	$+\infty$	$w[2] + u[2] = +\infty \geq u[5] = +\infty$: no update

2	4	$+\infty$	$w[2]+u[1]=+\infty \geq u[4]=+\infty$: no update
2	3	2	$w[2]+u[0]=3 \geq u[3]=2$: no update
3	10	8	$w[3]+u[6]=8 < u[10]=+\infty$: $u[10] \leftarrow 8$
3	9	$+\infty$	$w[3]+u[5]=+\infty \geq u[9]=+\infty$: no update
3	8	$+\infty$	$w[3]+u[4]=+\infty \geq u[8]=+\infty$: no update
3	7	5	$w[3]+u[3]=5 < u[7]=+\infty$: $u[7] \leftarrow 5$
3	6	5	$w[3]+u[2]=+\infty \geq u[6]=5$: no update
3	5	$+\infty$	$w[3]+u[1]=+\infty \geq u[5]=+\infty$: no update
3	4	3	$w[3]+u[0]=3 < u[4]=+\infty$: $u[4] \leftarrow 3$

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

$u[q]$	0	1	2	3	4	5	6	7	8	9	10
initialization	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
pass $j=1$	0	$+\infty$	$+\infty$	2	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
pass $j=2$	0	$+\infty$	$+\infty$	2	$+\infty$	$+\infty$	5	$+\infty$	$+\infty$	$+\infty$	$+\infty$
pass $j=3$	0	$+\infty$	$+\infty$	2	3	$+\infty$	5	5	$+\infty$	$+\infty$	8

Each pass transforms the vector u from the row $j-1$ into the row j of the matrix u reported in Example 2.16. The last loop then scans the vector and gives the optimal value $z=7$ of the KP instance, as in Example 2.16.

- 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 target profit q, and therefore cannot be stored in a single vector.

- Step 2: computation of an optimal solution
  - 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 target profit q, in addition to the optimal value $u[j][q]$, we store $s[j][q]$, which tells whether item j is selected in a minimum-weight subset of the items $\{1, 2, \ldots, j\}$ with total profit equal to q.
  - More precisely, $s[j][q]$ is equal to 1 if item j is selected when computing the value $u[j][q]$, 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

Algorithm: DP2_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 $s[0..n][0..P]$

```
1  Function DP2_InfoSol(n, p, w, c)
2      P := 0
3      for j := 1 to n do
4          P ← P + p[j]
5      end
6      u := new array[0..n][0..P]
7      s := new array[0..n][0..P]
8      u[0][0] ← 0
9      s[0][0] ← 0
10     for q := 1 to P do
11         u[0][q] ← +∞
12         s[0][q] ← 0
13     end
14     for j := 1 to n do
15         for q := 0 to P do
16             if q < p[j] then
17                 u[j][q] ← u[j − 1][q]
18                 s[j][q] ← 0
19             else
20                 if w[j] + u[j − 1][q − p[j]] < u[j − 1][q] then
21                     u[j][q] ← w[j] + u[j − 1][q − p[j]]
22                     s[j][q] ← 1
23                 else
24                     u[j][q] ← u[j − 1][q]
25                     s[j][q] ← 0
26                 end
27             end
28         end
29     end
30     z := 0
31     for q := 0 to P do
32         if u[n][q] ≤ c then
33             z ← q
34         end
35     end
36     return z, s
```

∗ The DP algorithm uses the additional matrix s, which keeps track of the decision

made when $u[j][q]$ is computed. If item j is selected, $s[j][q] \leftarrow 1$ stores this decision; otherwise, $s[j][q] \leftarrow 0$ stores that item j is not selected.

Example 2.18: Computation of the additional information

We consider the KP instance of Example 2.13 with a knapsack of capacity $c = 5$ and the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$
p_j	3	3	4
w_j	2	3	3

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

j	q	$s[j][q]$	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	$s[0][7] \leftarrow 0$
0	8	0	$s[0][8] \leftarrow 0$
0	9	0	$s[0][9] \leftarrow 0$
0	10	0	$s[0][10] \leftarrow 0$
1	0	0	$0 < 3$: $s[1][0] \leftarrow 0$
1	1	0	$1 < 3$: $s[1][1] \leftarrow 0$
1	2	0	$2 < 3$: $s[1][2] \leftarrow 0$
1	3	1	$w[1] + u[0][0] = 2 + 0 < u[0][3] = +\infty$: $s[1][3] \leftarrow 1$
1	4	0	$w[1] + u[0][1] = +\infty \geq u[0][4] = +\infty$: $s[1][4] \leftarrow 0$
1	5	0	$w[1] + u[0][2] = +\infty \geq u[0][5] = +\infty$: $s[1][5] \leftarrow 0$
1	6	0	$w[1] + u[0][3] = +\infty \geq u[0][6] = +\infty$: $s[1][6] \leftarrow 0$
1	7	0	$w[1] + u[0][4] = +\infty \geq u[0][7] = +\infty$: $s[1][7] \leftarrow 0$
1	8	0	$w[1] + u[0][5] = +\infty \geq u[0][8] = +\infty$: $s[1][8] \leftarrow 0$
1	9	0	$w[1] + u[0][6] = +\infty \geq u[0][9] = +\infty$: $s[1][9] \leftarrow 0$
1	10	0	$w[1] + u[0][7] = +\infty \geq u[0][10] = +\infty$: $s[1][10] \leftarrow 0$
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	0	$w[2] + u[1][0] = 3 + 0 \geq u[1][3] = 2$: $s[2][3] \leftarrow 0$
2	4	0	$w[2] + u[1][1] = +\infty \geq u[1][4] = +\infty$: $s[2][4] \leftarrow 0$
2	5	0	$w[2] + u[1][2] = +\infty \geq u[1][5] = +\infty$: $s[2][5] \leftarrow 0$
2	6	1	$w[2] + u[1][3] = 3 + 2 < u[1][6] = +\infty$: $s[2][6] \leftarrow 1$
2	7	0	$w[2] + u[1][4] = +\infty \geq u[1][7] = +\infty$: $s[2][7] \leftarrow 0$
2	8	0	$w[2] + u[1][5] = +\infty \geq u[1][8] = +\infty$: $s[2][8] \leftarrow 0$
2	9	0	$w[2] + u[1][6] = +\infty \geq u[1][9] = +\infty$: $s[2][9] \leftarrow 0$

2	10	0	$w[2]+u[1][7]=+\infty \geq u[1][10]=+\infty$: $s[2][10] \leftarrow 0$
3	0	0	$0<4$: $s[3][0] \leftarrow 0$
3	1	0	$1<4$: $s[3][1] \leftarrow 0$
3	2	0	$2<4$: $s[3][2] \leftarrow 0$
3	3	0	$3<4$: $s[3][3] \leftarrow 0$
3	4	1	$w[3]+u[2][0]=3+0<u[2][4]=+\infty$: $s[3][4] \leftarrow 1$
3	5	0	$w[3]+u[2][1]=+\infty \geq u[2][5]=+\infty$: $s[3][5] \leftarrow 0$
3	6	0	$w[3]+u[2][2]=+\infty \geq u[2][6]=5$: $s[3][6] \leftarrow 0$
3	7	1	$w[3]+u[2][3]=3+2<u[2][7]=+\infty$: $s[3][7] \leftarrow 1$
3	8	0	$w[3]+u[2][4]=+\infty \geq u[2][8]=+\infty$: $s[3][8] \leftarrow 0$
3	9	0	$w[3]+u[2][5]=+\infty \geq u[2][9]=+\infty$: $s[3][9] \leftarrow 0$
3	10	1	$w[3]+u[2][6]=3+5<u[2][10]=+\infty$: $s[3][10] \leftarrow 1$

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

$s[j][q]$	0	1	2	3	4	5	6	7	8	9	10
$j=0$	0	0	0	0	0	0	0	0	0	0	0
$j=1$	0	0	0	1	0	0	0	0	0	0	0
$j=2$	0	0	0	0	0	0	1	0	0	0	0
$j=3$	0	0	0	0	1	0	0	1	0	0	1

– 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 the target profit $q=z$ returned by the DP algorithm and $x_j=0$ for all $j \in \{1,2,\ldots,n\}$:

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

– 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

Algorithm: DP2_Reconstruct

Input: The number n of items, the optimal value z of the KP instance, the array $p[1..n]$ of item profits, and the matrix $s[0..n][0..P]$

Output: An optimal solution of the KP instance

```
1  Function DP2_Reconstruct(n, z, p, s)
2      x := new array[1..n]
3      for j := 1 to n do
4          x[j] ← 0
5      end
6      q := z
7      for j := n downto 1 do
8          if s[j][q] = 1 then
9              x[j] ← 1
10             q ← q − p[j]
11         end
12     end
13     return x
```

Example 2.19: Computing an optimal solution

After the execution of the DP algorithm on the instance of Example 2.13, the optimal value is $z = 7$ and we have the following matrix s:

$s[j][q]$	0	1	2	3	4	5	6	7	8	9	10
$j = 0$	0	0	0	0	0	0	0	0	0	0	0
$j = 1$	0	0	0	1	0	0	0	0	0	0	0
$j = 2$	0	0	0	0	0	0	1	0	0	0	0
$j = 3$	0	0	0	0	1	0	0	1	0	0	1

We can reconstruct an optimal solution by starting from $j = 3$ and $q = z = 7$ as follows:

Iteration	j	q	$s[j][q]$	$x[1]$	$x[2]$	$x[3]$	Action
–	–	–	–	0	0	0	$q = 7$
1	3	7	1	0	0	1	$q \leftarrow 7 - 4 = 3$
2	2	3	0	0	0	1	no update
3	1	3	1	1	0	1	$q \leftarrow 3 - 3 = 0$

Thus, the vector

$$(1, 0, 1)$$

is an optimal solution. The corresponding optimal value is $3 + 4 = 7$.

2.3.1 Number of steps

- Let $S(n, P)$ denote the number of steps executed by `DP2` in the worst case, where n is the number of items and $P = \sum_{j=1}^{n} p_j$ is the total profit of all the items. We compute $S(n, P)$ by counting how many times the lines of the pseudo-code of `DP2` are executed.

- The condition at line 16 is checked only when $q \geq p[j]$, that is for $P+1-p_j$ target profits for each item j, and therefore $n(P+1) - P$ times in total. The number of executions of the relevant lines is:

  - Line 2 is executed 1 time.
  - Line 3 is executed n times.
  - Line 4 is executed n times.
  - Line 6 is executed 1 time.
  - Line 7 is executed 1 time.
  - Line 8 is executed P times.
  - Line 9 is executed P times.
  - Line 11 is executed n times.
  - Line 12 is executed $n(P+1)$ times.
  - Line 13 is executed $n(P+1)$ times.
  - Line 16 is executed $n(P+1) - P$ times.
  - one among Line 14, Line 17, and Line 19 is executed for each pair (j, q), and therefore these assignment lines are executed $n(P+1)$ times in total.
  - Line 24 is executed 1 time.
  - Line 25 is executed $P+1$ times.
  - Line 26 is executed $P+1$ times.
  - Line 27 is executed at most $P+1$ times.
  - Line 30 is executed 1 time.

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

$$\begin{aligned} &1 + n + n + 1 + 1 + P + P + n + n(P+1) + n(P+1) \\ &+ \big(n(P+1) - P\big) + n(P+1) + 1 + (P+1) + (P+1) + (P+1) + 1. \end{aligned}$$

  Thus,

$$S(n, P) = 4nP + 7n + 4P + 8.$$

  Hence,

$$S(n, P) = \Theta(nP).$$

  Indeed,

$$\lim_{n,P\to\infty} \frac{4nP + 7n + 4P + 8}{nP} = \lim_{n,P\to\infty} \left(4 + \frac{7}{P} + \frac{4}{n} + \frac{8}{nP}\right) = 4 \in \mathbb{R}_{>0}.$$

The DP algorithm executes

$$S(n, P) = \Theta(nP)$$

steps in the worst case, where $P = \sum_{j=1}^{n} p_j$.

- The version `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 $\Theta(nP)$.
- The version `DP2_Vector` uses a single vector instead of the matrix and executes the same two nested loops, whose inner loop performs at most $P + 1$ iterations for each item. Therefore, its number of steps remains $\Theta(nP)$, while the number of cells of the lookup table decreases from $(n + 1)(P + 1)$ to $P + 1$.

2.3.2 An implementation with lists

- Step 1: computation of the optimal value
  - The DP algorithm of this section stores one value for every pair given by an item and a target profit, that is $(n + 1)(P + 1)$ values in total. Many of these values are equal to $+\infty$, because no subset of the available items has that total profit.
  - This subsection describes a **different implementation of the same DP algorithm**. The recursive decomposition and the optimal substructure are the ones of formula (2.4): what changes is the data structure.
  - Instead of a whole row of the matrix, only the pairs that are actually obtained by a subset of items are stored, and among them only those that can lead to an optimal solution. These pairs are kept in a list, one list for each item.

A **state** is a pair (w, p), where w is the total weight and p is the total profit of a subset of items.

  - 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, and in this case it is enough to keep one of them. More generally, a state can be discarded whenever another state is at least as good in both components.

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.

For any $j \in \{0, 1, \ldots, n\}$, let L_j be the list of the non-dominated states that can be obtained by selecting a subset of the items $\{1, 2, \ldots, j\}$, sorted by strictly increasing weight.

The recursive formula to compute L_j reads as follows:

$$L_j = \begin{cases} \{(0,0)\} & \text{if } j = 0, \\ \mathrm{nd}\big(L_{j-1} \cup M_j\big) & \text{if } 1 \leq j \leq n, \end{cases} \tag{2.6}$$

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

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

– The recursive decomposition process of formula (2.6) generates 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.

– Here, $L_0 = \{(0, 0)\}$ since the only subset of an empty set of items is the empty subset.

– As in formula (2.4), the knapsack capacity does not appear in the recursion: the list L_j collects the states of *all* the subsets of the items $\{1, 2, \ldots, j\}$, including those whose total weight exceeds the capacity.

– 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 $\{1, 2, \ldots, j-1\}$.

$\rightsquigarrow$ **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.

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

– The knapsack capacity is used only at the end, exactly as in formula (2.5): among the states of L_n, only those whose total weight does not exceed the capacity correspond to a feasible solution.

The optimal value of the KP instance is

$$\max\{\, p \;:\; (w,p) \in L_n,\ w \le c \,\}. \tag{2.7}$$

– The set in formula (2.7) is never empty, since $(0,0) \in L_n$ and $0 \le c$, and therefore the maximum is always well defined.

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

$$\min\{\, w \;:\; (w,p) \in L_j,\ p \ge q \,\} = \min_{q' \ge q} u[j][q'] \qquad \forall q \in \{0,1,\ldots,P\}.$$

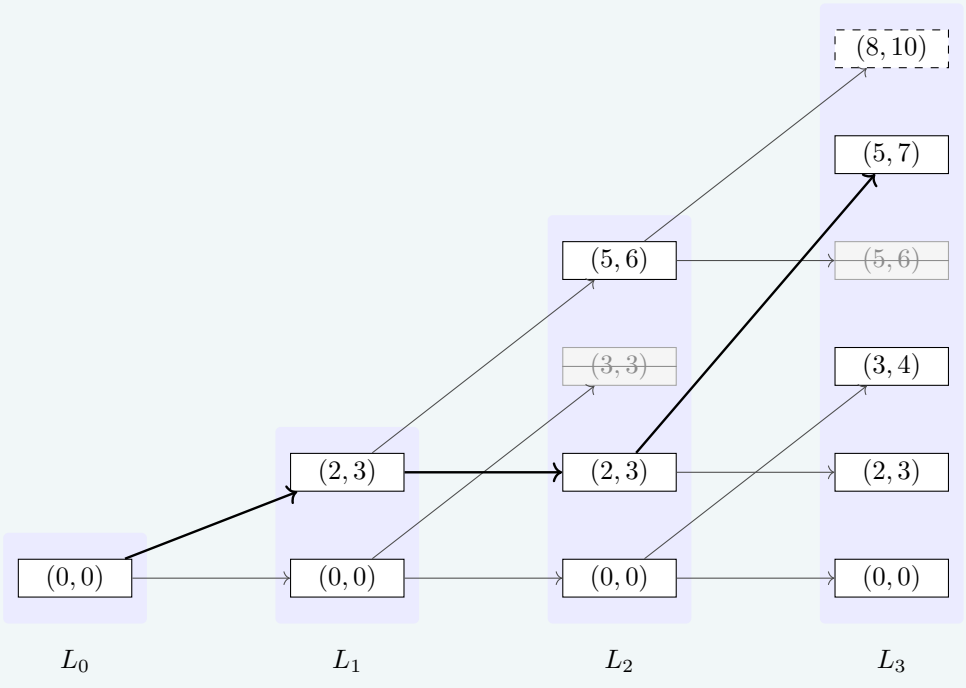
The target profits whose row entry is $+\infty$ leave no state in the list, and a target profit whose minimum weight is not smaller than the one of a larger target profit leaves no state either.

Example 2.20: The lists of states

We consider the KP instance of Example 2.13 with a knapsack of capacity $c = 5$ and the following profits and weights:

	$j=1$	$j=2$	$j=3$
p_j	3	3	4
w_j	2	3	3

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.



The two crossed states are dominated and are therefore discarded: $(3, 3)$ is dominated by $(2, 3)$ and $(5, 6)$ is dominated by $(5, 7)$. The state $(8, 10)$, drawn with a dashed border, has a total weight larger than the knapsack capacity $c = 5$: it is kept in the list, because the capacity is not used during the recursion, and it is discarded only by formula (2.7).

– 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 (2.6) is computed by scanning the two lists at the same time, as in the merge of two sorted lists.

– 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

Algorithm: DP2_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 DP2_Lists(n, p, w, c)
2      L := new list containing the single state (0, 0)
3      for j := 1 to n do
4          M := new empty list
5          for h := 1 to |L| do
6              append (L[h].w + w[j], L[h].p + p[j]) to M
7          end
8          L ← Merge(L, M)
9      end
10     z := 0
11     for h := 1 to |L| do
12         if L[h].w ≤ c then
13             z ← L[h].p
14         end
15     end
16     return z
```

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

Algorithm

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
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
```

* 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. No state is discarded here, because the knapsack capacity is not used during the recursion.
* 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.
* The last loop implements formula (2.7). The states of L have strictly increasing weights and strictly increasing profits, and therefore the auxiliary variable z ends with the profit of the last state whose weight does not exceed the knapsack capacity.
* Finally, the optimal value of the KP instance is returned as z.

Example 2.21: Execution of the DP algorithm

We consider the KP instance of Example 2.13 with a knapsack of capacity $c = 5$ and the following profits and weights:

	$j=1$	$j=2$	$j=3$
p_j	3	3	4
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	$\{(2,3)\}$	$\{(0,0),(2,3)\}$
2	$\{(3,3),(5,6)\}$	$\{(0,0),(2,3),(5,6)\}$
3	$\{(3,4),(5,7),(8,10)\}$	$\{(0,0),(2,3),(3,4),(5,7),(8,10)\}$

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	$(2,3)$	M_1	$3 > \pi = 0$: append
2	$(0,0)$	L_1	$0 > \pi = -1$: append
2	$(2,3)$	L_1	$3 > \pi = 0$: append
2	$(3,3)$	M_2	$3 \leq \pi = 3$: discard (dominated)
2	$(5,6)$	M_2	$6 > \pi = 3$: append
3	$(0,0)$	L_2	$0 > \pi = -1$: append
3	$(2,3)$	L_2	$3 > \pi = 0$: append
3	$(3,4)$	M_3	$4 > \pi = 3$: append
3	$(5,7)$	M_3	$7 > \pi = 4$: append
3	$(5,6)$	L_2	$6 \leq \pi = 7$: discard (dominated)
3	$(8,10)$	M_3	$10 > \pi = 7$: append

The last loop scans L_3 and keeps the profit of the last state whose weight does not exceed $c = 5$: the states $(0,0)$, $(2,3)$, $(3,4)$ and $(5,7)$ give $z \leftarrow 0$, $z \leftarrow 3$, $z \leftarrow 4$ and $z \leftarrow 7$, while the state $(8,10)$ causes no update since $8 > 5$. Therefore the optimal value of the KP instance is 7.

  - After the execution of the DP algorithm, the list L contains the non-dominated states L_n, sorted by strictly increasing weight and strictly increasing profit.

- Step 2: computation of an optimal solution
  - 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.

– 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 2.22: Computation of the additional information

We consider the KP instance of Example 2.13. 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	$(2,3)$	1	1
2	1	$(0,0)$	0	1
2	2	$(2,3)$	0	2
2	3	$(5,6)$	1	2
3	1	$(0,0)$	0	1
3	2	$(2,3)$	0	2
3	3	$(3,4)$	1	1
3	4	$(5,7)$	1	2
3	5	$(8,10)$	1	3

For instance, the state $L_3[4] = (5,7)$ has $\sigma_3[4] = 1$ and $\rho_3[4] = 2$: it is obtained by selecting item 3 from the state $L_2[2] = (2,3)$, since $2 + w_3 = 2 + 3 = 5$ and $3 + p_3 = 3 + 4 = 7$.

– The reconstruction starts from the position $\tilde{h}$ of the state that attains the maximum of formula (2.7), that is the position at which the last loop of `DP2_Lists` updates z for the last time. It is not in general the last position of L_n, because the last states of L_n may have a total weight larger than the knapsack capacity.

– An optimal solution is then reconstructed by scanning the items from $j = n$ down to $j = 1$, starting from $h = \tilde{h}$:

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

– 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

Algorithm: DP2_Lists_Reconstruct

Input: The number n of items, the position $\tilde{h}$ of the optimal state in L_n, and the stored values σ and ρ

Output: An optimal solution of the KP instance

```
1  Function DP2_Lists_Reconstruct(n, h̃, σ, ρ)
2      x := new array[1..n]
3      for j := 1 to n do
4          x[j] ← 0
5      end
6      h := h̃
7      for j := n downto 1 do
8          x[j] ← σ_j[h]
9          h ← ρ_j[h]
10     end
11     return x
```

Example 2.23: Computing an optimal solution

After the execution of the DP algorithm on the instance of Example 2.13, the last list is

$$L_3 = \{(0,0), (2,3), (3,4), (5,7), (8,10)\}$$

and the maximum of formula (2.7) is attained by the state $L_3[4] = (5,7)$, so that $\tilde{h} = 4$ and the optimal value is 7.

We can reconstruct an optimal solution by starting from $j = 3$ and $h = \tilde{h} = 4$ 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	$(5,7)$	1	2	0	0	1	$h \leftarrow 2$
2	2	2	$(2,3)$	0	2	0	0	1	$h \leftarrow 2$
3	1	2	$(2,3)$	1	1	1	0	1	$h \leftarrow 1$

Thus, the vector

$$(1,0,1)$$

is an optimal solution. The corresponding optimal value is $3 + 4 = 7$.

2.3.3 Number of steps

- Let $S(n, P)$ denote the number of steps executed by `DP2_Lists` in the worst case, where n is the number of items and $P = \sum_{j=1}^{n} p_j$ is the total profit of all the items.
- The states of a list have strictly increasing profits, and every profit belongs to $\{0, 1, \ldots, P\}$: therefore $|L_j| \leq P + 1$ for all $j \in \{0, 1, \ldots, n\}$, and $|M_j| = |L_{j-1}|$.
- For each item, `DP2_Lists` scans L_{j-1} once to build M_j and then calls `Merge`, which scans $|L_{j-1}| + |M_j| \leq 2(P + 1)$ states and performs a constant number of operations for each of them. The final loop scans L_n once. Therefore

$$S(n, P) = \Theta(nP).$$

- The matrix u always has $(n+1)(P+1)$ cells, most of which are equal to $+\infty$, while the lists contain only the states that are actually reachable. On the KP instance of Example 2.13 the lists contain 1, 2, 3, and 5 states, against the 44 cells of the matrix u.
- The list L_j also satisfies $|L_j| \leq 2^j$, since the items $\{1, 2, \ldots, j\}$ have 2^j subsets. Therefore

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

2.4 Typical mistakes

Warning

- **Updating the single vector in increasing order of capacity.** The cell $v[k - w_j]$ would already contain row j, and item j could be selected more than once; the capacities are scanned from c downwards.
- **Forgetting the base cases.** With no item available the optimal value is 0 for every capacity: $v(0, k) = 0$ for all $k \in \{0, 1, \ldots, c\}$, not only for $k = 0$.
- **Reconstructing with the wrong index.** At every step the item index decreases by one, while the capacity decreases by w_j only if item j is selected.
- **Calling $\Theta(nc)$ polynomial.** The capacity c is written with $\log c$ digits, so the running time is exponential in the size of the input.
- **Reading $u(n, q)$ as the optimal value.** In the second DP algorithm $u(n, q)$ is a minimum weight; the optimal value is the largest q with $u(n, q) \leq c$.

2.5 In summary

Summary

- The recurrence $v(j,k)$ chooses between leaving item j out and taking it; it has the optimal substructure and overlapping subproblems.
- The matrix is filled bottom-up in $\Theta(nc)$ steps; a single vector scanned downwards gives the optimal value with only $c+1$ cells.
- An optimal solution comes from the matrix s of the decisions, read backwards from (n,c).
- The running time $\Theta(nc)$ is pseudo-polynomial; the lists of non-dominated states store only the states that can be reached.
- Exchanging profits and weights gives $\Theta(nP)$ steps, which is better when the total profit P is small.

Dashboard of the KP example. Example 2.1 ($c = 5$, three items) with the bounds found so far:

Method	Lower bound	Upper bound	Origin
DP algorithm	14	14	optimal value, $\Theta(nc)$ steps

Chapter 3

DP for the unbounded knapsack problem

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 (Chapter 2) and the size of the input (Chapter 11).

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.

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

3.1 Instances and feasible solutions

Example 3.1: 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 Example 3.1.

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.

3.1.1 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 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.

3.2 A first 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 recursive formula to compute $v(k)$ reads as follows:

$$v(k) = \begin{cases} 0, & \text{if } k = 0, \\ \max\limits_{j\,:\,w_j \le k} \{p_j + v(k - w_j)\}, & \text{if } k \ge 1 \text{ and } w_j \le k \text{ for some } j, \\ 0, & \text{if } k \ge 1 \text{ and } w_j > k \text{ for every } j. \end{cases} \tag{3.1}$$

- The recursive decomposition process of formula (3.1) generates at most n **independent subproblems**, each corresponding to a possible choice of an item type j, among those with $w_j \le k$, 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 $v(k) = 0$.
- The optimal value $v(k)$ is obtained by trying every item type j with $w_j \le k$ 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!**

- Different choices of the first item type, possibly made for different values of k, can lead to the same residual capacity, and therefore the same value $v(k)$ may need to be computed several times during the recursive evaluation of formula (3.1).

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

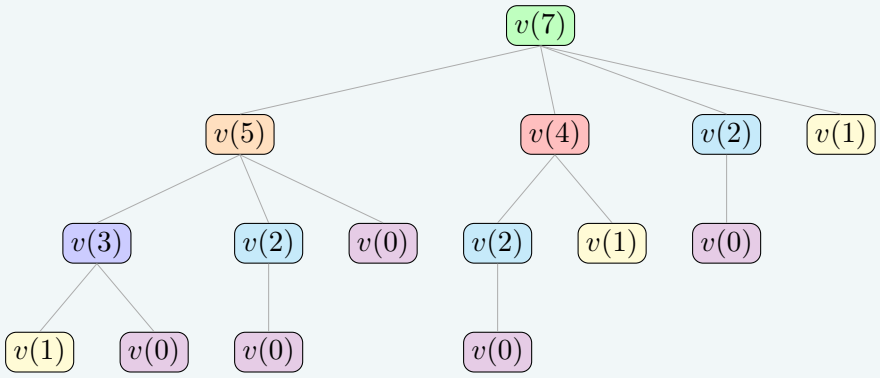
– A **first method** to compute the optimal value $v(c)$ of the UKP instance is the **top-down approach**.

– The method starts from the call with $k = c$ and evaluates formula (3.1) 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 3.2: Recursion tree

We consider the UKP instance of Example 3.1, 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 $v(7)$:



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, and $v(0)$ is called 5 times. All the other calls are made only once.

– 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 3.3: Computing the values of the v function

We consider the UKP instance of Example 3.1 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:

$$v(1) = 0.$$

* 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:

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

* 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:

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

* 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:

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

– 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.

– 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 formula (3.1) 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.

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

Algorithm

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
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]
```

* The vector $v[0..c]$ is the **lookup table**: it stores the optimal values of all sub-problems, 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 3.4: Execution of the DP algorithm

We consider the UKP instance of Example 3.1 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.

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	$v[1] \leftarrow m = 0$
	–	–	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	$v[3] \leftarrow m = 5$
	–	–	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	$v[5] \leftarrow m = 10$

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	$v[7] \leftarrow m = 14$

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.

    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

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
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
```

* 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 3.5: Computation of the additional information

We consider the UKP instance of Example 3.1 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.

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	$v[1] \leftarrow m = 0$, $s[1] \leftarrow t = 0$
	–	–	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	$v[3] \leftarrow m = 5$, $s[3] \leftarrow t = 2$
	–	–	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	$v[5] \leftarrow m = 10$, $s[5] \leftarrow t = 3$

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	$v[7] \leftarrow m = 14$, $s[7] \leftarrow t = 1$

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

– 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

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
```

Example 3.6: Computing an optimal solution

After the execution of the DP algorithm on the instance of Example 3.1, 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	$k = 5 - 5 = 0$

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 $v(7) = 14$.

3.2.1 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. 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.

  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 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 $\Theta(nc)$.

3.3 A second 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 recursive formula to compute $v(j, k)$ reads as follows:

$$v(j,k) = \begin{cases} 0, & \text{if } j = 0, \\ v(j-1,k), & \text{if } j \geq 1 \text{ and } k < w_j, \\ \max\{v(j-1,k),\ p_j + v(j,k-w_j)\}, & \text{if } j \geq 1 \text{ and } k \geq w_j. \end{cases} \tag{3.2}$$

- The recursive decomposition process of formula (3.2) 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.
- 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.

– 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.

  $\rightsquigarrow$ **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 formula (3.2).

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

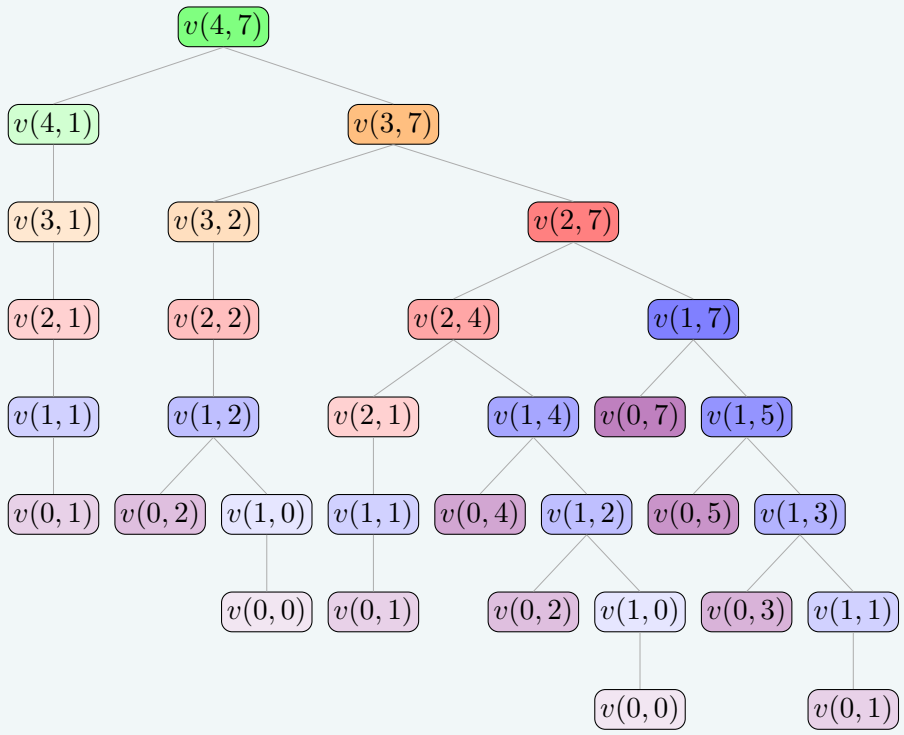
– A **first method** to compute the optimal value $v(n, c)$ of the UKP instance is the **top-down approach**.

– The method starts from the call with $(j, k) = (n, c)$ and evaluates formula (3.2) 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 3.7: Recursion tree

We consider the UKP instance of Example 3.1, 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 $v(4, 7)$:



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, and $v(2,1)$ is called 2 times. All the other calls are made only once.

– 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 3.8: Computing the values of the v function

We consider the UKP instance of Example 3.1 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:

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

$$v(1,0) = 0, \qquad v(1,1) = v(0,1) = 0,$$

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

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

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

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

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

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

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

– 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.

– 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 formula (3.2) 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.

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

Algorithm

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
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]
```

∗ 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 $1, 2, \ldots, j$.
* 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 3.9: Execution of the DP algorithm

We consider the UKP instance of Example 3.1 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.

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	$v[0][7] \leftarrow 0$
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	$\max\{v[0][7], p[1] + v[1][5]\} = \max\{0, 4+8\} = 12$
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	$\max\{v[1][7], p[2] + v[2][4]\} = \max\{12, 5+8\} = 13$

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	$\max\{v[2][7], p[3] + v[3][2]\} = \max\{13, 10 + 4\} = 14$
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	$\max\{v[3][7], p[4] + v[4][1]\} = \max\{14, 11 + 0\} = 14$

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.

    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

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
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
```

∗ 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 3.10: Computation of the additional information

We consider the UKP instance of Example 3.1 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.

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	$s[0][7] \leftarrow 0$
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	$p[1] + v[1][5] = 4 + 8 > v[0][7] = 0$: $s[1][7] \leftarrow 1$
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	$p[2] + v[2][4] = 5 + 8 > v[1][7] = 12$: $s[2][7] \leftarrow 1$

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	$p[3] + v[3][2] = 10 + 4 > v[2][7] = 13$: $s[3][7] \leftarrow 1$
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	$p[4] + v[4][1] = 11 + 0 \leq v[3][7] = 14$: $s[4][7] \leftarrow 0$

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

– 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.

Algorithm

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
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
```

Example 3.11: Computing an optimal solution

After the execution of the DP algorithm on the instance of Example 3.1, 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

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 $v(4,7) = 14$.

3.3.1 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. 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.

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 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 $\Theta(nc)$.

3.4 Typical mistakes

Warning

- **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 Example 3.1, $v(1) = 0$.
- **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.

3.5 In summary

Summary

- In the UKP an item type can be used any number of times, so the optimal value depends only on the residual capacity: $v(k) = \max_{j:\, w_j \le k}\{p_j + v(k - w_j)\}$.
- 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.

Dashboard of the UKP example. Example 3.1 ($c = 7$, four item types):

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

Chapter 4

DP for the bounded knapsack problem

Prerequisites

- From this module, the chapters «DP for the 0/1 knapsack problem» and «DP for the unbounded knapsack problem»: the recurrence $v(j, k)$ of the 0/1 knapsack problem (KP) and of the unbounded knapsack problem (UKP) 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 (Chapter 2) and the size of the input (Chapter 11).

Objectives

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

- say why the KP ($b_j = 1$) and the UKP ($b_j \geq \lfloor c/w_j \rfloor$) are special cases of the BKP, and write its recurrence;
- run the DP algorithm by hand, trying every feasible number d of copies of each item type;
- fill the matrix s of the numbers of copies and reconstruct an optimal solution from it;
- count the steps of the DP algorithm and say when $\Theta(nc\,b_{\max})$ is tight.

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

4.1 Instances and feasible solutions

Example 4.1: BKP instance

We consider a knapsack of capacity $c = 10$ and 3 item types, with the following profits, weights, and bounds:

	$j=1$	$j=2$	$j=3$
p_j	6	10	12
w_j	2	3	5
b_j	2	2	1

We refer to this BKP instance as Example 4.1.

The main difference with the 0/1 knapsack problem (KP) is that each item type may be selected more than once: variable x_j is no longer binary, but it is bounded by b_j. The BKP reduces to the KP when $b_j = 1$ for every item type j, and to the unbounded knapsack problem (UKP) when every bound b_j is large enough not to be binding.

- The number of possible vectors $(x_1, x_2, \ldots, x_n)$, with $x_j \in \{0, 1, \ldots, b_j\}$ for every item type j, is
$$\prod_{j=1}^{n} (b_j + 1).$$
Some of these vectors may not respect the knapsack capacity. Therefore, the number of feasible solutions is at most $\prod_{j=1}^{n}(b_j + 1)$, a quantity that grows exponentially with n. Enumerating all possible solutions quickly becomes impractical as n increases.

4.1.1 Encoding of feasible solutions

A *feasible solution* of the BKP is a choice of the number of copies of each item type, respecting both the bounds b_j and the knapsack capacity c.

- A BKP feasible solution can be described by the integer vector
$$(x_1, x_2, \ldots, x_n)$$
where each component has the following meaning:
$$x_j = \text{number of copies of item type } j \text{ selected}, \qquad \forall j \in \{1, 2, \ldots, n\}, \quad 0 \le x_j \le b_j.$$
Since the solution is feasible, the total weight of the selected items must not exceed the knapsack capacity:
$$\sum_{j=1}^{n} w_j x_j \le c.$$

The total profit of a feasible solution is

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

4.2 A 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 BKP subinstance given by:

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

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

$$v(j,k) = \begin{cases} 0 & \text{if } j = 0 \text{ or } k = 0, \\ \max\limits_{d \in \{0,1,\ldots,\min\{b_j, \lfloor k/w_j \rfloor\}\}} \left\{ d\,p_j + v(j-1, k - d\,w_j) \right\} & \text{if } 1 \le j \le n \\ & \text{and } 1 \le k \le c. \end{cases} \tag{4.1}$$

- The recursive decomposition process of formula (4.1) generates at most $b_j + 1$ **independent subproblems**, corresponding to the possible numbers $d \in \{0, 1, \ldots, \min\{b_j, \lfloor k/w_j \rfloor\}\}$ of copies of item type j inserted into a knapsack of capacity k, using only item types $1, 2, \ldots, j$.
- The base cases are $v(0, k) = 0$ since no item type is available, and $v(j, 0) = 0$ since no capacity is available.
- The optimal value $v(j, k)$ is obtained by trying every feasible number d of copies of item type j and selecting the one that maximizes the sum $d\,p_j + v(j-1, k - d\,w_j)$ of the profit of the d copies and the optimal value of the residual subproblem, restricted to item types $1, 2, \ldots, j-1$.
- Hence, an optimal solution for the subinstance with capacity k and available item types $\{1, 2, \ldots, j\}$ is built by combining the chosen number d of copies of item type j with an optimal solution of the corresponding residual subproblem.

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

– Different choices of the number of copies of an item type, 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 formula (4.1).

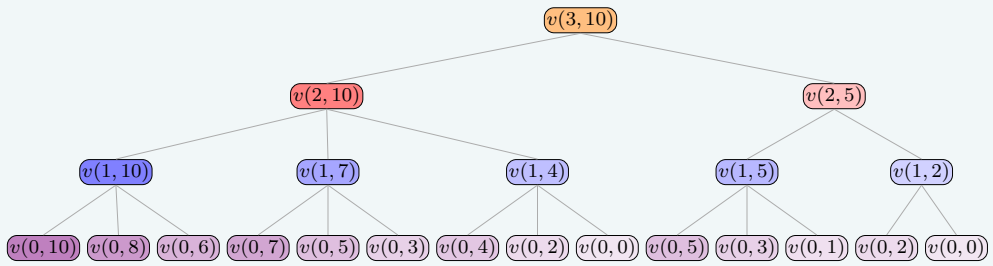
$\rightsquigarrow$ **Overlapping Subproblems verified!**

– A **first method** to compute the optimal value $v(n,c)$ of the BKP instance is the **top-down approach**. The method starts from the call with $j = n$ and $k = c$ and evaluates formula (4.1) recursively: each call considers item type j and tries every feasible number d of copies of it, until no item type is left or 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 4.2: Recursion tree

We consider the BKP instance of Example 4.1. This is the recursion tree of the top-down approach to compute $v(3,10)$:



The function v is called 22 times, and the same subproblems are solved multiple times: $v(0,5)$ is called 2 times, $v(0,3)$ is called 2 times, $v(0,2)$ is called 2 times, and $v(0,0)$ is called 2 times. All the other calls are made only once.

– The value $v(j,k)$ can be computed once the values $v(j-1,k')$, for every residual capacity $k' \leq k$, are known. 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 4.3: Computing the values of the v function

We consider the BKP instance of Example 4.1 with a knapsack of capacity $c = 10$ and the following profits, weights, and bounds:

	$j=1$	$j=2$	$j=3$
p_j	6	10	12
w_j	2	3	5
b_j	2	2	1

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 $v(1,4)$, trying $d=0,1,2$ copies of item type 1 (since $b_1=2$ and $\lfloor 4/2 \rfloor = 2$):

$$v(1,4) = \max\{v(0,4),\ p_1 + v(0,2),\ 2p_1 + v(0,0)\} = \max\{0,\ 6,\ 12\} = 12$$

* We can also compute $v(1,10)$, trying $d=0,1,2$ copies (since $b_1=2$ caps the number of copies even though $\lfloor 10/2 \rfloor = 5$):

$$v(1,10) = \max\{v(0,10),\ p_1 + v(0,8),\ 2p_1 + v(0,6)\} = \max\{0,\ 6,\ 12\} = 12$$

* In the same way we compute $v(1,7)$, $v(1,5)$ and $v(1,2)$:

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

$$v(1,5) = \max\{v(0,5),\ p_1 + v(0,3),\ 2p_1 + v(0,1)\} = \max\{0,\ 6,\ 12\} = 12$$

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

* Once we have computed $v(1,5)$ and $v(1,2)$, we can compute $v(2,5)$, trying $d=0,1$ copies of item type 2 (since $b_2=2$ but $\lfloor 5/3 \rfloor = 1$):

$$v(2,5) = \max\{v(1,5),\ p_2 + v(1,2)\} = \max\{12,\ 10+6\} = 16$$

* Once we have computed $v(1,10)$, $v(1,7)$ and $v(1,4)$, we can compute $v(2,10)$, trying $d=0,1,2$ copies of item type 2:

$$\begin{aligned} v(2,10) &= \max\{v(1,10),\ p_2 + v(1,7),\ 2p_2 + v(1,4)\} \\ &= \max\{12,\ 10+12,\ 20+12\} = 32 \end{aligned}$$

* Finally, once we have computed $v(2,10)$ and $v(2,5)$, we can compute $v(3,10)$, trying $d=0,1$ copies of item type 3 (since $b_3=1$):

$$v(3,10) = \max\{v(2,10),\ p_3 + v(2,5)\} = \max\{32,\ 12+16\} = 32$$

– 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, starting from the base cases $v(0,k) = 0$ and $v(j,0) = 0$ 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 BKP instance.

– A **second method** to compute the optimal value $v(n,c)$ of the BKP instance is the **bottom-up approach**. The method starts from the base cases and evaluates formula (4.1) 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 BKP instance is obtained.

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

Algorithm

Algorithm: BKP_DP

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

Output: The optimal value of the BKP instance

```
1  Function BKP_DP(n, p, w, b, c)
2      v := new array[0..n][0..c]
3      for k := 0 to c do
4          v[0][k] ← 0
5      end
6      for j := 1 to n do
7          for k := 0 to c do
8              m := 0
9              for d := 0 to min{b[j], ⌊k/w[j]⌋} do
10                 if d p[j] + v[j − 1][k − d w[j]] > m then
11                     m ← d p[j] + v[j − 1][k − d w[j]]
12                 end
13             end
14             v[j][k] ← m
15         end
16     end
17     return v[n][c]
```

* 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 type is available.
* The outer loop, over $j \in \{1,2,\ldots,n\}$, computes the rows of the matrix in increasing order of item type.

* The inner loop, over $k \in \{0, 1, \ldots, c\}$, computes the optimal value $v[j][k]$ for each capacity k.
* The innermost loop, over $d \in \{0, 1, \ldots, \min\{b[j], \lfloor k/w[j] \rfloor\}\}$, tries every feasible number of copies of item type j and keeps in m the largest value found so far.
* At the end of the innermost loop, $v[j][k] \leftarrow m$ stores the optimal value $v[j][k]$.
* Finally, the optimal value of the BKP instance is returned as $v[n][c]$.

Example 4.4: Execution of the DP algorithm

We consider the BKP instance of Example 4.1 with a knapsack of capacity $c = 10$ and the following profits, weights, and bounds:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	10	12
w_j	2	3	5
b_j	2	2	1

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	$v[0][5] \leftarrow 0$
0	6	0	$v[0][6] \leftarrow 0$
0	7	0	$v[0][7] \leftarrow 0$
0	8	0	$v[0][8] \leftarrow 0$
0	9	0	$v[0][9] \leftarrow 0$
0	10	0	$v[0][10] \leftarrow 0$
1	0	0	$\max\{v[0][0]\} = 0$
1	1	0	$\max\{v[0][1]\} = 0$
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	12	$\max\{v[0][4],\ p[1]+v[0][2],\ 2p[1]+v[0][0]\} = \max\{0, 6, 12\} = 12$
1	5	12	$\max\{v[0][5],\ p[1]+v[0][3],\ 2p[1]+v[0][1]\} = \max\{0, 6, 12\} = 12$
1	6	12	$\max\{v[0][6],\ p[1]+v[0][4],\ 2p[1]+v[0][2]\} = \max\{0, 6, 12\} = 12$
1	7	12	$\max\{v[0][7],\ p[1]+v[0][5],\ 2p[1]+v[0][3]\} = \max\{0, 6, 12\} = 12$
1	8	12	$\max\{v[0][8],\ p[1]+v[0][6],\ 2p[1]+v[0][4]\} = \max\{0, 6, 12\} = 12$
1	9	12	$\max\{v[0][9],\ p[1]+v[0][7],\ 2p[1]+v[0][5]\} = \max\{0, 6, 12\} = 12$
1	10	12	$\max\{v[0][10],\ p[1]+v[0][8],\ 2p[1]+v[0][6]\} = \max\{0, 6, 12\} = 12$

j	k	$v[j][k]$	Action
2	0	0	$\max\{v[1][0]\} = 0$
2	1	0	$\max\{v[1][1]\} = 0$
2	2	6	$\max\{v[1][2]\} = 6$
2	3	10	$\max\{v[1][3],\ p[2]+v[1][0]\} = \max\{6, 10+0\} = 10$
2	4	12	$\max\{v[1][4],\ p[2]+v[1][1]\} = \max\{12, 10+0\} = 12$
2	5	16	$\max\{v[1][5],\ p[2]+v[1][2]\} = \max\{12, 10+6\} = 16$
2	6	20	$\max\{v[1][6],\ p[2]+v[1][3],\ 2p[2]+v[1][0]\} = \max\{12, 16, 20\} = 20$
2	7	22	$\max\{v[1][7],\ p[2]+v[1][4],\ 2p[2]+v[1][1]\} = \max\{12, 22, 20\} = 22$
2	8	26	$\max\{v[1][8],\ p[2]+v[1][5],\ 2p[2]+v[1][2]\} = \max\{12, 22, 26\} = 26$
2	9	26	$\max\{v[1][9],\ p[2]+v[1][6],\ 2p[2]+v[1][3]\} = \max\{12, 22, 26\} = 26$
2	10	32	$\max\{v[1][10],\ p[2]+v[1][7],\ 2p[2]+v[1][4]\} = \max\{12, 22, 32\} = 32$
3	0	0	$\max\{v[2][0]\} = 0$
3	1	0	$\max\{v[2][1]\} = 0$
3	2	6	$\max\{v[2][2]\} = 6$
3	3	10	$\max\{v[2][3]\} = 10$
3	4	12	$\max\{v[2][4]\} = 12$
3	5	16	$\max\{v[2][5],\ p[3]+v[2][0]\} = \max\{16, 12+0\} = 16$
3	6	20	$\max\{v[2][6],\ p[3]+v[2][1]\} = \max\{20, 12+0\} = 20$
3	7	22	$\max\{v[2][7],\ p[3]+v[2][2]\} = \max\{22, 12+6\} = 22$
3	8	26	$\max\{v[2][8],\ p[3]+v[2][3]\} = \max\{26, 12+10\} = 26$
3	9	26	$\max\{v[2][9],\ p[3]+v[2][4]\} = \max\{26, 12+12\} = 26$
3	10	32	$\max\{v[2][10],\ p[3]+v[2][5]\} = \max\{32, 12+16\} = 32$

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

$v[j][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$
$j=0$	0	0	0	0	0	0	0	0	0	0	0
$j=1$	0	0	6	6	12	12	12	12	12	12	12
$j=2$	0	0	6	10	12	16	20	22	26	26	32
$j=3$	0	0	6	10	12	16	20	22	26	26	32

  - After the execution of the DP algorithm, the matrix v contains the optimal value $v[j][k]$ for every item type $j \in \{0, 1, \ldots, n\}$ and every capacity value $k \in \{0, 1, \ldots, c\}$.

- Step 2: computation of an optimal solution
  - To compute an optimal solution, i.e., the integer vector $(x_1, x_2, \ldots, x_n)$, it is necessary to store some additional information.
  - For each item type j and capacity value k, in addition to the optimal value $v[j][k]$, we store $s[j][k]$, the number of copies of item type j selected in an optimal solution for the subproblem restricted to item types $\{1, 2, \ldots, j\}$ and capacity k.

– More precisely, $s[j][k]$ is equal to the value d that maximizes the right-hand side of formula (4.1) when computing $v[j][k]$.

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

Algorithm: BKP_DP_InfoSol

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

Output: The optimal value of the BKP instance and the matrix of copy counts $s[0..n][0..c]$

```
1  Function BKP_DP_InfoSol(n, p, w, b, 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
6      for j := 1 to n do
7          for k := 0 to c do
8              m := 0; t := 0
9              for d := 0 to min{b[j], ⌊k/w[j]⌋} do
10                 if d p[j] + v[j − 1][k − d w[j]] > m then
11                     m ← d p[j] + v[j − 1][k − d w[j]]; t ← d
12                 end
13             end
14             v[j][k] ← m; s[j][k] ← t
15         end
16     end
17     return v[n][c], s
```

∗ The DP algorithm uses the additional temporary variable t to keep track of the decision made when m is updated. If d copies of item type j are selected, $t \leftarrow d$ stores this decision. At the end of the innermost loop, $s[j][k] \leftarrow t$ stores the number of copies of item type j used, together with the optimal value $v[j][k] \leftarrow m$.

Example 4.5: Computation of the additional information

We consider the BKP instance of Example 4.1 with a knapsack of capacity $c = 10$ and the following profits, weights, and bounds:

	$j = 1$	$j = 2$	$j = 3$
p_j	6	10	12
w_j	2	3	5
b_j	2	2	1

For each cell with $j \geq 1$ the table shows the values $d\,p[j] + v[j-1][k - d\,w[j]]$ for $d = 0, 1, \ldots, \min\{b[j], \lfloor k/w[j] \rfloor\}$, in this order, and the number t of copies stored in $s[j][k]$. In case of ties the smallest d is kept, because m is updated only when the value is strictly larger. The row $j = 0$ is $s[0][k] = 0$ for every k.

j	k	$s[j][k]$	Values for $d = 0, 1, \ldots$
1	0	0	0
1	1	0	0
1	2	1	0, 6
1	3	1	0, 6
1	4	2	0, 6, 12
1	5	2	0, 6, 12
1	6	2	0, 6, 12
1	7	2	0, 6, 12
1	8	2	0, 6, 12
1	9	2	0, 6, 12
1	10	2	0, 6, 12
2	0	0	0
2	1	0	0
2	2	0	6
2	3	1	6, 10
2	4	0	12, 10
2	5	1	12, 16
2	6	2	12, 16, 20
2	7	1	12, 22, 20
2	8	2	12, 22, 26
2	9	2	12, 22, 26
2	10	2	12, 22, 32
3	0	0	0
3	1	0	0
3	2	0	6
3	3	0	10
3	4	0	12
3	5	0	16, 12
3	6	0	20, 12
3	7	0	22, 18
3	8	0	26, 22
3	9	0	26, 24
3	10	0	32, 28

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

$s[j][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$
$j=0$	0	0	0	0	0	0	0	0	0	0	0
$j=1$	0	0	1	1	2	2	2	2	2	2	2
$j=2$	0	0	0	1	0	1	2	1	2	2	2
$j=3$	0	0	0	0	0	0	0	0	0	0	0

For example, $s[1][4] = 2$ since formula (4.1) selects $d = 2$ copies of item type 1 when computing $v[1][4] = 12$, and $s[2][10] = 2$ since $d = 2$ copies of item type 2

are selected when computing $v[2][10] = 32$. Since $s[3][10] = 0$, no copy of item type 3 is used in the optimal solution for the original instance.

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

$$x_j \leftarrow s[j][k], \qquad k \leftarrow k - s[j][k]\, w_j.$$

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.

Algorithm

Algorithm: BKP_Reconstruct

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

Output: An optimal solution of the BKP instance

```
1 Function BKP_Reconstruct(n, c, w, s)
2     x := new array[1..n]
3     k := c
4     for j := n downto 1 do
5         x[j] ← s[j][k]
6         k ← k − x[j] w[j]
7     end
8     return x
```

Example 4.6: Computing an optimal solution

After the execution of the DP algorithm on the instance of Example 4.1, we have the following matrix s:

$s[j][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$
$j=0$	0	0	0	0	0	0	0	0	0	0	0
$j=1$	0	0	1	1	2	2	2	2	2	2	2
$j=2$	0	0	0	1	0	1	2	1	2	2	2
$j=3$	0	0	0	0	0	0	0	0	0	0	0

We can reconstruct an optimal solution by starting from $j = 3$ and $k = 10$ as follows:

Iteration	j	k	$s[j][k]$	$x[1]$	$x[2]$	$x[3]$	Action
–	–	–	–	0	0	0	$k = 10$
1	3	10	0	0	0	0	$x[3] \leftarrow 0$, $k = 10$
2	2	10	2	0	2	0	$x[2] \leftarrow 2$, $k \leftarrow 10 - 2 \cdot 3 = 4$
3	1	4	2	2	2	0	$x[1] \leftarrow 2$, $k \leftarrow 4 - 2 \cdot 2 = 0$

Thus, the vector

$$(2, 2, 0)$$

is an optimal solution. The corresponding total weight is $2 \cdot 2 + 2 \cdot 3 = 4 + 6 = 10 = c$. The corresponding optimal value is $v(3, 10) = 32$.

4.2.1 Number of steps

- Let $b_{\max} = \max\{b[j] : j \in \{1, \ldots, n\}\}$, and let $S(n, c, b_{\max})$ denote the number of steps executed by `BKP_DP` in the worst case, where n is the number of item types and c is the knapsack capacity. We compute $S(n, c, b_{\max})$ by counting how many times the lines of the pseudo-code of `BKP_DP` are executed.

- For each pair (j, k) the innermost loop over d executes $\min\{b_j, \lfloor k/w_j \rfloor\} + 1$ iterations, which is at most $b_{\max} + 1$. We count $b_{\max} + 1$ iterations for every pair (j, k) and we assume that the condition at line 10 is true at every iteration, so that $S(n, c, b_{\max})$ is an upper bound on the number of steps. The number of executions of the relevant lines is, at most:

  - 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 9 is executed at most $n(c + 1)(b_{\max} + 1)$ times.
  - Line 10 is executed at most $n(c + 1)(b_{\max} + 1)$ times.
  - Line 11 is executed at most $n(c + 1)(b_{\max} + 1)$ times.
  - Line 14 is executed $n(c + 1)$ times.
  - Line 17 is executed 1 time.

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

  $$1 + (c + 1) + (c + 1) + n + n(c + 1) + n(c + 1) + 3n(c + 1)(b_{\max} + 1) + n(c + 1) + 1.$$

  Thus,

  $$S(n, c, b_{\max}) = 3n(c + 1)(b_{\max} + 1) + 3n(c + 1) + n + 2c + 4.$$

  Hence,

  $$S(n, c, b_{\max}) = \Theta(nc\, b_{\max}).$$

  Indeed,

  $$\begin{aligned}
  &\lim_{n,c,b_{\max} \to \infty} \frac{3n(c + 1)(b_{\max} + 1) + 3n(c + 1) + n + 2c + 4}{nc\, b_{\max}} \\
  &= \lim_{n,c,b_{\max} \to \infty} \left(\frac{3(c + 1)(b_{\max} + 1)}{c\, b_{\max}} + \frac{3(c + 1)}{c\, b_{\max}} + \frac{1}{c\, b_{\max}} + \frac{2}{n\, b_{\max}} + \frac{4}{nc\, b_{\max}} \right) \\
  &= 3 \in \mathbb{R}_{>0}.
  \end{aligned}$$

The DP algorithm executes

$$S(n, c, b_{\max}) = \Theta(nc\, b_{\max})$$

steps in the worst case.

- The version `BKP_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 $\Theta(nc\, b_{\max})$.
- Replacing every bound b_j by $\min\{b_j, \lfloor c/w_j \rfloor\}$ does not change the problem, so we can assume $b_{\max} \leq c$. Under this assumption the bound $\Theta(nc\, b_{\max})$ is tight: it is attained by the instances in which all the weights are equal to 1 and all the bounds are equal to $b_{\max}$. Indeed, the innermost loop then performs $\sum_{k=0}^{c}(\min\{b_{\max}, k\}+1) \geq (c+1)\, b_{\max}/2$ iterations for each item type.

4.3 Typical mistakes

Warning

- **Trying more than b_j copies.** The number of copies of item type j ranges in $\{0, 1, \ldots, \min\{b_j, \lfloor k/w_j \rfloor\}\}$: the bound and the capacity both matter.
- **Reconstructing with the wrong update.** After reading $s[j][k]$ the capacity decreases by $s[j][k]\, w_j$, not by w_j, and the item type index always decreases by one.
- **Counting the steps with $b_{\max}$ for every cell.** The innermost loop has $\min\{b_j, \lfloor k/w_j \rfloor\} + 1$ iterations, so $b_{\max} + 1$ is only an upper bound.
- **Breaking ties inconsistently.** The DP algorithm updates m only on a strictly larger value, so $s[j][k]$ keeps the smallest number of copies that attains the maximum.

4.4 In summary

Summary

- The BKP generalizes both the KP ($b_j = 1$) and the UKP ($b_j = \lfloor c/w_j \rfloor$); the recurrence tries every number d of copies of item type j.
- The matrix $v(j, k)$ is filled in $\Theta(nc\, b_{\max})$ steps, which is pseudo-polynomial.
- The matrix s stores the number of copies of each item type, and an optimal solution is read backwards from (n, c).
- Replacing b_j by $\min\{b_j, \lfloor c/w_j \rfloor\}$ keeps the problem unchanged and gives $b_{\max} \leq c$.

Dashboard of the BKP example. Example 4.1 ($c = 10$, three item types):

Method	Lower bound	Upper bound	Origin
DP algorithm	32	32	optimal value, solution $(2, 2, 0)$

Chapter 5

DP for the subset sum problem

Prerequisites

- From this module, the chapter «DP for the 0/1 knapsack problem»: the recurrence $v(j, k)$ of the 0/1 knapsack problem (KP) over items and capacities, the matrix s of the decisions and the reconstruction of a solution.
- From the module *Combinatorial Optimization Problems*, the chapter «Knapsack problems»: the subset sum problem and its solution as a 0/1 knapsack problem with $p_j = w_j$.
- From the module *Algorithms*: pseudocode conventions (Chapter 2) and the size of the input (Chapter 11).

Objectives

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

- write the recurrence $r(j, k) \in \{true, false\}$ of the reachable sums, with its base cases;
- run the DP algorithm by hand and read the answer of the SSP instance in $r(n, c)$;
- fill the matrix s and reconstruct a subset of total weight c when the answer is yes;
- relate $r(j, k)$ to the DP algorithm of the KP with $p_j = w_j$, and count the $\Theta(nc)$ steps.

The problem of this chapter is the subset sum problem (SSP): its definition and its assumptions are on its page.

5.1 A DP algorithm

Example 5.1: SSP instance

We consider a target value $c = 10$ and a set $\mathscr{I} = \{1, 2, 3, 4\}$ of 4 items ($n = 4$), with the following weights:

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
w_j	3	5	4	7

We refer to this SSP instance as Example 5.1.

- A subset $S \subseteq \mathscr{I}$ is described by the binary vector $(x_1, x_2, \ldots, x_n)$, where $x_j = 1$ if item j belongs to S and $x_j = 0$ otherwise.
- The answer to the SSP instance is **yes** if some vector satisfies $\sum_{j=1}^{n} w_j x_j = c$, and **no** otherwise.
- Step 1: computation of the answer

  For any $j \in \{0, 1, \ldots, n\}$ and $k \in \{0, 1, \ldots, c\}$, let $r(j, k) \in \{true, false\}$ be the answer of the SSP subinstance given by:

  1. a target value k,
  2. a set $\{1, 2, \ldots, j\}$ of available items,
  3. for each item $h \in \{1, 2, \ldots, j\}$, a weight w_h.

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

$$r(j,k) = \begin{cases} true & \text{if } k = 0, \\ false & \text{if } j = 0 \text{ and } k \geq 1, \\ r(j-1,k) & \text{if } j,k \geq 1 \text{ and } w_j > k, \\ r(j-1,k) \vee r(j-1,k-w_j) & \text{if } j,k \geq 1 \text{ and } w_j \leq k. \end{cases} \tag{5.1}$$

  - The symbol $\vee$ is the logical **or**: $r(j, k)$ is *true* if at least one of the two values is *true*.
  - The base cases are $r(j, 0) = true$, since the empty subset has total weight 0, and $r(0, k) = false$ for $k \geq 1$, since no item is available.

- A subset of $\{1, 2, \ldots, j\}$ with total weight k either does not contain item j, and then it is a subset of $\{1, 2, \ldots, j-1\}$ with total weight k, or it contains item j, and then the other items form a subset of $\{1, 2, \ldots, j-1\}$ with total weight $k - w_j$.
- Hence, the formula (5.1) generates at most 2 **independent subproblems**, and a subset for (j, k) is built from a subset of one of them by adding item j or not.

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

- Different decisions on the items can lead to the same pair (j, k), and therefore the same value $r(j, k)$ may be needed several times during the recursive evaluation of formula (5.1).

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

- The value $r(j, k)$ depends only on values $r(j-1, k')$ with $k' \leq k$. Therefore, the **bottom-up approach** computes the rows $j = 0, 1, \ldots, n$ in this order, each one for increasing k, and stores every value in a **lookup table**.

- The SSP can also be solved by the DP algorithm of the KP with $p_j = w_j$ for every item j. Its value $v(j, k)$ is the largest total weight, not larger than k, of a subset of $\{1, 2, \ldots, j\}$, and
$$r(j, k) = \mathit{true} \iff v(j, k) = k.$$
- The recurrence (5.1) stores only one bit per cell, and its logical operations replace the sums and the comparisons of the KP recurrence.

- Step 2: computation of a subset
  - When the answer is yes, a subset of total weight c is reconstructed from the matrix s: $s[j][k] = 1$ if item j is used to reach the value *true* of $r[j][k]$, and $s[j][k] = 0$ otherwise.
  - The DP algorithm prefers not to use item j: $s[j][k] = 1$ only if $r[j-1][k]$ is *false* and $r[j-1][k-w_j]$ is *true*.

This is the pseudo-code of the **DP algorithm using the bottom-up approach**, which stores the matrix s together with the matrix r:

Algorithm

Algorithm: SSP_DP

Input: An SSP instance given by n, the array $w[1..n]$, and the target value c
Output: The answer $r[n][c]$ of the SSP instance and the matrix $s[0..n][0..c]$

```
1  Function SSP_DP(n, w, c)
2      r := new array[0..n][0..c]; s := new array[0..n][0..c]
3      r[0][0] ← true; s[0][0] ← 0
4      for k := 1 to c do
5          r[0][k] ← false; s[0][k] ← 0
6      end
7      for j := 1 to n do
8          for k := 0 to c do
9              r[j][k] ← r[j − 1][k]; s[j][k] ← 0
10             if r[j][k] = false and w[j] ≤ k and r[j − 1][k − w[j]] = true then
11                 r[j][k] ← true; s[j][k] ← 1
12             end
13         end
14     end
15     return r[n][c], s
```

– The matrix $r[0..n][0..c]$ is the **lookup table** of the answers, and the matrix $s[0..n][0..c]$ stores the decisions.

– Lines 3–5 fill the row $j = 0$: only the target value 0 is reachable without items.

– For each pair (j, k), the DP algorithm first copies the answer without item j, and then uses item j only if this answer is *false* and item j makes the value k reachable.

– Finally, the answer of the SSP instance is returned as $r[n][c]$, together with the matrix s.

Example 5.2: Execution of the DP algorithm

We consider the SSP instance of Example 5.1, with target value $c = 10$ and weights $w = (3, 5, 4, 7)$.

For each step of the DP algorithm with $j \geq 1$, the table shows the values of j and k, the cells $r[j][k]$ and $s[j][k]$, and the operation performed; T stands for *true* and F for *false*. The row $j = 0$ is $r[0][0] =$ T and $r[0][k] =$ F for $k \geq 1$, with $s[0][k] = 0$.

j	k	$r[j][k]$	$s[j][k]$	Action
1	0	T	0	$w_1 = 3 > 0$: $r[0][0] = \mathrm{T}$
1	1	F	0	$w_1 = 3 > 1$: $r[0][1] = \mathrm{F}$
1	2	F	0	$w_1 = 3 > 2$: $r[0][2] = \mathrm{F}$
1	3	T	1	$r[0][3] \vee r[0][0] = \mathrm{F} \vee \mathrm{T} = \mathrm{T}$
1	4	F	0	$r[0][4] \vee r[0][1] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
1	5	F	0	$r[0][5] \vee r[0][2] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
1	6	F	0	$r[0][6] \vee r[0][3] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
1	7	F	0	$r[0][7] \vee r[0][4] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
1	8	F	0	$r[0][8] \vee r[0][5] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
1	9	F	0	$r[0][9] \vee r[0][6] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
1	10	F	0	$r[0][10] \vee r[0][7] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
2	0	T	0	$w_2 = 5 > 0$: $r[1][0] = \mathrm{T}$
2	1	F	0	$w_2 = 5 > 1$: $r[1][1] = \mathrm{F}$
2	2	F	0	$w_2 = 5 > 2$: $r[1][2] = \mathrm{F}$
2	3	T	0	$w_2 = 5 > 3$: $r[1][3] = \mathrm{T}$
2	4	F	0	$w_2 = 5 > 4$: $r[1][4] = \mathrm{F}$
2	5	T	1	$r[1][5] \vee r[1][0] = \mathrm{F} \vee \mathrm{T} = \mathrm{T}$
2	6	F	0	$r[1][6] \vee r[1][1] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
2	7	F	0	$r[1][7] \vee r[1][2] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
2	8	T	1	$r[1][8] \vee r[1][3] = \mathrm{F} \vee \mathrm{T} = \mathrm{T}$
2	9	F	0	$r[1][9] \vee r[1][4] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
2	10	F	0	$r[1][10] \vee r[1][5] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$

j	k	$r[j][k]$	$s[j][k]$	Action
3	0	T	0	$w_3 = 4 > 0$: $r[2][0] = \mathrm{T}$
3	1	F	0	$w_3 = 4 > 1$: $r[2][1] = \mathrm{F}$
3	2	F	0	$w_3 = 4 > 2$: $r[2][2] = \mathrm{F}$
3	3	T	0	$w_3 = 4 > 3$: $r[2][3] = \mathrm{T}$
3	4	T	1	$r[2][4] \vee r[2][0] = \mathrm{F} \vee \mathrm{T} = \mathrm{T}$
3	5	T	0	$r[2][5] \vee r[2][1] = \mathrm{T} \vee \mathrm{F} = \mathrm{T}$
3	6	F	0	$r[2][6] \vee r[2][2] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
3	7	T	1	$r[2][7] \vee r[2][3] = \mathrm{F} \vee \mathrm{T} = \mathrm{T}$
3	8	T	0	$r[2][8] \vee r[2][4] = \mathrm{T} \vee \mathrm{F} = \mathrm{T}$
3	9	T	1	$r[2][9] \vee r[2][5] = \mathrm{F} \vee \mathrm{T} = \mathrm{T}$
3	10	F	0	$r[2][10] \vee r[2][6] = \mathrm{F} \vee \mathrm{F} = \mathrm{F}$
4	0	T	0	$w_4 = 7 > 0$: $r[3][0] = \mathrm{T}$
4	1	F	0	$w_4 = 7 > 1$: $r[3][1] = \mathrm{F}$
4	2	F	0	$w_4 = 7 > 2$: $r[3][2] = \mathrm{F}$
4	3	T	0	$w_4 = 7 > 3$: $r[3][3] = \mathrm{T}$
4	4	T	0	$w_4 = 7 > 4$: $r[3][4] = \mathrm{T}$
4	5	T	0	$w_4 = 7 > 5$: $r[3][5] = \mathrm{T}$
4	6	F	0	$w_4 = 7 > 6$: $r[3][6] = \mathrm{F}$
4	7	T	0	$r[3][7] \vee r[3][0] = \mathrm{T} \vee \mathrm{T} = \mathrm{T}$
4	8	T	0	$r[3][8] \vee r[3][1] = \mathrm{T} \vee \mathrm{F} = \mathrm{T}$
4	9	T	0	$r[3][9] \vee r[3][2] = \mathrm{T} \vee \mathrm{F} = \mathrm{T}$
4	10	T	1	$r[3][10] \vee r[3][3] = \mathrm{F} \vee \mathrm{T} = \mathrm{T}$

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

$r[j][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$
$j=0$	T	F	F	F	F	F	F	F	F	F	F
$j=1$	T	F	F	T	F	F	F	F	F	F	F
$j=2$	T	F	F	T	F	T	F	F	T	F	F
$j=3$	T	F	F	T	T	T	F	T	T	T	F
$j=4$	T	F	F	T	T	T	F	T	T	T	T

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

$s[j][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$
$j=0$	0	0	0	0	0	0	0	0	0	0	0
$j=1$	0	0	0	1	0	0	0	0	0	0	0
$j=2$	0	0	0	0	0	1	0	0	1	0	0
$j=3$	0	0	0	0	1	0	0	1	0	1	0
$j=4$	0	0	0	0	0	0	0	0	0	0	1

Since $r[4][10] = \mathrm{T}$, the answer to the SSP instance is **yes**. For example, $s[4][10] = 1$ since $r[3][10] = \mathrm{F}$ and $r[3][3] = \mathrm{T}$: item 4 is needed to reach the target value 10.

– Once the matrix s has been computed and $r[n][c]$ is *true*, a subset of total weight c is reconstructed by scanning the items from $j = n$ down to $j = 1$, starting from $k = c$ and $x_j = 0$ for every item j:

$$x_j \leftarrow s[j][k], \qquad k \leftarrow k - s[j][k]\, w_j.$$

– The reconstruction ends with $k = 0$, since every step keeps $r[j-1][k]$ equal to *true*.

Algorithm

Algorithm: SSP_Reconstruct

Input: The number n of items, the target value c, the array $w[1..n]$, and the matrix $s[0..n][0..c]$, with $r[n][c] = true$

Output: A subset of total weight c, as a binary vector

```
1 Function SSP_Reconstruct(n, c, w, s)
2     x := new array[1..n]
3     k := c
4     for j := n downto 1 do
5         x[j] ← s[j][k]
6         k ← k − x[j] w[j]
7     end
8     return x
```

Example 5.3: Computing a subset

After the execution of the DP algorithm on the instance of Example 5.1, we start from $j = 4$ and $k = 10$:

Iteration	j	k	$s[j][k]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	Action
–	–	–	–	0	0	0	0	$k = 10$
1	4	10	1	0	0	0	1	$x[4] \leftarrow 1$, $k \leftarrow 10 - 7 = 3$
2	3	3	0	0	0	0	1	$x[3] \leftarrow 0$, $k = 3$
3	2	3	0	0	0	0	1	$x[2] \leftarrow 0$, $k = 3$
4	1	3	1	1	0	0	1	$x[1] \leftarrow 1$, $k \leftarrow 3 - 3 = 0$

The answer to the SSP instance of Example 5.1 is **yes**: the vector $(1, 0, 0, 1)$ selects the subset $\{1, 4\}$, whose total weight is $3 + 7 = c$.

5.1.1 Number of steps

- Let $S(n, c)$ denote the number of steps executed by `SSP_DP` in the worst case. We count how many times each line of the pseudo-code is executed, assuming that the condition at line 10 is always true:
  - Lines 2 and 3 are executed 1 time each.
  - Line 4 is executed c times, and line 5 c times.
  - Line 7 is executed n times.
  - Lines 8, 9, 10 and 11 are executed $n(c + 1)$ times each.
  - Line 15 is executed 1 time.
- Therefore, the total number of steps in the worst case is

$$S(n, c) = 1 + 1 + 2c + n + 4n(c + 1) + 1 = 4n(c + 1) + n + 2c + 3.$$

Hence,

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

since

$$\lim_{n,c\to\infty} \frac{4n(c + 1) + n + 2c + 3}{nc} = \lim_{n,c\to\infty} \left(4 + \frac{4}{c} + \frac{1}{c} + \frac{2}{n} + \frac{3}{nc}\right) = 4 \in \mathbb{R}_{>0}.$$

The DP algorithm for the SSP executes $S(n, c) = \Theta(nc)$ steps in the worst case, the same number as the DP algorithm of the KP.

- The reconstruction executes $\Theta(n)$ steps, one for each item.
- The number of steps contains the target value c, a number of the input, and not only the size of the input: it is **pseudo-polynomial**.

5.2 Typical mistakes

Warning

- **Setting $r(j,0) =$ *false*.** The target value 0 is always reachable with the empty subset; with $r(j,0) = \mathit{false}$ no value would ever become *true*.
- **Setting $r(0,k) =$ *true* for $k \geq 1$.** Without items, only the target value 0 is reachable.
- **Reading the answer in the wrong cell.** The answer is $r(n,c)$; a *true* value in another cell of the last row says that a different target value is reachable.
- **Reconstructing when the answer is no.** If $r(n,c)$ is *false*, no subset has total weight c, and the matrix s gives nothing to reconstruct.

5.3 In summary

Summary

- The recurrence $r(j,k) = r(j-1,k) \vee r(j-1,k-w_j)$ tells whether the target value k is reachable with the items $1,2,\ldots,j$.
- The base cases are $r(j,0) = \mathit{true}$ and $r(0,k) = \mathit{false}$ for $k \geq 1$; the answer is $r(n,c)$.
- The matrix s stores whether item j is used, and a subset of total weight c is read backwards from (n,c).
- The DP algorithm executes $\Theta(nc)$ steps, as the DP algorithm of the KP with $p_j = w_j$.

Dashboard of the SSP example. Example 5.1 ($c = 10$, four items):

Method	Answer	Origin
DP algorithm	yes	$r(4,10) = \mathit{true}$, subset $\{1,4\}$

Chapter 6

DP for the multiple-choice knapsack problem

Prerequisites

- From this module, the chapter «DP for the 0/1 knapsack problem»: the recurrence $v(j, k)$ of the 0/1 knapsack problem (KP) over items and capacities, the matrix s of the decisions and the reconstruction of a solution.
- From the module *Combinatorial Optimization Problems*, the chapter «Knapsack problems»: the multiple-choice knapsack problem, its classes $\mathscr{J}_h$ and its feasibility condition.
- From the module *Algorithms*: pseudocode conventions (Chapter 2) and the size of the input (Chapter 11).

Objectives

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

- write the recurrence $v(h, k)$ over the classes and the capacities, with the value $-\infty$ for the subinstances without a feasible solution;
- run the DP algorithm by hand, trying every item of class h that fits;
- fill the matrix s of the chosen items and reconstruct an optimal solution, one item per class;
- count the $\Theta(nc)$ steps of the DP algorithm.

The problem of this chapter is the multiple-choice knapsack problem (MCKP): its definition and its assumptions are on its page.

6.1 A DP algorithm

Example 6.1: MCKP instance

We consider a knapsack of capacity $c = 10$ and a set $\mathscr{I} = \{1, 2, \ldots, 7\}$ of 7 items ($n = 7$), partitioned into $q = 3$ classes: $\mathscr{J}_1 = \{1, 2\}$, $\mathscr{J}_2 = \{3, 4, 5\}$ and $\mathscr{J}_3 = \{6, 7\}$. The profits and weights are:

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$
p_j	4	8	3	5	7	3	7
w_j	1	4	1	2	5	2	3

We refer to this MCKP instance as Example 6.1.

- A feasible solution chooses exactly one item in each class, so it is described by the selected subset $S \subseteq \mathscr{I}$, with $|S \cap \mathscr{J}_h| = 1$ for every class h, or by the binary vector $(x_1, x_2, \ldots, x_n)$ of S.

- The classes are the stages of the DP algorithm: class h is decided after the classes $1, 2, \ldots, h-1$, and its decision is the item chosen in it.

- Step 1: computation of the optimal value

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

  1. a knapsack of capacity k,
  2. the classes $\mathscr{J}_1, \mathscr{J}_2, \ldots, \mathscr{J}_h$,
  3. for each item $j \in \mathscr{J}_1 \cup \mathscr{J}_2 \cup \cdots \cup \mathscr{J}_h$, a profit p_j and a weight w_j,

  with $v(h, k) = -\infty$ if this subinstance has no feasible solution.

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

$$v(h,k) = \begin{cases} 0 & \text{if } h = 0, \\ -\infty & \text{if } h \geq 1 \text{ and } \forall j \in \mathscr{J}_h\colon\ w_j > k, \\ \max\limits_{j \in \mathscr{J}_h\colon w_j \leq k} \{p_j + v(h-1, k-w_j)\} & \text{if } h \geq 1 \text{ and } \exists j \in \mathscr{J}_h\colon\ w_j \leq k. \end{cases} \tag{6.1}$$

  – The value $-\infty$ follows the rules $p_j + (-\infty) = -\infty$ and $\max\{-\infty, a\} = a$: a subinstance without feasible solutions never gives a better value.

- The base cases are $v(0, k) = 0$ for every k: with no class, the empty subset is feasible and has profit 0, whatever the capacity.
- The formula (6.1) generates at most $|\mathscr{J}_h|$ **independent subproblems**, one for each item j of class h that fits in capacity k: the residual subproblem has the classes $1, 2, \ldots, h-1$ and the capacity $k - w_j$.
- Hence, an optimal solution for (h, k) is built by combining the best item j of class h with an optimal solution of its residual subproblem.

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

- Different choices in the classes $h+1, h+2, \ldots, q$ can leave the same residual capacity, and therefore the same value $v(h, k)$ may be needed several times during the recursive evaluation of formula (6.1).

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

- The value $v(h, k)$ depends only on values $v(h-1, k')$ with $k' < k$. Therefore, the **bottom-up approach** computes the rows $h = 0, 1, \ldots, q$ in this order, each one for increasing k, and stores every value in a **lookup table**.
- The optimal value of the MCKP instance is $v(q, c)$, and the instance is infeasible if and only if $v(q, c) = -\infty$.

- Step 2: computation of an optimal solution
  - For each class h and capacity k, we store $s[h][k]$, the item of class h chosen in an optimal solution of the subinstance (h, k), that is, the item that maximizes the right-hand side of formula (6.1).
  - If $v[h][k] = -\infty$, no item is chosen and $s[h][k] = 0$.

This is the pseudo-code of the **DP algorithm using the bottom-up approach**, which stores the matrix s together with the matrix v:

Algorithm

Algorithm: MCKP_DP

Input: An MCKP instance given by q, the classes $\mathscr{J}_1, \ldots, \mathscr{J}_q$, arrays $p[1..n]$, $w[1..n]$, and capacity c

Output: The optimal value of the MCKP instance and the matrix $s[0..q][0..c]$

```
1  Function MCKP_DP(q, J, p, w, c)
2      v := new array[0..q][0..c]; s := new array[0..q][0..c]
3      for k := 0 to c do
4          v[0][k] ← 0; s[0][k] ← 0
5      end
6      for h := 1 to q do
7          for k := 0 to c do
8              m := −∞; t := 0
9              foreach j ∈ J_h do
10                 if w[j] ≤ k and p[j] + v[h − 1][k − w[j]] > m then
11                     m ← p[j] + v[h − 1][k − w[j]]; t ← j
12                 end
13             end
14             v[h][k] ← m; s[h][k] ← t
15         end
16     end
17     return v[q][c], s
```

- The variable m starts from $-\infty$, not from 0: exactly one item of class h must be chosen, so the value of an empty choice is not available.
- The innermost loop tries every item j of class h that fits, keeps in m the largest value found so far, and in t the item that gives it.
- If no item of class h gives a value larger than $-\infty$, the cell keeps $v[h][k] = -\infty$ and $s[h][k] = 0$.
- In case of ties the item met first in $\mathscr{J}_h$ is kept, because m is updated only when the value is strictly larger.

Example 6.2: Execution of the DP algorithm

We consider the MCKP instance of Example 6.1, with capacity $c = 10$ and classes $\mathscr{J}_1 = \{1, 2\}$, $\mathscr{J}_2 = \{3, 4, 5\}$, $\mathscr{J}_3 = \{6, 7\}$.

For each step of the DP algorithm with $h \geq 1$, the table shows the values of h and k, the cells $v[h][k]$ and $s[h][k]$, and the operation performed. The row $h = 0$ is $v[0][k] = 0$ and $s[0][k] = 0$ for every k.

h	k	$v[h][k]$	$s[h][k]$	Action
1	0	$-\infty$	0	no item of $\mathscr{J}_1$ fits: $-\infty$
1	1	4	1	$\max\{p_1+v[0][0]\} = \max\{4\} = 4$
1	2	4	1	$\max\{p_1+v[0][1]\} = \max\{4\} = 4$
1	3	4	1	$\max\{p_1+v[0][2]\} = \max\{4\} = 4$
1	4	8	2	$\max\{p_1+v[0][3],\ p_2+v[0][0]\} = \max\{4,8\} = 8$
1	5	8	2	$\max\{p_1+v[0][4],\ p_2+v[0][1]\} = \max\{4,8\} = 8$
1	6	8	2	$\max\{p_1+v[0][5],\ p_2+v[0][2]\} = \max\{4,8\} = 8$
1	7	8	2	$\max\{p_1+v[0][6],\ p_2+v[0][3]\} = \max\{4,8\} = 8$
1	8	8	2	$\max\{p_1+v[0][7],\ p_2+v[0][4]\} = \max\{4,8\} = 8$
1	9	8	2	$\max\{p_1+v[0][8],\ p_2+v[0][5]\} = \max\{4,8\} = 8$
1	10	8	2	$\max\{p_1+v[0][9],\ p_2+v[0][6]\} = \max\{4,8\} = 8$
2	0	$-\infty$	0	no item of $\mathscr{J}_2$ fits: $-\infty$
2	1	$-\infty$	0	$\max\{p_3+v[1][0]\} = \max\{-\infty\} = -\infty$
2	2	7	3	$\max\{p_3+v[1][1],\ p_4+v[1][0]\} = \max\{7,-\infty\} = 7$
2	3	9	4	$\max\{p_3+v[1][2],\ p_4+v[1][1]\} = \max\{7,9\} = 9$
2	4	9	4	$\max\{p_3+v[1][3],\ p_4+v[1][2]\} = \max\{7,9\} = 9$
2	5	11	3	$\max\{p_3+v[1][4],\ p_4+v[1][3],\ p_5+v[1][0]\} = \max\{11,9,-\infty\} = 11$
2	6	13	4	$\max\{p_3+v[1][5],\ p_4+v[1][4],\ p_5+v[1][1]\} = \max\{11,13,11\} = 13$
2	7	13	4	$\max\{p_3+v[1][6],\ p_4+v[1][5],\ p_5+v[1][2]\} = \max\{11,13,11\} = 13$
2	8	13	4	$\max\{p_3+v[1][7],\ p_4+v[1][6],\ p_5+v[1][3]\} = \max\{11,13,11\} = 13$
2	9	15	5	$\max\{p_3+v[1][8],\ p_4+v[1][7],\ p_5+v[1][4]\} = \max\{11,13,15\} = 15$
2	10	15	5	$\max\{p_3+v[1][9],\ p_4+v[1][8],\ p_5+v[1][5]\} = \max\{11,13,15\} = 15$

h	k	$v[h][k]$	$s[h][k]$	Action
3	0	$-\infty$	0	no item of $\mathscr{J}_3$ fits: $-\infty$
3	1	$-\infty$	0	no item of $\mathscr{J}_3$ fits: $-\infty$
3	2	$-\infty$	0	$\max\{p_6+v[2][0]\} = \max\{-\infty\} = -\infty$
3	3	$-\infty$	0	$\max\{p_6+v[2][1],\ p_7+v[2][0]\} = \max\{-\infty,-\infty\} = -\infty$
3	4	10	6	$\max\{p_6+v[2][2],\ p_7+v[2][1]\} = \max\{10,-\infty\} = 10$
3	5	14	7	$\max\{p_6+v[2][3],\ p_7+v[2][2]\} = \max\{12,14\} = 14$
3	6	16	7	$\max\{p_6+v[2][4],\ p_7+v[2][3]\} = \max\{12,16\} = 16$
3	7	16	7	$\max\{p_6+v[2][5],\ p_7+v[2][4]\} = \max\{14,16\} = 16$
3	8	18	7	$\max\{p_6+v[2][6],\ p_7+v[2][5]\} = \max\{16,18\} = 18$
3	9	20	7	$\max\{p_6+v[2][7],\ p_7+v[2][6]\} = \max\{16,20\} = 20$
3	10	20	7	$\max\{p_6+v[2][8],\ p_7+v[2][7]\} = \max\{16,20\} = 20$

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

$v[h][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$
$h=0$	0	0	0	0	0	0	0	0	0	0	0
$h=1$	$-\infty$	4	4	4	8	8	8	8	8	8	8
$h=2$	$-\infty$	$-\infty$	7	9	9	11	13	13	13	15	15
$h=3$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	10	14	16	16	18	20	20

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

$s[h][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$
$h=0$	0	0	0	0	0	0	0	0	0	0	0
$h=1$	0	1	1	1	2	2	2	2	2	2	2
$h=2$	0	0	3	4	4	3	4	4	4	5	5
$h=3$	0	0	0	0	6	7	7	7	7	7	7

For example, $v[2][1] = -\infty$: every choice of one item in $\mathscr{J}_1$ and one in $\mathscr{J}_2$ weighs at least $w_1 + w_3 = 2$. The optimal value of the instance is $v[3][10] = 20$, attained with item $s[3][10] = 7$ of class 3.

– Once the matrix s has been computed and $v[q][c] > -\infty$, an optimal solution is reconstructed by scanning the classes from $h = q$ down to $h = 1$, starting from $k = c$ and $x_j = 0$ for every item j:

$$j \leftarrow s[h][k], \qquad x_j \leftarrow 1, \qquad k \leftarrow k - w_j.$$

– Every class gives exactly one item, so the reconstruction selects q items.

Algorithm

Algorithm: MCKP_Reconstruct

Input: The number q of classes, the capacity c, the array $w[1..n]$, and the matrix $s[0..q][0..c]$, with $v[q][c] > -\infty$

Output: An optimal solution of the MCKP instance, as a binary vector

```
1 Function MCKP_Reconstruct(q, c, w, s)
2     x := new array[1..n], with x[j] = 0 for every j
3     k := c
4     for h := q downto 1 do
5         j ← s[h][k]; x[j] ← 1
6         k ← k − w[j]
7     end
8     return x
```

Example 6.3: Computing an optimal solution

After the execution of the DP algorithm on the instance of Example 6.1, we start from $h = 3$ and $k = 10$:

Iteration	h	k	$j = s[h][k]$	Selection	Capacity
–	–	–	–	–	$k = 10$
1	3	10	7	$x[7] \leftarrow 1$	$k \leftarrow 10 - 3 = 7$
2	2	7	4	$x[4] \leftarrow 1$	$k \leftarrow 7 - 2 = 5$
3	1	5	2	$x[2] \leftarrow 1$	$k \leftarrow 5 - 4 = 1$

An optimal solution of the MCKP instance of Example 6.1 is $S = \{2, 4, 7\}$, that is, the vector $x = (0, 1, 0, 1, 0, 0, 1)$. Its total weight is $4 + 2 + 3 = 9 \leq c$ and its optimal value is $v(3, 10) = 20$.

6.1.1 Number of steps

- Let $S(n, q, c)$ denote the number of steps executed by `MCKP_DP` in the worst case. The innermost loop runs over the items of class h, so over all the classes it runs $|\mathscr{J}_1| + |\mathscr{J}_2| + \cdots + |\mathscr{J}_q| = n$ times for each capacity k. Assuming that the condition at line 10 is always true:
  - Lines 2 and 17 are executed 1 time each.
  - Lines 3 and 4 are executed $c + 1$ times each.
  - Line 6 is executed q times.
  - Lines 7, 8 and 14 are executed $q(c + 1)$ times each.
  - Lines 9, 10 and 11 are executed $n(c + 1)$ times each.
- Therefore, the total number of steps in the worst case is

$$S(n, q, c) = 3n(c + 1) + 3q(c + 1) + q + 2c + 4.$$

  Since every class contains at least one item, $1 \leq q \leq n$, and

$$3n(c + 1) \;\leq\; S(n, q, c) \;\leq\; 6n(c + 1) + n + 2c + 4.$$

  Both bounds are $\Theta(nc)$, hence $S(n, q, c) = \Theta(nc)$.

  The DP algorithm for the MCKP executes

$$S(n, q, c) = \Theta\big(c\,(|\mathscr{J}_1| + |\mathscr{J}_2| + \cdots + |\mathscr{J}_q|)\big) = \Theta(nc)$$

  steps in the worst case.
- The reconstruction executes $\Theta(q)$ steps, one for each class.

6.2 Typical mistakes

Warning

- **Starting m from 0 instead of $-\infty$.** With $m = 0$ a class with no item that fits would be skipped, and the solution would not contain exactly one item per class.
- **Adding the option «no item of class h».** The term $v(h-1, k)$ of the KP recurrence has no place here: every class gives exactly one item.
- **Setting $v(0, k) = -\infty$ for $k \geq 1$.** With no class the empty subset is feasible for every capacity: the capacity does not have to be filled.
- **Reconstructing an infeasible instance.** If $v(q, c) = -\infty$ the instance has no feasible solution, and $s[q][c] = 0$ gives no item to select.

6.3 In summary

Summary

- The stages of the DP algorithm are the classes: $v(h, k) = \max_{j \in \mathscr{J}_h : w_j \leq k} \{p_j + v(h-1, k-w_j)\}$, with $v(0, k) = 0$.
- The value $-\infty$ marks the subinstances without a feasible solution, and the instance is infeasible if and only if $v(q, c) = -\infty$.
- The matrix s stores the item chosen in each class, and an optimal solution is read backwards from (q, c), one item per class.
- The DP algorithm executes $\Theta(c \sum_{h=1}^{q} |\mathscr{J}_h|) = \Theta(nc)$ steps.

Dashboard of the MCKP example. Example 6.1 ($c = 10$, three classes):

Method	Lower bound	Upper bound	Origin
DP algorithm	20	20	optimal value, solution $S = \{2, 4, 7\}$

Chapter 7

DP for the multidimensional knapsack problem

Prerequisites

- From this module, the chapter «DP for the 0/1 knapsack problem»: the recurrence $v(j, k)$ of the 0/1 knapsack problem (KP) over items and capacities, the matrix s of the decisions and the reconstruction of a solution.
- From the module *Combinatorial Optimization Problems*, the chapter «Knapsack problems»: the multidimensional knapsack problem, with one capacity c_d and one weight w_{dj} for every dimension d.
- From the module *Algorithms*: pseudocode conventions (Chapter 2) and the size of the input (Chapter 11).

Objectives

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

- write the recurrence $v(j, k_1, \ldots, k_m)$ over the items and the vector of the capacities;
- run the DP algorithm by hand on an instance with $m = 2$ dimensions, one matrix for each item;
- reconstruct an optimal solution from the tables s;
- count the $\Theta(m\, n \prod_{d=1}^{m}(c_d + 1))$ steps and explain why the number of states grows exponentially with m.

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

7.1 A DP algorithm

- We write m for the number of dimensions, so $\mathscr{D} = \{1, 2, \ldots, m\}$, because in the DP chapters the letter k denotes a capacity. Item j has the **weight vector** $\boldsymbol{w}_j = (w_{1j}, w_{2j}, \ldots, w_{mj})$, and the knapsack has the capacity vector $\boldsymbol{c} = (c_1, c_2, \ldots, c_m)$.
- For two vectors, $\boldsymbol{w}_j \leq \boldsymbol{k}$ means $w_{dj} \leq k_d$ for every dimension $d \in \mathscr{D}$.
- A solution is described by the binary vector $(x_1, x_2, \ldots, x_n)$, where $x_j = 1$ if item j is selected; it is feasible if $\sum_{j=1}^{n} w_{dj} x_j \leq c_d$ for every $d \in \mathscr{D}$.

Example 7.1: MDKP instance

We consider a knapsack with $m = 2$ dimensions and capacities $c_1 = 5$ and $c_2 = 4$, and a set $\mathscr{I} = \{1, 2, 3, 4\}$ of 4 items ($n = 4$), with the following profits and weights:

	$j=1$	$j=2$	$j=3$	$j=4$
p_j	9	8	3	9
w_{1j}	1	4	2	3
w_{2j}	2	1	1	4

We refer to this MDKP instance as Example 7.1.

- Step 1: computation of the optimal value

For any $j \in \{0, 1, \ldots, n\}$ and any vector $\boldsymbol{k} = (k_1, k_2, \ldots, k_m)$ with $k_d \in \{0, 1, \ldots, c_d\}$ for every $d \in \mathscr{D}$, let $v(j, \boldsymbol{k}) = v(j, k_1, \ldots, k_m)$ be the optimal value of the MDKP subinstance given by:

1. a knapsack with capacity k_d in each dimension $d \in \mathscr{D}$,
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 vector $\boldsymbol{w}_h$.

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

$$v(j, \boldsymbol{k}) = \begin{cases} 0 & \text{if } j = 0, \\ v(j-1, \boldsymbol{k}) & \text{if } j \geq 1 \text{ and } w_{dj} > k_d \text{ for some } d, \\ \max\left\{ v(j-1, \boldsymbol{k}), \; p_j + v(j-1, \boldsymbol{k} - \boldsymbol{w}_j) \right\} & \text{if } j \geq 1 \text{ and } \boldsymbol{w}_j \leq \boldsymbol{k}. \end{cases} \tag{7.1}$$

- With $m = 1$ the formula (7.1) is the recurrence of the KP: the only change is that the capacity k becomes the vector $\boldsymbol{k}$, and «item j fits» means $\boldsymbol{w}_j \leq \boldsymbol{k}$.
- The base cases are $v(0, \boldsymbol{k}) = 0$ for every vector $\boldsymbol{k}$, since no item is available.
- The formula generates at most 2 **independent subproblems**: item j is not selected, and the residual subproblem has the same vector $\boldsymbol{k}$, or it is selected, and the residual subproblem has the vector $\boldsymbol{k} - \boldsymbol{w}_j$.

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

- Different decisions on the items $j + 1, \ldots, n$ can leave the same vector of residual capacities, and therefore the same value $v(j, \boldsymbol{k})$ may be needed several times.

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

- The value $v(j, \boldsymbol{k})$ depends only on values of row $j - 1$. Therefore, the **bottom-up approach** computes the rows $j = 0, 1, \ldots, n$ in this order, and in each row all the vectors $\boldsymbol{k}$ in any order.
- The **state** of the DP algorithm is the pair $(j, \boldsymbol{k})$: for every j the table has one cell for each vector $\boldsymbol{k}$, that is, $\prod_{d=1}^{m}(c_d + 1)$ cells. With $m = 2$ the row j is a matrix, with one row for each k_1 and one column for each k_2.

- Step 2: computation of an optimal solution
  - For each item j and vector $\boldsymbol{k}$ we store $s[j][\boldsymbol{k}] = 1$ if item j is selected in the optimal solution found for $(j, \boldsymbol{k})$, and $s[j][\boldsymbol{k}] = 0$ otherwise.
  - The set of all the vectors $\boldsymbol{k}$ is $\mathscr{C} = \{0, 1, \ldots, c_1\} \times \{0, 1, \ldots, c_2\} \times \cdots \times \{0, 1, \ldots, c_m\}$.

This is the pseudo-code of the **DP algorithm using the bottom-up approach**, which stores the tables s together with the tables v:

Algorithm

Algorithm: MDKP_DP

Input: An MDKP instance given by n, m, the array $p[1..n]$, the weight vectors $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n$, and the capacity vector $\boldsymbol{c}$

Output: The optimal value of the MDKP instance and the tables $s[j][\boldsymbol{k}]$

```
1  Function MDKP_DP(n, m, p, w, c)
2      v := new array[0..n][0..c_1] ··· [0..c_m]; s := new array[0..n][0..c_1] ··· [0..c_m]
3      foreach k ∈ 𝒞 do
4          v[0][k] ← 0; s[0][k] ← 0
5      end
6      for j := 1 to n do
7          foreach k ∈ 𝒞 do
8              v[j][k] ← v[j − 1][k]; s[j][k] ← 0
9              if w_j ≤ k and p[j] + v[j − 1][k − w_j] > v[j][k] then
10                 v[j][k] ← p[j] + v[j − 1][k − w_j]; s[j][k] ← 1
11             end
12         end
13     end
14     return v[n][c], s
```

- The structure is the one of the DP algorithm of the KP; the loop over the capacities becomes a loop over all the vectors $\boldsymbol{k} \in \mathscr{C}$.
- The test $\boldsymbol{w}_j \leq \boldsymbol{k}$ compares m pairs of numbers, and the vector $\boldsymbol{k} - \boldsymbol{w}_j$ needs m subtractions.
- In case of ties item j is not selected, because the value is updated only when it is strictly larger.

Example 7.2: Execution of the DP algorithm

We consider the MDKP instance of Example 7.1, with capacities $c_1 = 5$ and $c_2 = 4$. The row $j = 0$ is $v[0][\boldsymbol{k}] = 0$ for every $\boldsymbol{k}$. For each item j we compute the matrix $v[j][k_1][k_2]$, with one row for each $k_1 \in \{0, 1, \ldots, 5\}$ and one column for each $k_2 \in \{0, 1, \ldots, 4\}$.

These are the computations of the cell $\boldsymbol{k} = \boldsymbol{c} = (5, 4)$ for each item, and of the cells used by the reconstruction:

- $v[1][(5,4)] = \max\{v[0][(5,4)],\ p_1 + v[0][(4,2)]\} = \max\{0,\ 9+0\} = 9$
- $v[1][(1,3)] = \max\{v[0][(1,3)],\ p_1 + v[0][(0,1)]\} = \max\{0,\ 9+0\} = 9$
- $v[2][(5,4)] = \max\{v[1][(5,4)],\ p_2 + v[1][(1,3)]\} = \max\{9,\ 8+9\} = 17$
- $v[3][(5,4)] = \max\{v[2][(5,4)],\ p_3 + v[2][(3,3)]\} = \max\{17,\ 3+9\} = 17$

– $v[4][(5,4)] = \max\{v[3][(5,4)],\ p_4 + v[3][(2,0)]\} = \max\{17,\ 9+0\} = 17$

Item $j = 1$, with $p_1 = 9$ and $\boldsymbol{w}_1 = (1,2)$:

$v[1][k_1][k_2]$	$k_2=0$	$k_2=1$	$k_2=2$	$k_2=3$	$k_2=4$
$k_1=0$	0	0	0	0	0
$k_1=1$	0	0	9	9	9
$k_1=2$	0	0	9	9	9
$k_1=3$	0	0	9	9	9
$k_1=4$	0	0	9	9	9
$k_1=5$	0	0	9	9	9

Item $j = 2$, with $p_2 = 8$ and $\boldsymbol{w}_2 = (4,1)$:

$v[2][k_1][k_2]$	$k_2=0$	$k_2=1$	$k_2=2$	$k_2=3$	$k_2=4$
$k_1=0$	0	0	0	0	0
$k_1=1$	0	0	9	9	9
$k_1=2$	0	0	9	9	9
$k_1=3$	0	0	9	9	9
$k_1=4$	0	8	9	9	9
$k_1=5$	0	8	9	17	17

Item $j = 3$, with $p_3 = 3$ and $\boldsymbol{w}_3 = (2,1)$:

$v[3][k_1][k_2]$	$k_2=0$	$k_2=1$	$k_2=2$	$k_2=3$	$k_2=4$
$k_1=0$	0	0	0	0	0
$k_1=1$	0	0	9	9	9
$k_1=2$	0	3	9	9	9
$k_1=3$	0	3	9	12	12
$k_1=4$	0	8	9	12	12
$k_1=5$	0	8	9	17	17

Item $j = 4$, with $p_4 = 9$ and $\boldsymbol{w}_4 = (3,4)$:

$v[4][k_1][k_2]$	$k_2=0$	$k_2=1$	$k_2=2$	$k_2=3$	$k_2=4$
$k_1=0$	0	0	0	0	0
$k_1=1$	0	0	9	9	9
$k_1=2$	0	3	9	9	9
$k_1=3$	0	3	9	12	12
$k_1=4$	0	8	9	12	12
$k_1=5$	0	8	9	17	17

Example 7.3: The tables s

For the instance of Example 7.1, these are the matrices $s[j][k_1][k_2]$ computed by the DP algorithm; a 1 means that item j is selected in the optimal solution found for (j, k_1, k_2).

Item $j = 1$, with $p_1 = 9$ and $\boldsymbol{w}_1 = (1, 2)$:

$s[1][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	0	0
$k_1 = 1$	0	0	1	1	1
$k_1 = 2$	0	0	1	1	1
$k_1 = 3$	0	0	1	1	1
$k_1 = 4$	0	0	1	1	1
$k_1 = 5$	0	0	1	1	1

Item $j = 2$, with $p_2 = 8$ and $\boldsymbol{w}_2 = (4, 1)$:

$s[2][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	0	0
$k_1 = 1$	0	0	0	0	0
$k_1 = 2$	0	0	0	0	0
$k_1 = 3$	0	0	0	0	0
$k_1 = 4$	0	1	0	0	0
$k_1 = 5$	0	1	0	1	1

Item $j = 3$, with $p_3 = 3$ and $\boldsymbol{w}_3 = (2, 1)$:

$s[3][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	0	0
$k_1 = 1$	0	0	0	0	0
$k_1 = 2$	0	1	0	0	0
$k_1 = 3$	0	1	0	1	1
$k_1 = 4$	0	0	0	1	1
$k_1 = 5$	0	0	0	0	0

Item $j = 4$, with $p_4 = 9$ and $\boldsymbol{w}_4 = (3, 4)$:

$s[4][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	0	0
$k_1 = 1$	0	0	0	0	0
$k_1 = 2$	0	0	0	0	0
$k_1 = 3$	0	0	0	0	0
$k_1 = 4$	0	0	0	0	0
$k_1 = 5$	0	0	0	0	0

– Once the tables s have been computed, an optimal solution is reconstructed by scanning the items from $j = n$ down to $j = 1$, starting from $\boldsymbol{k} = \boldsymbol{c}$ and $x_j = 0$ for every item j:

$$x_j \leftarrow s[j][\boldsymbol{k}], \qquad \boldsymbol{k} \leftarrow \boldsymbol{k} - s[j][\boldsymbol{k}]\, \boldsymbol{w}_j.$$

Algorithm

Algorithm: MDKP_Reconstruct

Input: The number n of items, the capacity vector $\boldsymbol{c}$, the weight vectors $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_n$, and the tables s

Output: An optimal solution of the MDKP instance

```
1 Function MDKP_Reconstruct(n, c, w, s)
2     x := new array[1..n]
3     k := c
4     for j := n downto 1 do
5         x[j] ← s[j][k]
6         k ← k − x[j] w_j
7     end
8     return x
```

Example 7.4: Computing an optimal solution

After the execution of the DP algorithm on the instance of Example 7.1, we start from $j = 4$ and $\boldsymbol{k} = (5, 4)$:

Iteration	j	$\boldsymbol{k}$	$s[j][\boldsymbol{k}]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	Action
–	–	–	–	0	0	0	0	$\boldsymbol{k} = (5, 4)$
1	4	$(5, 4)$	0	0	0	0	0	$x[4] \leftarrow 0$, $\boldsymbol{k} = (5, 4)$
2	3	$(5, 4)$	0	0	0	0	0	$x[3] \leftarrow 0$, $\boldsymbol{k} = (5, 4)$
3	2	$(5, 4)$	1	0	1	0	0	$x[2] \leftarrow 1$, $\boldsymbol{k} \leftarrow (5, 4) - (4, 1) = (1, 3)$
4	1	$(1, 3)$	1	1	1	0	0	$x[1] \leftarrow 1$, $\boldsymbol{k} \leftarrow (1, 3) - (1, 2) = (0, 1)$

The vector $x = (1, 1, 0, 0)$ is an optimal solution of the MDKP instance of Example 7.1. Its weights are $1 + 4 = 5 \leq c_1$ and $2 + 1 = 3 \leq c_2$, and its optimal value is $v(4, 5, 4) = 17$.

With the first dimension alone, the KP with capacity $c_1 = 5$ has optimal value 18, attained by the items $\{1, 4\}$; their total weight in the second dimension is $2 + 4 = 6 > c_2$. Both dimensions must be checked at the same time, as the formula (7.1) does.

7.1.1 Number of steps

- Let $N = |\mathscr{C}| = \prod_{d=1}^{m}(c_d + 1)$ be the number of vectors $\boldsymbol{k}$, and let $S(n, m, \boldsymbol{c})$ denote the number of steps executed by `MDKP_DP` in the worst case. Lines 9 and 10 work on vectors of m components, so we count m steps for each of their executions:
  - Lines 2 and 14 are executed 1 time each.
  - Lines 3 and 4 are executed N times each.
  - Line 6 is executed n times.
  - Lines 7 and 8 are executed nN times each.
  - Lines 9 and 10 are executed at most nN times each, with m steps each.
- Therefore, the total number of steps in the worst case is

$$S(n, m, \boldsymbol{c}) = 2m\,nN + 2nN + 2N + n + 2,$$

  and, since $2m\,nN \le S(n, m, \boldsymbol{c}) \le 9m\,nN$,

$$S(n, m, \boldsymbol{c}) = \Theta\Big(m\,n \prod_{d=1}^{m}(c_d + 1)\Big).$$

  The DP algorithm for the MDKP executes $\Theta(m\,n\prod_{d=1}^{m}(c_d+1))$ steps in the worst case. For a fixed number m of dimensions this is $\Theta(n\prod_{d=1}^{m} c_d)$, a pseudo-polynomial number of steps.
- The reconstruction executes $\Theta(m\,n)$ steps.

7.1.2 The curse of dimensionality

- Each new dimension multiplies the number of states by $c_d + 1$. If all the capacities are equal to c, the table has $(n+1)(c+1)^m$ cells: the number of states grows **exponentially** with m.
- The size of the input grows only linearly with m: one more dimension adds $n+1$ numbers, the weights w_{dj} and the capacity c_d.
- This explosion of the number of states when the state becomes a vector is called the **curse of dimensionality**.

m	cells $(n+1)(c+1)^m$, with $n = 50$ and $c = 100$
1	5 151
2	520 251
3	52 545 351
4	5 307 080 451
5	536 015 125 551

- With $m = 1$ the table has a few thousand cells; with $m = 5$ it has more than $5 \cdot 10^{11}$, too many to be stored. The DP algorithm is therefore useful for the MDKP only when m and the capacities are small.

7.2 Typical mistakes

Warning

- **Checking only one dimension.** Item j fits in $\boldsymbol{k}$ only if $w_{dj} \leq k_d$ for *every* dimension d; one dimension is not enough.
- **Updating only one capacity.** When item j is selected, all the components of $\boldsymbol{k}$ decrease, by $\boldsymbol{w}_j$.
- **Solving one KP per dimension.** The KP with the capacity c_d alone gives an upper bound, but its optimal solution may violate another dimension.
- **Calling the DP algorithm polynomial.** Its number of steps contains the product of the capacities, which is exponential in m and in the size of the capacities.

7.3 In summary

Summary

- The recurrence of the KP extends to the MDKP by replacing the capacity k with the vector $\boldsymbol{k} = (k_1, \ldots, k_m)$: $v(j, \boldsymbol{k}) = \max\{v(j-1, \boldsymbol{k}),\ p_j + v(j-1, \boldsymbol{k} - \boldsymbol{w}_j)\}$ when $\boldsymbol{w}_j \leq \boldsymbol{k}$.
- The table has $(n+1)\prod_{d=1}^{m}(c_d+1)$ cells and the DP algorithm executes $\Theta(m\,n\prod_{d=1}^{m}(c_d+1))$ steps, that is, $\Theta(n\prod_{d=1}^{m} c_d)$ for a fixed m.
- The number of states grows exponentially with m: the curse of dimensionality.
- The tables s give an optimal solution, read backwards from $(n, \boldsymbol{c})$.

Dashboard of the MDKP example. Example 7.1 ($\boldsymbol{c} = (5, 4)$, four items):

Method	Lower bound	Upper bound	Origin
DP algorithm	17	17	optimal value, solution $x = (1, 1, 0, 0)$

Chapter 8

DP for the knapsack problem with setups

Prerequisites

- From this module, the chapter «DP for the 0/1 knapsack problem»: the recurrence $v(j, k)$ of the 0/1 knapsack problem (KP) over items and capacities, the matrix s of the decisions and the reconstruction of a solution.
- From the module *Combinatorial Optimization Problems*, the chapter «Knapsack problems»: the knapsack problem with setups, its classes, setup costs f_h and setup weights s_h.
- From the module *Algorithms*: pseudocode conventions (Chapter 2) and the size of the input (Chapter 11).

Objectives

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

- write the recurrence of the KPS: for each class, the best value without opening it and the best value after paying its setup;
- run the DP algorithm by hand, class by class, with a KP recurrence inside each opened class;
- reconstruct an optimal solution from the choice tables s and o;
- count the $\Theta(nc)$ steps of the DP algorithm.

The problem of this chapter is the knapsack problem with setups (KPS): its definition and its assumptions are on its page.

8.1 A DP algorithm

- We number the items class by class, so that $\mathscr{J}_h = \{\ell_{h-1}+1, \ldots, \ell_h\}$ for every class h, with $\ell_0 = 0$ and $\ell_q = n$. Class h has $t_h = |\mathscr{J}_h| = \ell_h - \ell_{h-1}$ items, and its i-th item is $j = \ell_{h-1} + i$.

- A solution is described by the binary vector $(x_1, x_2, \ldots, x_n)$. Class h is **open** if at least one of its items is selected: then its setup weight s_h is added to the total weight, and its setup cost f_h is subtracted from the total revenue.

Example 8.1: KPS instance

We consider a knapsack of capacity $c = 10$ and a set $\mathscr{I} = \{1, 2, \ldots, 5\}$ of 5 items ($n = 5$), partitioned into $q = 2$ classes: $\mathscr{J}_1 = \{1, 2\}$, with setup cost $f_1 = 2$ and setup weight $s_1 = 2$, and $\mathscr{J}_2 = \{3, 4, 5\}$, with setup cost $f_2 = 3$ and setup weight $s_2 = 1$. The revenues and weights are:

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$
p_j	6	5	4	7	3
w_j	3	2	2	4	1

We refer to this KPS instance as Example 8.1.

- Step 1: computation of the optimal value
  - The classes are the stages of the DP algorithm. For each class h there are two options: class h stays closed, or class h is opened, its setup is paid, and some of its items are selected.
  - Inside an open class, the choice of the items is a KP: the DP algorithm of the KP runs over the items of the class, starting from the value left after paying the setup.

For any $h \in \{0, 1, \ldots, q\}$ and $k \in \{0, 1, \ldots, c\}$, let $v(h, k)$ be the optimal value of the KPS subinstance with capacity k and the classes $\mathscr{J}_1, \ldots, \mathscr{J}_h$. For any $h \in \{1, \ldots, q\}$ and $i \in \{0, 1, \ldots, t_h\}$, let $u(h, i, k)$ be the optimal value of the same subinstance when class h is open (its setup is paid) and only its first i items are available, with $u(h, i, k) = -\infty$ if the setup weight does not fit.

The recursive formulas to compute $u(h, i, k)$ and $v(h, k)$ read as follows, where $j = \ell_{h-1} + i$ is the i-th item of class h:

$$u(h,i,k) = \begin{cases} -\infty & \text{if } i = 0 \text{ and } k < s_h, \\ v(h-1, k-s_h) - f_h & \text{if } i = 0 \text{ and } k \geq s_h, \\ u(h, i-1, k) & \text{if } i \geq 1 \text{ and } w_j > k, \\ \max\Big\{u(h,i-1,k), & \text{if } i \geq 1 \text{ and } w_j \leq k, \\ \quad p_j + u(h, i-1, k-w_j)\Big\} & \end{cases} \tag{8.1}$$

$$v(h,k) = \begin{cases} 0 & \text{if } h = 0, \\ \max\big\{v(h-1,k),\ u(h,t_h,k)\big\} & \text{if } h \geq 1. \end{cases} \tag{8.2}$$

- The row $u(h, 0, \cdot)$ **pays the setup** of class h: the setup weight s_h leaves the capacity $k - s_h$ to the classes $1, 2, \ldots, h-1$, and the setup cost f_h is subtracted.
- The rows $u(h, 1, \cdot), \ldots, u(h, t_h, \cdot)$ are the recurrence of the KP over the items of class h, with $-\infty + p_j = -\infty$.
- The formula (8.2) compares the two options for class h: closed, with value $v(h-1, k)$, or open, with value $u(h, t_h, k)$.
- The row $u(h, 0, \cdot)$ allows an open class with no selected item, which is not a solution of the KPS. This value is never chosen in (8.2), since $f_h \geq 1$ gives

  $$v(h-1, k-s_h) - f_h \ < \ v(h-1, k).$$

  Hence, when the maximum in (8.2) is attained only by $u(h, t_h, k)$, at least one item of class h is selected.
- An optimal solution for (h, k) is built from an optimal solution of a smaller subinstance: $(h-1, k)$ if class h is closed, and, inside an open class, $(h, i-1, \cdot)$ with or without item j.

  $\rightsquigarrow$ **Optimal Substructure verified!**
- Different decisions on the classes $h+1, \ldots, q$ can leave the same capacity k, and therefore the same value $v(h, k)$ may be needed several times.

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

  - The **bottom-up approach** computes, for $h = 1, 2, \ldots, q$, first the rows $u(h, i, \cdot)$ for $i = 0, 1, \ldots, t_h$, and then the row $v(h, \cdot)$, each one for increasing k. The optimal value of the KPS instance is $v(q, c)$.

- Step 2: computation of an optimal solution
  - Two choice tables are stored: $s[h][i][k] = 1$ if the i-th item of class h is selected in the optimal value $u[h][i][k]$, and $o[h][k] = 1$ if class h is open in the optimal value $v[h][k]$.
  - The setup weight s_h has one index; the choice $s[h][i][k]$ has three.

  This is the pseudo-code of the **DP algorithm using the bottom-up approach**, which stores the choice tables together with the values:

Algorithm

Algorithm: KPS_DP

Input: A KPS instance given by q, the classes $\mathscr{J}_1, \ldots, \mathscr{J}_q$ numbered as above, the setup costs f_h and weights s_h, arrays $p[1..n]$, $w[1..n]$, and capacity c

Output: The optimal value of the KPS instance and the choice tables s and o

Function *KPS_DP*$(q, \mathscr{J}, p, w, c)$
  $v, o :=$ new arrays$[0..q][0..c]$; $u, s :=$ new arrays$[1..q][0..t_h][0..c]$
  for $k := 0$ **to** c **do**
    $v[0][k] \leftarrow 0$; $o[0][k] \leftarrow 0$
  end
  for $h := 1$ **to** q **do**
    for $k := 0$ **to** c **do**
      $u[h][0][k] \leftarrow -\infty$
      if $k \geq s_h$ **then**
        $u[h][0][k] \leftarrow v[h-1][k-s_h] - f_h$
      end
    end
    for $i := 1$ **to** t_h **do**
      $j := \ell_{h-1} + i$
      for $k := 0$ **to** c **do**
        $u[h][i][k] \leftarrow u[h][i-1][k]$; $s[h][i][k] \leftarrow 0$
        if $w[j] \leq k$ and $p[j] + u[h][i-1][k-w[j]] > u[h][i][k]$ **then**
          $u[h][i][k] \leftarrow p[j] + u[h][i-1][k-w[j]]$; $s[h][i][k] \leftarrow 1$
        end
      end
    end
    for $k := 0$ **to** c **do**
      $v[h][k] \leftarrow v[h-1][k]$; $o[h][k] \leftarrow 0$
      if $u[h][t_h][k] > v[h][k]$ **then**
        $v[h][k] \leftarrow u[h][t_h][k]$; $o[h][k] \leftarrow 1$
      end
    end
  end
  return $v[q][c], s, o$

– For each class, the first loop over k pays the setup where it fits, the loops over i and k run the DP algorithm of the KP over the items of the class, and the last loop over k compares the closed and the open class.

– In case of ties the item is not selected and the class stays closed, because the values are updated only when they are strictly larger.

Example 8.2: Execution of the DP algorithm

We consider the KPS instance of Example 8.1. The rows are computed in the order of the table below, each one for $k = 0, 1, \ldots, 10$; these are some of the computations:

- $u[1][0][1] = -\infty$, since $k = 1 < s_1 = 2$
- $v[1][7] = \max\{v[0][7],\ u[1][2][7]\} = \max\{0,\ 9\} = 9$
- $v[2][10] = \max\{v[1][10],\ u[2][3][10]\} = \max\{9,\ 11\} = 11$
- $u[2][3][10] = \max\{u[2][2][10],\ p_5 + u[2][2][9]\} = \max\{10,\ 3 + 8\} = 11$
- $u[2][2][9] = \max\{u[2][1][9],\ p_4 + u[2][1][5]\} = \max\{6,\ 7 + 1\} = 8$
- $u[2][1][5] = \max\{u[2][0][5],\ p_3 + u[2][0][3]\} = \max\{0,\ 4 - 3\} = 1$
- $u[2][0][3] = v[1][2] - f_2 = 0 - 3 = -3$
- $v[1][2] = \max\{v[0][2],\ u[1][2][2]\} = \max\{0,\ -2\} = 0$

This is the table of the values after the execution of the DP algorithm, row by row in the order of computation:

Row	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$
$v[0][k]$	0	0	0	0	0	0	0	0	0	0	0
$u[1][0][k]$ (setup)	$-\infty$	$-\infty$	-2	-2	-2	-2	-2	-2	-2	-2	-2
$u[1][1][k]$ $(j=1)$	$-\infty$	$-\infty$	-2	-2	-2	4	4	4	4	4	4
$u[1][2][k]$ $(j=2)$	$-\infty$	$-\infty$	-2	-2	3	4	4	9	9	9	9
$v[1][k]$	0	0	0	0	3	4	4	9	9	9	9
$u[2][0][k]$ (setup)	$-\infty$	-3	-3	-3	-3	0	1	1	6	6	6
$u[2][1][k]$ $(j=3)$	$-\infty$	-3	-3	1	1	1	1	4	6	6	10
$u[2][2][k]$ $(j=4)$	$-\infty$	-3	-3	1	1	4	4	8	8	8	10
$u[2][3][k]$ $(j=5)$	$-\infty$	-3	0	1	4	4	7	8	11	11	11
$v[2][k]$	0	0	0	1	4	4	7	9	11	11	11

The optimal value of the instance is $v[2][10] = 11$. The values -2 and -3 in the setup rows are the setup costs paid before any item is selected.

Example 8.3: The choice tables

For the instance of Example 8.1, these are the choice tables computed by the DP algorithm: the rows $s[h][i][\cdot]$ of the items, and the rows $o[h][\cdot]$ of the classes.

Row	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8$	$k=9$	$k=10$
$s[1][1][k]$ $(j=1)$	0	0	0	0	0	1	1	1	1	1	1
$s[1][2][k]$ $(j=2)$	0	0	0	0	1	0	0	1	1	1	1
$o[1][k]$	0	0	0	0	1	1	1	1	1	1	1
$s[2][1][k]$ $(j=3)$	0	0	0	1	1	1	0	1	0	0	1
$s[2][2][k]$ $(j=4)$	0	0	0	0	0	1	1	1	1	1	0
$s[2][3][k]$ $(j=5)$	0	0	1	0	1	0	1	0	1	1	1
$o[2][k]$	0	0	0	1	1	0	1	0	1	1	1

– An optimal solution is reconstructed by scanning the classes from $h = q$ down to $h = 1$, starting from $k = c$ and $x_j = 0$ for every item j.

– If $o[h][k] = 0$, class h is closed and k does not change. If $o[h][k] = 1$, the items of class h are scanned from $i = t_h$ down to $i = 1$ as in the KP, and at the end the setup weight s_h is subtracted from k.

Algorithm

Algorithm: KPS_Reconstruct

Input: The number q of classes, the setup weights s_h, the array $w[1..n]$, the capacity c, and the choice tables s and o

Output: An optimal solution of the KPS instance

```
1  Function KPS_Reconstruct(q, w, c, s, o)
2      x := new array[1..n], with x[j] = 0 for every j
3      k := c
4      for h := q downto 1 do
5          if o[h][k] = 1 then
6              for i := t_h downto 1 do
7                  j := ℓ_{h-1} + i; x[j] ← s[h][i][k]
8                  k ← k − x[j] w[j]
9              end
10             k ← k − s_h
11         end
12     end
13     return x
```

Example 8.4: Computing an optimal solution

After the execution of the DP algorithm on the instance of Example 8.1, we start from $h = 2$ and $k = 10$:

Iteration	Stage	k	Choice	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$x[5]$	Action
–	–	–	–	0	0	0	0	0	$k = 10$
1	$h = 2$	10	$o[2][10] = 1$	0	0	0	0	0	class 2 open
2	$j = 5$	10	$s[2][3][10] = 1$	0	0	0	0	1	$k \leftarrow 10 - 1 = 9$
3	$j = 4$	9	$s[2][2][9] = 1$	0	0	0	1	1	$k \leftarrow 9 - 4 = 5$
4	$j = 3$	5	$s[2][1][5] = 1$	0	0	1	1	1	$k \leftarrow 5 - 2 = 3$
–	–	3	–	0	0	1	1	1	setup: $k \leftarrow 3 - s_2 = 2$
5	$h = 1$	2	$o[1][2] = 0$	0	0	1	1	1	class 1 closed

The vector $x = (0, 0, 1, 1, 1)$ is an optimal solution of the KPS instance of Example 8.1: only class 2 is open, and the items $\{3, 4, 5\}$ are selected. Its total weight is $2 + 4 + 1 + s_2 = 8 \leq c$ and its optimal value is $v(2, 10) = 4 + 7 + 3 - f_2 = 11$.

The items of class 1 have the larger ratio between total revenue and total weight (11/5 against 14/7), but class 1 stays closed. With class 1 alone the best value is $v[1][10] = 9$, while the open class 2 gives $u[2][3][10] = 11$; after its items and its setup weight only $k = 2$ is left for class 1, and $o[1][2] = 0$.

8.1.1 Number of steps

- Let $S(n, q, c)$ denote the number of steps executed by `KPS_DP` in the worst case. The loops over i run over the items of each class, so over all the classes they run $t_1 + t_2 + \cdots + t_q = n$ times. Assuming that the conditions at lines 9, 17 and 24 are always true:
  - Lines 2 and 29 are executed 1 time each.
  - Lines 3 and 4 are executed $c + 1$ times each.
  - Line 6 is executed q times.
  - Lines 7, 8, 9, 10, 22, 23, 24 and 25 are executed $q(c + 1)$ times each.
  - Lines 13 and 14 are executed n times each.
  - Lines 15, 16, 17 and 18 are executed $n(c + 1)$ times each.
- Therefore, the total number of steps in the worst case is

$$S(n, q, c) = 4n(c + 1) + 8q(c + 1) + 2n + q + 2c + 4.$$

  Since every class contains at least one item, $1 \leq q \leq n$, and

$$4n(c + 1) \;\leq\; S(n, q, c) \;\leq\; 12n(c + 1) + 3n + 2c + 4.$$

  Both bounds are $\Theta(nc)$, hence $S(n, q, c) = \Theta(nc)$.

  The DP algorithm for the KPS executes $S(n, q, c) = \Theta(nc)$ steps in the worst case: the setups add $2q$ rows to the n rows of the KP recurrence.

- The reconstruction executes $\Theta(n)$ steps.

8.2 Typical mistakes

Warning

- **Paying the setup for every item.** The setup of class h is paid once, in the row $u(h, 0, \cdot)$; the rows of the items add only p_j and w_j.
- **Starting the items of a class from $v(h-1,k)$.** The rows of an open class start from $v(h-1, k-s_h)-f_h$: starting from $v(h-1,k)$ would select items without paying the setup.
- **Forgetting the setup weight in the reconstruction.** After the items of an open class, the capacity decreases also by s_h.
- **Confusing s_h with $s[h][i][k]$.** The setup weight of class h has one index; the choice of the i-th item of class h has three.

8.3 In summary

Summary

- For each class h the DP algorithm compares the closed class, $v(h-1,k)$, with the open class, $u(h, t_h, k)$.
- The open class starts from $u(h, 0, k) = v(h-1, k-s_h) - f_h$, which pays the setup once, and continues with the KP recurrence over its items.
- The choice tables s (items) and o (classes) give an optimal solution, read backwards from (q, c).
- The DP algorithm executes $\Theta(nc)$ steps, as the DP algorithm of the KP.

Dashboard of the KPS example. Example 8.1 ($c = 10$, two classes):

Method	Lower bound	Upper bound	Origin
DP algorithm	11	11	optimal value, solution $x = (0, 0, 1, 1, 1)$

Chapter 9

DP for the multiple knapsack problem

Prerequisites

- From this module, the chapter «DP for the 0/1 knapsack problem»: the recurrence $v(j, k)$ of the 0/1 knapsack problem (KP) over items and capacities, the matrix s of the decisions and the reconstruction of a solution.
- From the module *Combinatorial Optimization Problems*, the chapter «Knapsack problems»: the multiple knapsack problem, with m knapsacks of capacities $c_1, \ldots, c_m$.
- From the module *Algorithms*: pseudocode conventions (Chapter 2) and the size of the input (Chapter 11).

Objectives

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

- write the recurrence $v(j, k_1, \ldots, k_m)$ over the items and the residual capacities of the m knapsacks;
- run the DP algorithm by hand on an instance with $m = 2$ knapsacks, one matrix for each item;
- reconstruct an optimal solution, with the knapsack of every selected item, from the tables s;
- count the $\Theta(m\, n \prod_{i=1}^{m}(c_i + 1))$ steps and explain why the number of states grows exponentially with m.

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

9.1 A DP algorithm

- We write $i \in \mathscr{K} = \{1, 2, \ldots, m\}$ for a generic knapsack, because in the DP chapters the letter k denotes a capacity. Knapsack i has capacity c_i.

- A solution assigns each item j to one knapsack or leaves it out: it is described by the vector $(a_1, a_2, \ldots, a_n)$, where $a_j = i$ if item j is inserted into knapsack i, and $a_j = 0$ if item j is not selected.

- The vector $\boldsymbol{e}_i$ is the i-th unit vector of length m: $\boldsymbol{k} - w_j\boldsymbol{e}_i$ is the vector $\boldsymbol{k}$ with the component k_i decreased by w_j.

Example 9.1: MKP instance

We consider $m = 2$ knapsacks with capacities $c_1 = 5$ and $c_2 = 4$, and a set $\mathscr{I} = \{1, 2, 3, 4\}$ of 4 items ($n = 4$), with the following profits and weights:

	$j=1$	$j=2$	$j=3$	$j=4$
p_j	3	9	7	5
w_j	3	4	4	2

We refer to this MKP instance as Example 9.1.

- Step 1: computation of the optimal value

For any $j \in \{0, 1, \ldots, n\}$ and any vector $\boldsymbol{k} = (k_1, k_2, \ldots, k_m)$ with $k_i \in \{0, 1, \ldots, c_i\}$ for every $i \in \mathscr{K}$, let $v(j, \boldsymbol{k}) = v(j, k_1, \ldots, k_m)$ be the optimal value of the MKP subinstance given by:

1. m knapsacks, knapsack i with capacity k_i,
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 recursive formula to compute $v(j, \boldsymbol{k})$ reads as follows:

$$v(j,\boldsymbol{k}) = \begin{cases} 0 & \text{if } j = 0, \\ v(j-1,\boldsymbol{k}) & \text{if } j \geq 1 \text{ and } w_j > k_i \text{ for all } i, \\ \max\left\{v(j-1,\boldsymbol{k}),\ p_j + \max\limits_{i:\, w_j \leq k_i} v(j-1,\boldsymbol{k}-w_j\boldsymbol{e}_i)\right\} & \text{if } j \geq 1 \text{ and } w_j \leq k_i \text{ for some } i. \end{cases} \tag{9.1}$$

- The components of $\boldsymbol{k}$ are the **residual capacities** of the knapsacks: in the top-down view, the decisions on the items $j+1, \ldots, n$ have already used $c_i - k_i$ units of knapsack i.
- With $m = 1$ the formula (9.1) is the recurrence of the KP.
- The base cases are $v(0, \boldsymbol{k}) = 0$ for every vector $\boldsymbol{k}$, since no item is available.
- The formula generates at most $m + 1$ **independent subproblems**: item j is left out, or it is inserted into one of the knapsacks i with $w_j \leq k_i$, and then only the capacity of knapsack i decreases.

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

- Different assignments of the items $j+1, \ldots, n$ can leave the same residual capacities, and therefore the same value $v(j, \boldsymbol{k})$ may be needed several times.

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

- The value $v(j, \boldsymbol{k})$ depends only on values of row $j - 1$. Therefore, the **bottom-up approach** computes the rows $j = 0, 1, \ldots, n$ in this order, and in each row all the vectors $\boldsymbol{k}$ in any order. With $m = 2$ the row j is a matrix, with one row for each k_1 and one column for each k_2.

- Step 2: computation of an optimal solution
  - For each item j and vector $\boldsymbol{k}$ we store $s[j][\boldsymbol{k}] = i$ if item j is inserted into knapsack i in the optimal solution found for $(j, \boldsymbol{k})$, and $s[j][\boldsymbol{k}] = 0$ if item j is left out.
  - The set of all the vectors $\boldsymbol{k}$ is $\mathscr{C} = \{0, 1, \ldots, c_1\} \times \{0, 1, \ldots, c_2\} \times \cdots \times \{0, 1, \ldots, c_m\}$.

This is the pseudo-code of the **DP algorithm using the bottom-up approach**, which stores the tables s together with the tables v:

Algorithm

Algorithm: MKP_DP

Input: An MKP instance given by n, m, arrays $p[1..n]$, $w[1..n]$, and the capacity vector $\boldsymbol{c}$

Output: The optimal value of the MKP instance and the tables $s[j][\boldsymbol{k}]$

Function *MKP_DP*$(n, m, p, w, \boldsymbol{c})$
  $v :=$ new array$[0..n][0..c_1]\cdots[0..c_m]$; $s :=$ new array$[0..n][0..c_1]\cdots[0..c_m]$
  foreach $\boldsymbol{k} \in \mathscr{C}$ **do**
    $v[0][\boldsymbol{k}] \leftarrow 0$; $s[0][\boldsymbol{k}] \leftarrow 0$
  end
  for $j := 1$ **to** n **do**
    foreach $\boldsymbol{k} \in \mathscr{C}$ **do**
      $v[j][\boldsymbol{k}] \leftarrow v[j-1][\boldsymbol{k}]$; $s[j][\boldsymbol{k}] \leftarrow 0$
      for $i := 1$ **to** m **do**
        if $w[j] \le k_i$ and $p[j] + v[j-1][\boldsymbol{k} - w[j]\,\boldsymbol{e}_i] > v[j][\boldsymbol{k}]$ **then**
          $v[j][\boldsymbol{k}] \leftarrow p[j] + v[j-1][\boldsymbol{k} - w[j]\,\boldsymbol{e}_i]$; $s[j][\boldsymbol{k}] \leftarrow i$
        end
      end
    end
  end
  return $v[n][\boldsymbol{c}], s$

- For each vector $\boldsymbol{k}$, the DP algorithm first leaves item j out, and then tries to insert it into each knapsack $i = 1, 2, \ldots, m$ in which it fits.
- In case of ties the first option met is kept: leaving item j out, then the knapsack of smallest index, because the value is updated only when it is strictly larger.

Example 9.2: Execution of the DP algorithm

We consider the MKP instance of Example 9.1, with capacities $c_1 = 5$ and $c_2 = 4$. The row $j = 0$ is $v[0][\boldsymbol{k}] = 0$ for every $\boldsymbol{k}$. For each item j we compute the matrix $v[j][k_1][k_2]$, with one row for each $k_1 \in \{0, 1, \ldots, 5\}$ and one column for each $k_2 \in \{0, 1, \ldots, 4\}$.

These are the computations of the cell $\boldsymbol{k} = \boldsymbol{c} = (5, 4)$ for each item, and of the cells used by the reconstruction:

- $v[1][(5,4)] = \max\{v[0][(5,4)],\ p_1 + v[0][(2,4)],\ p_1 + v[0][(5,1)]\} = \max\{0,\ 3 + 0,\ 3 + 0\} = 3$
- $v[1][(3,0)] = \max\{v[0][(3,0)],\ p_1 + v[0][(0,0)]\} = \max\{0,\ 3 + 0\} = 3$
- $v[2][(5,4)] = \max\{v[1][(5,4)],\ p_2 + v[1][(1,4)],\ p_2 + v[1][(5,0)]\} = \max\{3,\ 9 + 3,\ 9 + 3\} = 12$
- $v[2][(3,4)] = \max\{v[1][(3,4)],\ p_2 + v[1][(3,0)]\} = \max\{3,\ 9 + 3\} = 12$

- $v[3][(5,4)] = \max\{v[2][(5,4)],\ p_3 + v[2][(1,4)],\ p_3 + v[2][(5,0)]\} = \max\{12,\ 7+9,\ 7+9\} = 16$
- $v[3][(3,4)] = \max\{v[2][(3,4)],\ p_3 + v[2][(3,0)]\} = \max\{12,\ 7+3\} = 12$
- $v[4][(5,4)] = \max\{v[3][(5,4)],\ p_4 + v[3][(3,4)],\ p_4 + v[3][(5,2)]\} = \max\{16,\ 5+12,\ 5+9\} = 17$

Item $j = 1$, with $p_1 = 3$ and $w_1 = 3$:

$v[1][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	3	3
$k_1 = 1$	0	0	0	3	3
$k_1 = 2$	0	0	0	3	3
$k_1 = 3$	3	3	3	3	3
$k_1 = 4$	3	3	3	3	3
$k_1 = 5$	3	3	3	3	3

Item $j = 2$, with $p_2 = 9$ and $w_2 = 4$:

$v[2][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	3	9
$k_1 = 1$	0	0	0	3	9
$k_1 = 2$	0	0	0	3	9
$k_1 = 3$	3	3	3	3	12
$k_1 = 4$	9	9	9	12	12
$k_1 = 5$	9	9	9	12	12

Item $j = 3$, with $p_3 = 7$ and $w_3 = 4$:

$v[3][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	3	9
$k_1 = 1$	0	0	0	3	9
$k_1 = 2$	0	0	0	3	9
$k_1 = 3$	3	3	3	3	12
$k_1 = 4$	9	9	9	12	16
$k_1 = 5$	9	9	9	12	16

Item $j = 4$, with $p_4 = 5$ and $w_4 = 2$:

$v[4][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	5	5	9
$k_1 = 1$	0	0	5	5	9
$k_1 = 2$	5	5	5	8	14
$k_1 = 3$	5	5	8	8	14
$k_1 = 4$	9	9	14	14	16
$k_1 = 5$	9	9	14	14	17

Example 9.3: The tables s

For the instance of Example 9.1, these are the matrices $s[j][k_1][k_2]$ computed by the DP algorithm: 0 means that item j is left out, 1 and 2 the knapsack into which it is inserted.

Item $j = 1$, with $p_1 = 3$ and $w_1 = 3$:

$s[1][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	2	2
$k_1 = 1$	0	0	0	2	2
$k_1 = 2$	0	0	0	2	2
$k_1 = 3$	1	1	1	1	1
$k_1 = 4$	1	1	1	1	1
$k_1 = 5$	1	1	1	1	1

Item $j = 2$, with $p_2 = 9$ and $w_2 = 4$:

$s[2][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	0	2
$k_1 = 1$	0	0	0	0	2
$k_1 = 2$	0	0	0	0	2
$k_1 = 3$	0	0	0	0	2
$k_1 = 4$	1	1	1	1	1
$k_1 = 5$	1	1	1	1	1

Item $j = 3$, with $p_3 = 7$ and $w_3 = 4$:

$s[3][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	0	0
$k_1 = 1$	0	0	0	0	0
$k_1 = 2$	0	0	0	0	0
$k_1 = 3$	0	0	0	0	0
$k_1 = 4$	0	0	0	0	1
$k_1 = 5$	0	0	0	0	1

Item $j = 4$, with $p_4 = 5$ and $w_4 = 2$:

$s[4][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	2	2	0
$k_1 = 1$	0	0	2	2	0
$k_1 = 2$	1	1	1	1	1
$k_1 = 3$	1	1	2	1	1
$k_1 = 4$	0	0	2	2	0
$k_1 = 5$	0	0	2	2	1

– Once the tables s have been computed, an optimal solution is reconstructed by scanning the items from $j = n$ down to $j = 1$, starting from $\boldsymbol{k} = \boldsymbol{c}$ and $a_j = 0$ for every item j:

$$a_j \leftarrow s[j][\boldsymbol{k}], \qquad \text{and, if } a_j = i \geq 1, \quad \boldsymbol{k} \leftarrow \boldsymbol{k} - w_j \boldsymbol{e}_i.$$

Algorithm

Algorithm: MKP_Reconstruct

Input: The number n of items, the capacity vector $\boldsymbol{c}$, the array $w[1..n]$, and the tables s

Output: An optimal solution of the MKP instance, as the vector of the knapsacks

```
1  Function MKP_Reconstruct(n, c, w, s)
2      a := new array[1..n]
3      k := c
4      for j := n downto 1 do
5          a[j] ← s[j][k]
6          if a[j] ≥ 1 then
7              k ← k − w[j] e_{a[j]}
8          end
9      end
10     return a
```

Example 9.4: Computing an optimal solution

After the execution of the DP algorithm on the instance of Example 9.1, we start from $j = 4$ and $\boldsymbol{k} = (5, 4)$:

Iteration	j	$\boldsymbol{k}$	$s[j][\boldsymbol{k}]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	Action
–	–	–	–	0	0	0	0	$\boldsymbol{k} = (5, 4)$
1	4	$(5, 4)$	1	0	0	0	1	$a[4] \leftarrow 1$, $\boldsymbol{k} \leftarrow (3, 4)$
2	3	$(3, 4)$	0	0	0	0	1	$a[3] \leftarrow 0$, $\boldsymbol{k} = (3, 4)$
3	2	$(3, 4)$	2	0	2	0	1	$a[2] \leftarrow 2$, $\boldsymbol{k} \leftarrow (3, 0)$
4	1	$(3, 0)$	1	1	2	0	1	$a[1] \leftarrow 1$, $\boldsymbol{k} \leftarrow (0, 0)$

The vector $a = (1, 2, 0, 1)$ is an optimal solution of the MKP instance of Example 9.1: knapsack 1 contains the items $\{1, 4\}$, of total weight $3 + 2 = 5 \leq c_1$, and knapsack 2 contains the items $\{2\}$, of total weight $4 \leq c_2$. The optimal value is $v(4, 5, 4) = 17$.

Filling the knapsacks one after the other is not enough: the best filling of knapsack 1 alone is $\{2\}$, of profit 9, and the best filling of knapsack 2 with the remaining items is $\{3\}$, of profit 7, for a total of $16 < 17$.

9.1.1 Number of steps

- Let $N = |\mathscr{C}| = \prod_{i=1}^{m}(c_i + 1)$ be the number of vectors $\boldsymbol{k}$, and let $S(n, m, \boldsymbol{c})$ denote the number of steps executed by `MKP_DP` in the worst case. Inserting item j into knapsack i changes only the component k_i, so lines 10 and 11 count one step each:
  - Lines 2 and 16 are executed 1 time each.
  - Lines 3 and 4 are executed N times each.
  - Line 6 is executed n times.
  - Lines 7 and 8 are executed nN times each.
  - Lines 9, 10 and 11 are executed at most $m\,nN$ times each.
- Therefore, the total number of steps in the worst case is

$$S(n, m, \boldsymbol{c}) = 3m\,nN + 2nN + 2N + n + 2,$$

  and, since $3m\,nN \le S(n, m, \boldsymbol{c}) \le 10m\,nN$,

$$S(n, m, \boldsymbol{c}) = \Theta\Big(m\,n\prod_{i=1}^{m}(c_i + 1)\Big).$$

  The DP algorithm for the MKP executes $\Theta(m\,n\prod_{i=1}^{m}(c_i+1))$ steps in the worst case. For a fixed number m of knapsacks this is $\Theta(n\prod_{i=1}^{m} c_i)$, a pseudo-polynomial number of steps.
- The reconstruction executes $\Theta(n)$ steps.

9.1.2 The growth with the number of knapsacks

- Each new knapsack multiplies the number of states by $c_i + 1$. If all the capacities are equal to c, the table has $(n+1)(c+1)^m$ cells: the number of states grows **exponentially** with m, while the size of the input grows only by one number, the capacity c_i.
- With $n = 50$ items and capacities $c_i = 100$, the table has $51 \cdot 101 = 5151$ cells with $m = 1$, $51 \cdot 101^2 = 520\,251$ with $m = 2$, and $51 \cdot 101^5 \approx 5.4 \cdot 10^{11}$ with $m = 5$.
- Knapsacks with the same capacity give symmetric states: if $c_1 = c_2$, the vectors $(k_1, k_2, \dots)$ and $(k_2, k_1, \dots)$ have the same optimal value, and the DP algorithm computes both.

9.2 Typical mistakes

Warning

- **Decreasing all the capacities.** Item j goes into one knapsack only: $\boldsymbol{k} - w_j \boldsymbol{e}_i$ decreases the component k_i, not all of them as in the multidimensional knapsack problem.
- **Using one capacity $c_1 + \cdots + c_m$.** The KP with the total capacity gives only an upper bound: its solution may not be divisible among the knapsacks.
- **Filling the knapsacks one after the other.** An optimal filling of knapsack 1, followed by an optimal filling of knapsack 2 with the remaining items, can be worse than the optimum.
- **Forgetting the option «item left out».** The term $v(j-1, \boldsymbol{k})$ is always a candidate, also when item j fits into some knapsack.

9.3 In summary

Summary

- The state of the DP algorithm is the item j and the vector $\boldsymbol{k}$ of the residual capacities of the m knapsacks.
- Item j is left out or inserted into one knapsack i in which it fits: $m+1$ options, and only k_i decreases.
- The DP algorithm executes $\Theta(m\, n \prod_{i=1}^{m}(c_i+1))$ steps: the number of states grows exponentially with m.
- The tables s store the knapsack of each item, and an optimal solution is read backwards from $(n, \boldsymbol{c})$.

Dashboard of the MKP example. Example 9.1 ($\boldsymbol{c} = (5, 4)$, four items):

Method	Lower bound	Upper bound	Origin
DP algorithm	17	17	optimal value, solution $a = (1, 2, 0, 1)$

Lot sizing problems

Chapter 10

DP for the uncapacitated lot sizing problem

Prerequisites

- From the module *Combinatorial Optimization Problems*, the chapters «Lot sizing problems» and «Formulations of lot sizing problems»: the single-item problem, its reference instance, and its ILP model with the variables x_t, s_t, y_t.
- From this module, the chapter «Principles of dynamic programming» and the DP algorithms of the previous chapters: recurrence, base cases, lookup table and reconstruction of a solution.
- From the module *Algorithms*: pseudocode (Chapter 2), and the size of the input (Chapter 11).

Where we were

- **As a problem and as a model**: single-item lot sizing and its ILP formulations (chapters «Lot sizing problems» and «Formulations of lot sizing problems» of the module *Combinatorial Optimization Problems*).
- **Now, as a table**: the DP algorithm of Wagner and Whitin gives the optimal value of the uncapacitated problem in $\Theta(m^2)$ steps.

Objectives

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

- state the zero-inventory ordering property, and prove it with the exchange argument;
- compute the cost $c_{j,t}$ of a production run, and the matrix of the costs, in $O(m^2)$ steps;
- write the recurrence $v(t)$ over the last production run, run the DP algorithm by hand and reconstruct an optimal plan;
- read the recurrence as a shortest path in the network of the production runs;
- count the steps of the DP algorithm, and show that they are $\Theta(m^2)$.

The problem of this chapter is the uncapacitated lot sizing problem (ULS): its definition and

its assumptions are on its page.

10.1 Notation, model and feasible solutions

- For any $j, t \in T$, we denote by
$$D_{j,t} = \sum_{k=j}^{t} d_k$$
the total demand of periods $j, j+1, \ldots, t$, with $D_{j,t} = 0$ if $j > t$.

- The ILP model is the ILP_{ULS} model of the chapter «Formulations of lot sizing problems» of the module *Combinatorial Optimization Problems*, section «ILP models for the variants»: the ILP_{CLSP} model without the capacity constraints and with the big-M value $D_{t,m}$ of period t, the total demand of the remaining periods. Its variables are the production x_t, the inventory s_t at the end of period t, and the setup $y_t \in \{0, 1\}$, for every period $t \in T$.

We recall the ILP_{ULS} model of the uncapacitated lot sizing problem (ULS):

$$\min \quad \sum_{t \in T} (sc_t\, y_t + vc_t\, x_t + hc_t\, s_t) \tag{10.1a}$$

$$\text{subject to} \quad s_{t-1} + x_t = d_t + s_t, \qquad \forall t \in T, \tag{10.1b}$$

$$x_t \le D_{t,m}\, y_t, \qquad \forall t \in T, \tag{10.1c}$$

$$s_0 = s_m = 0, \tag{10.1d}$$

$$x_t \in \mathbb{Z}_{\ge 0}, \qquad \forall t \in T, \tag{10.1e}$$

$$s_t \in \mathbb{Z}_{\ge 0}, \qquad \forall t \in T, \tag{10.1f}$$

$$y_t \in \{0, 1\}, \qquad \forall t \in T. \tag{10.1g}$$

This ILP model is referred to as the ILP_{ULS} model.

- Description of the objective function and constraints of the ILP_{ULS} model:
  - the objective function (10.1a) minimizes the total cost of setups, production and holding;
  - constraints (10.1b) are the flow balancing constraints;
  - constraints (10.1c) force a setup in every period with positive production;
  - constraint (10.1d) imposes an empty inventory at the beginning and at the end of the horizon;
  - finally, constraints (10.1e)–(10.1g) define the variables of the ILP model.

Example 10.1: Uncapacitated lot sizing

We consider $m = 5$ periods with the following data:

t	1	2	3	4	5
d_t	20	10	30	40	10
sc_t	50	50	50	50	50
vc_t	0	0	0	0	0
hc_t	1	1	1	1	1

We refer to this ULS instance as Example 10.1. It is the reference instance of the chapter «Lot sizing problems» of the module *Combinatorial Optimization Problems*, counted in single items instead of units of ten and without the capacities (setup cost 50, holding cost 1, no variable production cost).

10.1.1 Encoding of feasible solutions

A *feasible solution* of the ULS is a production plan that satisfies the demand of every period, starting and ending with an empty inventory.

- A ULS feasible solution can be described by a solution vector

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

where each component has the following meaning:

$$x_t = \text{number of units produced in period } t, \qquad \forall t \in \{1, 2, \ldots, m\}.$$

The inventory at the end of period t is determined by the vector:

$$s_t = \sum_{j=1}^{t} x_j - D_{1,t}, \qquad \forall t \in \{1, 2, \ldots, m\}.$$

Since the solution is feasible, the inventories are non-negative and the last one is empty:

$$s_t \geq 0 \quad \forall t \in \{1, 2, \ldots, m-1\}, \qquad s_m = 0.$$

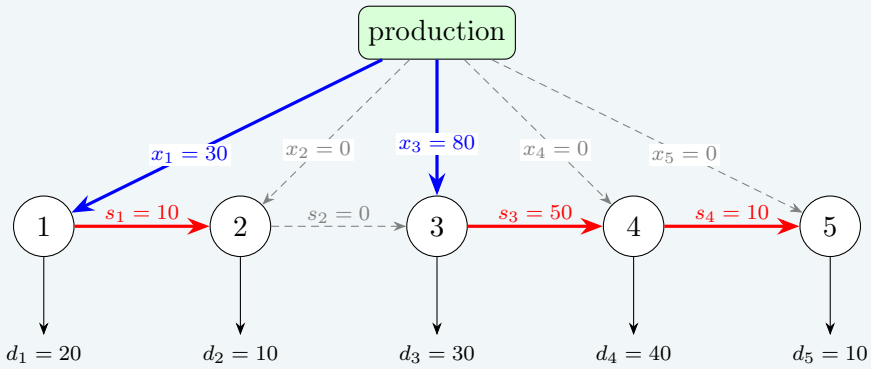
A setup is paid in period t if and only if $x_t > 0$. The total cost of a feasible solution is

$$\sum_{t=1,\ x_t>0}^{m} sc_t + \sum_{t=1}^{m} vc_t\, x_t + \sum_{t=1}^{m} hc_t\, s_t.$$

Example 10.2: A feasible solution

We consider the ULS instance of Example 10.1. The following network represents the feasible solution $(30, 0, 80, 0, 0)$: each period is a node, the production x_t enters node t, the demand d_t leaves it, and the inventory s_t flows from node t to node $t+1$. Thick arcs carry

a positive flow.



The setups are in periods 1, 3, the inventories are $(s_1, \dots, s_5) = (10, 0, 50, 10, 0)$, and the total cost is (there is no variable production cost)

$$\underbrace{50 + 50}_{\text{setup}} + \underbrace{1 \cdot 10 + 1 \cdot 0 + 1 \cdot 50 + 1 \cdot 10 + 1 \cdot 0}_{\text{holding}} = 100 + 70 = 170.$$

- A production quantity can take many values, and a ULS instance has a huge number of feasible solutions (infinitely many, if the quantities are not required to be integer). The next property shows that only very few of them need to be considered.

10.1.2 The zero-inventory ordering property

Theorem 10.1: Zero-inventory ordering (Wagner and Whitin, 1958)

Every ULS instance has an optimal solution in which no period with positive production starts with a positive inventory, that is,

$$s_{t-1}\, x_t = 0 \qquad \forall t \in \{1, 2, \dots, m\}.$$

- *Proof.* First, an optimal solution exists: the vector $(d_1, d_2, \dots, d_m)$ is feasible, and for each of the 2^m choices of the set of periods with a setup, the remaining problem is a linear program over a bounded polyhedron (since $s_m = 0$, every x_t is between 0 and $D_{1,m}$), so it has an optimal solution whenever it is feasible.

- Among all optimal solutions, consider one with the **smallest number of periods with positive production**, and suppose, by contradiction, that there exists a period t with $x_t > 0$ and $s_{t-1} > 0$.

- Since $s_0 = 0$ and $s_{t-1} > 0$, something is produced before period t: let $j < t$ be the last period before t with $x_j > 0$. No unit is produced in periods $j+1, \dots, t-1$, so the inventory can only decrease from period j to period $t-1$:

$$s_k \geq s_{t-1} > 0 \qquad \forall k \in \{j, j+1, \dots, t-1\}.$$

- For a number ε, move ε units of production from period j to period t: replace x_j by $x_j - \varepsilon$, x_t by $x_t + \varepsilon$, and s_k by $s_k - \varepsilon$ for all $k \in \{j, \dots, t-1\}$. All balancing constraints remain satisfied, and the new plan is feasible for every ε with
$$-x_t \leq \varepsilon \leq \min\{x_j,\ s_{t-1}\}.$$
- As long as x_j and x_t stay positive, the setups do not change and the cost changes by $\varepsilon\,\Delta$, where
$$\Delta = vc_t - vc_j - \sum_{k=j}^{t-1} hc_k$$
is the cost of producing one unit in period t minus the cost of producing it in period j and holding it until period t.
- If $\Delta < 0$, a small $\varepsilon > 0$ gives a feasible solution with a smaller cost, which contradicts optimality.
- If $\Delta \geq 0$, choose $\varepsilon = -x_t$: the whole production of period t is moved to period j, the cost changes by $-x_t\,\Delta \leq 0$, and the setup cost $sc_t \geq 0$ of period t is saved. The new solution is optimal and has one period with positive production less, which contradicts the choice of the solution.
- Hence no such period t exists. □

In an optimal solution with the zero-inventory ordering property, each period j with positive production produces exactly the demands of periods $j, j+1, \dots, t$, where $t+1$ is the next period with positive production (or $t = m$ if there is none):
$$x_j = D_{j,t}.$$

- Indeed, $s_{j-1} = 0$ (zero-inventory ordering in period j), $s_t = 0$ (zero-inventory ordering in period $t+1$, or $s_m = 0$), and nothing is produced in periods $j+1, \dots, t$; summing the balancing constraints of periods $j, \dots, t$ gives $x_j = D_{j,t}$.
- Period 1 always produces, since $s_0 = 0$ and $d_1 \geq 1$. A solution with the zero-inventory ordering property is therefore determined by the set of its production periods, which contains period 1 and any subset of $\{2, 3, \dots, m\}$.

The number of ULS solutions with the zero-inventory ordering property is 2^{m-1}, since each period $t \in \{2, 3, \dots, m\}$ can independently produce or not.

- The number of these solutions grows exponentially with the number m of periods:
$$2^{m-1} = O(2^m).$$
Therefore, enumerating them quickly becomes impractical as m increases.

Example 10.3: Enumeration of the solutions with the zero-inventory ordering property

We consider the ULS instance of Example 10.1:

t	1	2	3	4	5
d_t	20	10	30	40	10
sc_t	50	50	50	50	50
vc_t	0	0	0	0	0
hc_t	1	1	1	1	1

There are $2^{5-1} = 16$ solutions with the zero-inventory ordering property. For each of them, the table reports the production periods, the solution vector, the inventories, and the total cost, written as setup cost plus holding cost. The optimal costs are highlighted.

Label	Production periods	Solution vector	Inventories $(s_1, \ldots, s_5)$	Total cost
(1)	$\{1\}$	$(110, 0, 0, 0, 0)$	$(90, 80, 50, 10, 0)$	$50 + 230 = 280$
(2)	$\{1, 5\}$	$(100, 0, 0, 0, 10)$	$(80, 70, 40, 0, 0)$	$100 + 190 = 290$
(3)	$\{1, 4\}$	$(60, 0, 0, 50, 0)$	$(40, 30, 0, 10, 0)$	$100 + 80 = 180$
(4)	$\{1, 4, 5\}$	$(60, 0, 0, 40, 10)$	$(40, 30, 0, 0, 0)$	$150 + 70 = 220$
(5)	$\{1, 3\}$	$(30, 0, 80, 0, 0)$	$(10, 0, 50, 10, 0)$	$100 + 70 = 170$
(6)	$\{1, 3, 5\}$	$(30, 0, 70, 0, 10)$	$(10, 0, 40, 0, 0)$	$150 + 50 = 200$
(7)	$\{1, 3, 4\}$	$(30, 0, 30, 50, 0)$	$(10, 0, 0, 10, 0)$	$150 + 20 = 170$
(8)	$\{1, 3, 4, 5\}$	$(30, 0, 30, 40, 10)$	$(10, 0, 0, 0, 0)$	$200 + 10 = 210$
(9)	$\{1, 2\}$	$(20, 90, 0, 0, 0)$	$(0, 80, 50, 10, 0)$	$100 + 140 = 240$
(10)	$\{1, 2, 5\}$	$(20, 80, 0, 0, 10)$	$(0, 70, 40, 0, 0)$	$150 + 110 = 260$
(11)	$\{1, 2, 4\}$	$(20, 40, 0, 50, 0)$	$(0, 30, 0, 10, 0)$	$150 + 40 = 190$
(12)	$\{1, 2, 4, 5\}$	$(20, 40, 0, 40, 10)$	$(0, 30, 0, 0, 0)$	$200 + 30 = 230$
(13)	$\{1, 2, 3\}$	$(20, 10, 80, 0, 0)$	$(0, 0, 50, 10, 0)$	$150 + 60 = 210$
(14)	$\{1, 2, 3, 5\}$	$(20, 10, 70, 0, 10)$	$(0, 0, 40, 0, 0)$	$200 + 40 = 240$
(15)	$\{1, 2, 3, 4\}$	$(20, 10, 30, 50, 0)$	$(0, 0, 0, 10, 0)$	$200 + 10 = 210$
(16)	$\{1, 2, 3, 4, 5\}$	$(20, 10, 30, 40, 10)$	$(0, 0, 0, 0, 0)$	$250 + 0 = 250$

The optimal value of the ULS instance is 170, attained by 2 solutions: (5) and (7).

10.2 A DP algorithm for the ULS

10.2.1 The cost of a production run

- By the zero-inventory ordering property, an optimal solution is a sequence of **production runs**: a run produces in a period j the demands of periods $j, \ldots, t$, and the next run starts in period $t+1$ with an empty inventory.

- For any $j, t \in T$ with $j \leq t$, let $c_{j,t}$ be the cost of the production run that produces in period j the demands of periods $j, \ldots, t$:

$$c_{j,t} = sc_j + vc_j\, D_{j,t} + \sum_{k=j}^{t-1} hc_k\, D_{k+1,t}. \tag{10.2}$$

- The first term is the setup cost of period j, the second term is the variable cost of the $D_{j,t}$ produced units, and the last term is the holding cost: at the end of each period $k \in \{j, \ldots, t-1\}$ the inventory contains the demands $D_{k+1,t}$ of the following periods of the run.

- For a fixed period j, the costs $c_{j,j}, c_{j,j+1}, \ldots, c_{j,m}$ can be computed one after the other. A unit used in period t and produced in period j costs

$$u_{j,t} = vc_j + \sum_{k=j}^{t-1} hc_k,$$

so that $u_{j,j} = vc_j$, $u_{j,t} = u_{j,t-1} + hc_{t-1}$, and adding period t to the run adds d_t units of cost $u_{j,t}$:

$$c_{j,j} = sc_j + vc_j\, d_j, \qquad c_{j,t} = c_{j,t-1} + u_{j,t}\, d_t \quad \text{if } t > j.$$

This is the pseudo-code that computes the matrix of the production-run costs:

Algorithm

Algorithm: CostMatrix

Input: A ULS instance given by the number m of periods and the arrays $d[1..m]$, $sc[1..m]$, $vc[1..m]$, $hc[1..m]$

Output: The matrix of the production-run costs

```
1  Function CostMatrix(m, d, sc, vc, hc)
2      c := new array[1..m][1..m]
3      for j := 1 to m do
4          u := vc[j]
5          g := sc[j] + vc[j] · d[j]
6          c[j][j] ← g
7          for t := j + 1 to m do
8              u ← u + hc[t − 1]
9              g ← g + u · d[t]
10             c[j][t] ← g
11         end
12     end
13     return c
```

- The variable u is the unit cost $u_{j,t}$ and the variable g is the cost $c_{j,t}$ of the current run; only the cells $c[j][t]$ with $j \leq t$ are used.

Example 10.4: Computation of the production-run costs

We consider the ULS instance of Example 10.1:

t	1	2	3	4	5
d_t	20	10	30	40	10
sc_t	50	50	50	50	50
vc_t	0	0	0	0	0
hc_t	1	1	1	1	1

For each step, the table shows the values of j and t, the current value of u, the cell $c[j][t]$, and the operation performed; the cell $c[j][t]$ is highlighted in blue when assigned.

j	t	u	$c[j][t]$	Action
1	1	0	50	$u \leftarrow vc[1] = 0$, $g \leftarrow sc[1]+vc[1] \cdot d[1] = 50+0 \cdot 20 = 50$, $c[1][1] \leftarrow 50$
1	2	1	60	$u \leftarrow u+hc[1] = 0+1 = 1$, $g \leftarrow g+u \cdot d[2] = 50+1 \cdot 10 = 60$, $c[1][2] \leftarrow 60$
1	3	2	120	$u \leftarrow u+hc[2] = 1+1 = 2$, $g \leftarrow g+u \cdot d[3] = 60+2 \cdot 30 = 120$, $c[1][3] \leftarrow 120$
1	4	3	240	$u \leftarrow u+hc[3] = 2+1 = 3$, $g \leftarrow g+u \cdot d[4] = 120+3 \cdot 40 = 240$, $c[1][4] \leftarrow 240$
1	5	4	280	$u \leftarrow u+hc[4] = 3+1 = 4$, $g \leftarrow g+u \cdot d[5] = 240+4 \cdot 10 = 280$, $c[1][5] \leftarrow 280$
2	2	0	50	$u \leftarrow vc[2] = 0$, $g \leftarrow sc[2]+vc[2] \cdot d[2] = 50+0 \cdot 10 = 50$, $c[2][2] \leftarrow 50$
2	3	1	80	$u \leftarrow u+hc[2] = 0+1 = 1$, $g \leftarrow g+u \cdot d[3] = 50+1 \cdot 30 = 80$, $c[2][3] \leftarrow 80$
2	4	2	160	$u \leftarrow u+hc[3] = 1+1 = 2$, $g \leftarrow g+u \cdot d[4] = 80+2 \cdot 40 = 160$, $c[2][4] \leftarrow 160$
2	5	3	190	$u \leftarrow u+hc[4] = 2+1 = 3$, $g \leftarrow g+u \cdot d[5] = 160+3 \cdot 10 = 190$, $c[2][5] \leftarrow 190$
3	3	0	50	$u \leftarrow vc[3] = 0$, $g \leftarrow sc[3]+vc[3] \cdot d[3] = 50+0 \cdot 30 = 50$, $c[3][3] \leftarrow 50$
3	4	1	90	$u \leftarrow u+hc[3] = 0+1 = 1$, $g \leftarrow g+u \cdot d[4] = 50+1 \cdot 40 = 90$, $c[3][4] \leftarrow 90$
3	5	2	110	$u \leftarrow u+hc[4] = 1+1 = 2$, $g \leftarrow g+u \cdot d[5] = 90+2 \cdot 10 = 110$, $c[3][5] \leftarrow 110$
4	4	0	50	$u \leftarrow vc[4] = 0$, $g \leftarrow sc[4]+vc[4] \cdot d[4] = 50+0 \cdot 40 = 50$, $c[4][4] \leftarrow 50$
4	5	1	60	$u \leftarrow u+hc[4] = 0+1 = 1$, $g \leftarrow g+u \cdot d[5] = 50+1 \cdot 10 = 60$, $c[4][5] \leftarrow 60$
5	5	0	50	$u \leftarrow vc[5] = 0$, $g \leftarrow sc[5]+vc[5] \cdot d[5] = 50+0 \cdot 10 = 50$, $c[5][5] \leftarrow 50$

This is the matrix c after the execution of the algorithm (for instance, $c_{1,3} = sc_1 + vc_1\, D_{1,3} + hc_1\, D_{2,3} + hc_2\, D_{3,3} = 50 + 0 \cdot 60 + 1 \cdot 40 + 1 \cdot 30 = 120$ by formula (10.2)):

$c[j][t]$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
$j=1$	50	60	120	240	280
$j=2$	–	50	80	160	190
$j=3$	–	–	50	90	110
$j=4$	–	–	–	50	60
$j=5$	–	–	–	–	50

10.2.2 Step 1 and Step 2

- Step 1: computation of the optimal value

For any $t \in \{1, 2, \dots, m\}$, let $v(t)$ be the optimal value of the ULS subinstance given by:

1. a set $\{1, 2, \dots, t\}$ of t time periods,
2. for each period $j \in \{1, 2, \dots, t\}$, a demand d_j,
3. for each period $j \in \{1, 2, \dots, t\}$, a setup cost sc_j, a variable production cost vc_j, and a holding cost hc_j.

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

$$v(t) = \begin{cases} \min \left\{ v(j-1) + c_{j,t} : j \in \{1, 2, \dots, t\} \right\} & \text{if } 1 \leq t \leq m, \\ 0 & \text{if } t = 0. \end{cases} \tag{10.3}$$

- The subinstance with periods $\{1, \dots, t\}$ is a ULS instance of the same type: its inventory must be empty at the end of period t. Here, $v(0) = 0$, since an empty set of periods has no demand and no cost.
- The recursive decomposition process of formula (10.3) generates t **independent subproblems**, each corresponding to a possible choice of the period j of the **last production run**, which produces the demands of periods $j, \dots, t$.
- By the zero-inventory ordering property applied to the subinstance, the inventory is empty at the end of period $j-1$, and the residual subproblem is the ULS subinstance with periods $\{1, \dots, j-1\}$.
- The optimal value $v(t)$ is obtained by trying all possible periods of the last production run and selecting the one that minimizes the sum $v(j-1) + c_{j,t}$ of the optimal value of the residual subproblem and the cost of the last run.
- Hence, an optimal solution for periods $\{1, \dots, t\}$ is built by combining the chosen last production run in period j with an optimal solution of the residual subproblem with periods $\{1, \dots, j-1\}$.

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

- Different choices of the last production run, possibly made for different values of t, can lead to the same residual set of periods, and therefore the same value $v(t)$ may need to be computed several times during the recursive evaluation of formula (10.3).

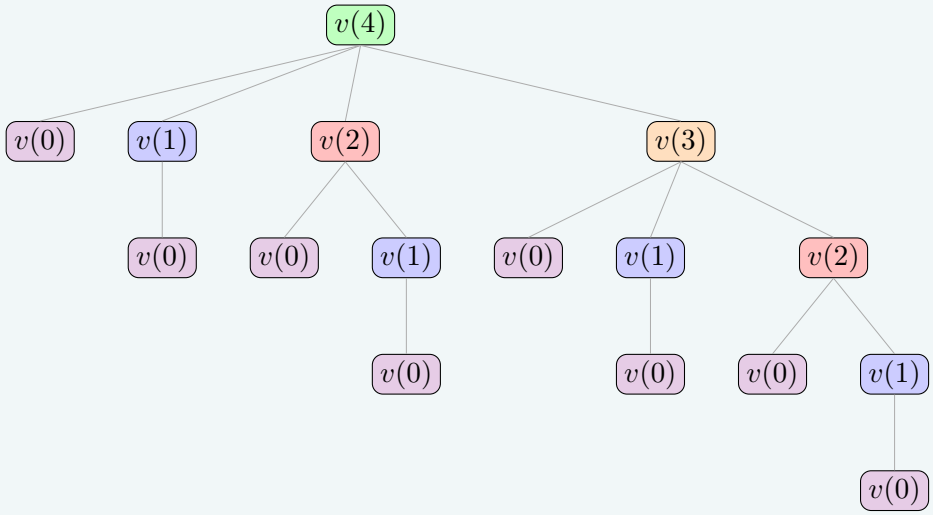
$\rightsquigarrow$ **Overlapping Subproblems verified!**

– A **first method** to compute the optimal value $v(m)$ of the ULS instance is the **top-down approach**. The method starts from the call with $t = m$ and evaluates formula (10.3) recursively: each call for periods $\{1, \ldots, t\}$ triggers further calls for periods $\{1, \ldots, j-1\}$, for all $j \in \{1, 2, \ldots, t\}$, until the base case $v(0)$ 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 10.5: Recursion tree

We consider the first four periods of the ULS instance of Example 10.1 (with five periods the tree has 32 nodes). This is the recursion tree of the top-down approach to compute $v(4)$:



The children of a node $v(t)$ are ordered by $j \in \{1, \ldots, t\}$: the child $v(j-1)$ corresponds to a last production run in period j. The function v is called 16 times, and the same subproblems are solved multiple times: $v(2)$ is called 2 times, $v(1)$ is called 4 times, and $v(0)$ is called 8 times. All the other calls are made only once.

– The value $v(t)$, for a generic period t, can be computed once the values $v(j)$ are known for all smaller periods $j < t$.

Example 10.6: Computing the values of the v function

We consider the ULS instance of Example 10.1, with the matrix c of the production-run costs:

$c[j][t]$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
$j=1$	50	60	120	240	280
$j=2$	–	50	80	160	190
$j=3$	–	–	50	90	110
$j=4$	–	–	–	50	60
$j=5$	–	–	–	–	50

The value of the function v can be computed by starting from the base case $v(0) = 0$ and then computing the values of $v(1)$, $v(2)$, $v(3)$, $v(4)$, and $v(5)$ in order, as follows:

* Since we know the value of $v(0)$, we can compute $v(1)$, as follows:

$$v(1) = v(0) + c_{1,1} = 0 + 50 = 50$$

* Once we have computed $v(1)$ and $v(0)$, we can compute $v(2)$, as follows:

$$\begin{aligned} v(2) &= \min\{v(0) + c_{1,2}, v(1) + c_{2,2}\} \\ &= \min\{0 + 60, 50 + 50\} = 60 \end{aligned}$$

* Once we have computed $v(2)$, $v(1)$ and $v(0)$, we can compute $v(3)$, as follows:

$$\begin{aligned} v(3) &= \min\{v(0) + c_{1,3}, v(1) + c_{2,3}, v(2) + c_{3,3}\} \\ &= \min\{0 + 120, 50 + 80, 60 + 50\} = 110 \end{aligned}$$

* Once we have computed $v(3)$, $v(2)$, $v(1)$ and $v(0)$, we can compute $v(4)$, as follows:

$$\begin{aligned} v(4) &= \min\{v(0) + c_{1,4}, v(1) + c_{2,4}, v(2) + c_{3,4}, v(3) + c_{4,4}\} \\ &= \min\{0 + 240, 50 + 160, 60 + 90, 110 + 50\} = 150 \end{aligned}$$

* Once we have computed $v(4)$, $v(3)$, $v(2)$, $v(1)$ and $v(0)$, we can compute $v(5)$, as follows:

$$\begin{aligned} v(5) &= \min\{v(0) + c_{1,5}, v(1) + c_{2,5}, v(2) + c_{3,5}, v(3) + c_{4,5}, v(4) + c_{5,5}\} \\ &= \min\{0 + 280, 50 + 190, 60 + 110, 110 + 60, 150 + 50\} = 170 \end{aligned}$$

– 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**, the vector $v[0..m]$, whose cell $v[t]$ contains $v(t)$.

– Since solving a subproblem with value $v(t)$ depends only on subproblems with values $v(j-1)$, with $j-1 < t$, we can solve the subproblems for increasing values of t, starting from the base case $v(0) = 0$ and building up to $v(m)$: this observation

naturally leads to a **much more efficient method** to compute the optimal value $v(m)$ of the ULS instance.

– A **second method** to compute the optimal value $v(m)$ of the ULS instance is the **bottom-up approach**. The method starts from the base case $v(0) = 0$ and evaluates formula (10.3) for increasing periods $t \in \{1, 2, \ldots, m\}$: each value $v(t)$ is computed using the already-computed values $v(0), v(1), \ldots, v(t-1)$, until the value $v(m)$ of the ULS instance is obtained.

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

Algorithm

Algorithm: DP_WW

Input: A ULS instance given by the number m of periods and the arrays $d[1..m]$, $sc[1..m]$, $vc[1..m]$, $hc[1..m]$

Output: The optimal value

```
1  Function DP_WW(m, d, sc, vc, hc)
2      c := CostMatrix(m, d, sc, vc, hc)
3      v := new array[0..m]
4      v[0] ← 0
5      for t := 1 to m do
6          b := +∞
7          for j := 1 to t do
8              if v[j − 1] + c[j][t] < b then
9                  b ← v[j − 1] + c[j][t]
10             end
11         end
12         v[t] ← b
13     end
14     return v[m]
```

* The matrix c contains the production-run costs, computed by the algorithm CostMatrix.
* The vector $v[0..m]$ is the **lookup table**: it stores the optimal values of all subproblems, with the base case $v[0] = 0$.
* The outer loop, over $t \in \{1, 2, \ldots, m\}$, computes $v[t]$ in increasing order of the last period.
* The inner loop, over $j \in \{1, 2, \ldots, t\}$, evaluates all possible periods of the last production run and keeps in b the smallest value $v[j-1] + c[j][t]$; the variable b starts from $+\infty$, so the first comparison always updates it.
* At the end of the inner loop, $v[t] \leftarrow b$ stores the optimal value $v[t]$.

∗ Finally, the optimal value $v[m]$ of the ULS instance is returned.

Example 10.7: Execution of the DP algorithm

We consider the ULS instance of Example 10.1, with the matrix c of the production-run costs:

$c[j][t]$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
$j=1$	50	60	120	240	280
$j=2$	–	50	80	160	190
$j=3$	–	–	50	90	110
$j=4$	–	–	–	50	60
$j=5$	–	–	–	–	50

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

t	j	$v[t]$	b	Action
–	–	0	–	$v[0] \leftarrow 0$
	–	–	$+\infty$	$b \leftarrow +\infty$
1	1	–	50	$v[0]+c[1][1] = 0+50 = 50 < +\infty$: $b \leftarrow 50$
	–	50	50	$v[1] \leftarrow b = 50$
	–	–	$+\infty$	$b \leftarrow +\infty$
2	1	–	60	$v[0]+c[1][2] = 0+60 = 60 < +\infty$: $b \leftarrow 60$
	2	–	60	$v[1]+c[2][2] = 50+50 = 100 > 60$: no update
	–	60	60	$v[2] \leftarrow b = 60$
	–	–	$+\infty$	$b \leftarrow +\infty$
3	1	–	120	$v[0]+c[1][3] = 0+120 = 120 < +\infty$: $b \leftarrow 120$
	2	–	120	$v[1]+c[2][3] = 50+80 = 130 > 120$: no update
	3	–	110	$v[2]+c[3][3] = 60+50 = 110 < 120$: $b \leftarrow 110$
	–	110	110	$v[3] \leftarrow b = 110$
	–	–	$+\infty$	$b \leftarrow +\infty$
4	1	–	240	$v[0]+c[1][4] = 0+240 = 240 < +\infty$: $b \leftarrow 240$
	2	–	210	$v[1]+c[2][4] = 50+160 = 210 < 240$: $b \leftarrow 210$
	3	–	150	$v[2]+c[3][4] = 60+90 = 150 < 210$: $b \leftarrow 150$
	4	–	150	$v[3]+c[4][4] = 110+50 = 160 > 150$: no update
	–	150	150	$v[4] \leftarrow b = 150$
	–	–	$+\infty$	$b \leftarrow +\infty$
5	1	–	280	$v[0]+c[1][5] = 0+280 = 280 < +\infty$: $b \leftarrow 280$
	2	–	240	$v[1]+c[2][5] = 50+190 = 240 < 280$: $b \leftarrow 240$
	3	–	170	$v[2]+c[3][5] = 60+110 = 170 < 240$: $b \leftarrow 170$
	4	–	170	$v[3]+c[4][5] = 110+60 = 170 = b$: no update
	5	–	170	$v[4]+c[5][5] = 150+50 = 200 > 170$: no update
	–	170	170	$v[5] \leftarrow b = 170$

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

	$t=0$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
$v[t]$	0	50	60	110	150	170

- After the execution of the DP algorithm, the vector v contains the optimal value $v[t]$ for every set of periods $\{1, \ldots, t\}$.

- *Remark (shortest path).* Formula (10.3) computes a shortest path in a network with nodes $0, 1, \ldots, m$ and one arc from node $j-1$ to node t, of cost $c_{j,t}$, for every $j \leq t$. The value $v(t)$ is the length of a shortest path from node 0 to node t. This is the shortest-path formulation of the chapter «Formulations of lot sizing problems» of the module *Combinatorial Optimization Problems*, without the capacities: an arc is a production run.

- Step 2: computation of an optimal solution
  - To compute an optimal solution, it is necessary to store some additional information.
  - For each period t, in addition to the optimal value $v[t]$, we store $r[t]$, the period of the last production run in an optimal solution for periods $\{1, \ldots, t\}$.
  - Starting from $t = m$, the value $r[t]$ gives the period of the last production run, which produces $D_{r[t],t}$ units; the remaining periods are $\{1, \ldots, r[t]-1\}$, and the same rule can be applied again, and so on, until no period remains.

    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

Algorithm: DP_WW_InfoSol

Input: A ULS instance given by the number m of periods and the arrays $d[1..m]$, $sc[1..m]$, $vc[1..m]$, $hc[1..m]$

Output: The optimal value and the information vector $r[1..m]$

```
1  Function DP_WW_InfoSol(m, d, sc, vc, hc)
2      c := CostMatrix(m, d, sc, vc, hc)
3      v := new array[0..m]
4      r := new array[1..m]
5      v[0] ← 0
6      for t := 1 to m do
7          b := +∞
8          a := 0
9          for j := 1 to t do
10             if v[j − 1] + c[j][t] < b then
11                 b ← v[j − 1] + c[j][t]
12                 a ← j
13             end
14         end
15         v[t] ← b
16         r[t] ← a
17     end
18     return v[m], r
```

* The DP algorithm uses the additional temporary variable a to keep track of the decision made when b is updated. If the last production run in period j is selected, $a \leftarrow j$ stores this decision. At the end of the inner loop, $r[t] \leftarrow a$ stores the period of the selected last production run, together with the optimal value $v[t] \leftarrow b$.

* Because of the strict comparison, in case of ties the smallest period j is stored.

Example 10.8: Computation of the additional information

We consider the ULS instance of Example 10.1, with the matrix c of the production-run costs:

$c[j][t]$	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
$j=1$	50	60	120	240	280
$j=2$	–	50	80	160	190
$j=3$	–	–	50	90	110
$j=4$	–	–	–	50	60
$j=5$	–	–	–	–	50

For each step of the DP algorithm, the table shows the values of t and j, the cell $r[t]$, the current value of b, and the current value of a. The cell of b is highlighted

in red when updated; the cell of a and the cell of $r[t]$ are highlighted in blue when updated.

t	j	$r[t]$	b	a	Action
–	–	–	–	–	Initial state
	–	–	$+\infty$	0	$b \leftarrow +\infty$, $a \leftarrow 0$
1	1	–	50	1	$v[0]+c[1][1] = 0+50 = 50 < +\infty$: $b \leftarrow 50$, $a \leftarrow 1$
	–	1	50	1	$v[1] \leftarrow b = 50$, $r[1] \leftarrow a = 1$
	–	–	$+\infty$	0	$b \leftarrow +\infty$, $a \leftarrow 0$
2	1	–	60	1	$v[0]+c[1][2] = 0+60 = 60 < +\infty$: $b \leftarrow 60$, $a \leftarrow 1$
	2	–	60	1	$v[1]+c[2][2] = 50+50 = 100 > 60$: no update
	–	1	60	1	$v[2] \leftarrow b = 60$, $r[2] \leftarrow a = 1$
	–	–	$+\infty$	0	$b \leftarrow +\infty$, $a \leftarrow 0$
3	1	–	120	1	$v[0]+c[1][3] = 0+120 = 120 < +\infty$: $b \leftarrow 120$, $a \leftarrow 1$
	2	–	120	1	$v[1]+c[2][3] = 50+80 = 130 > 120$: no update
	3	–	110	3	$v[2]+c[3][3] = 60+50 = 110 < 120$: $b \leftarrow 110$, $a \leftarrow 3$
	–	3	110	3	$v[3] \leftarrow b = 110$, $r[3] \leftarrow a = 3$
	–	–	$+\infty$	0	$b \leftarrow +\infty$, $a \leftarrow 0$
4	1	–	240	1	$v[0]+c[1][4] = 0+240 = 240 < +\infty$: $b \leftarrow 240$, $a \leftarrow 1$
	2	–	210	2	$v[1]+c[2][4] = 50+160 = 210 < 240$: $b \leftarrow 210$, $a \leftarrow 2$
	3	–	150	3	$v[2]+c[3][4] = 60+90 = 150 < 210$: $b \leftarrow 150$, $a \leftarrow 3$
	4	–	150	3	$v[3]+c[4][4] = 110+50 = 160 > 150$: no update
	–	3	150	3	$v[4] \leftarrow b = 150$, $r[4] \leftarrow a = 3$
	–	–	$+\infty$	0	$b \leftarrow +\infty$, $a \leftarrow 0$
5	1	–	280	1	$v[0]+c[1][5] = 0+280 = 280 < +\infty$: $b \leftarrow 280$, $a \leftarrow 1$
	2	–	240	2	$v[1]+c[2][5] = 50+190 = 240 < 280$: $b \leftarrow 240$, $a \leftarrow 2$
	3	–	170	3	$v[2]+c[3][5] = 60+110 = 170 < 240$: $b \leftarrow 170$, $a \leftarrow 3$
	4	–	170	3	$v[3]+c[4][5] = 110+60 = 170 = b$: no update
	5	–	170	3	$v[4]+c[5][5] = 150+50 = 200 > 170$: no update
	–	3	170	3	$v[5] \leftarrow b = 170$, $r[5] \leftarrow a = 3$

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

	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
$r[t]$	1	1	3	3	3

– Once the vector r has been computed, an optimal solution of the ULS instance can be computed by repeatedly applying the following rule, starting from $t = m$ and $x_j = 0$ for all $j \in \{1, 2, \ldots, m\}$:

$$x_{r[t]} \leftarrow D_{r[t],t}, \qquad t \leftarrow r[t] - 1,$$

until $t = 0$. This yields the vector $(x_1, x_2, \ldots, x_m)$, where each entry x_j is the number of units produced in period j in an optimal solution.

– The following pseudo-code implements the computation of an optimal solution, given the vector r.

Algorithm

Algorithm: DP_WW_SolComputation

Input: The number m of periods and the demands $d[1..m]$ of the ULS instance, and the vector $r[1..m]$

Output: An optimal solution of the ULS instance

```
1  Function DP_WW_SolComputation(m, d, r)
2      x := new array[1..m]
3      for j := 1 to m do
4          x[j] ← 0
5      end
6      t := m
7      while t > 0 do
8          x[r[t]] ← Σ_{k=r[t]}^{t} d[k]
9          t ← r[t] − 1
10     end
11     return x
```

Example 10.9: Computing an optimal solution

After the execution of the DP algorithm on the instance of Example 10.1, we have the following vector r:

	$t=1$	$t=2$	$t=3$	$t=4$	$t=5$
$r[t]$	1	1	3	3	3

We can compute an optimal solution for $m = 5$ periods as follows:

Iteration	t	$r[t]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$x[5]$	$t \leftarrow r[t]-1$
–	–	–	0	0	0	0	0	$t=5$
1	5	3	0	0	80	0	0	$t=3-1=2$
2	2	1	30	0	80	0	0	$t=1-1=0$

* $t = 5$, $r[5] = 3$: the last production run of periods $1, \ldots, 5$ is in period 3 and covers the demands of periods $3, \ldots, 5$, so $x[3] \leftarrow d_3 + d_4 + d_5 = 30 + 40 + 10 = 80$
* $t = 2$, $r[2] = 1$: the last production run of periods $1, \ldots, 2$ is in period 1 and covers the demands of periods $1, \ldots, 2$, so $x[1] \leftarrow d_1 + d_2 = 20 + 10 = 30$

An optimal solution is the vector:

$$(30, 0, 80, 0, 0)$$

The corresponding optimal value is $v[5] = 170$, the same value obtained by the enumeration of Example 10.3 and by Gurobi on the ILP_{ULS} model.

– *Remark (ties).* In the computation of $v[5]$, the run in period 4 gives the same value

as the run in period 3, but the strict comparison keeps $j = 3$. Choosing $j = 4$ instead leads to the other optimal solution $(30, 0, 30, 50, 0)$ of Example 10.3. The DP algorithm returns one optimal solution, not all of them.

10.2.3 Number of steps

- Let $S_c(m)$ denote the number of steps executed by CostMatrix in the worst case, where m is the number of periods. The number of executions of its lines is:
  - Line 2 is executed 1 time.
  - Line 3, Line 4, Line 5, and Line 6 are executed m times each.
  - Line 7, Line 8, Line 9, and Line 10 are executed $\sum_{j=1}^{m}(m-j) = \frac{m(m-1)}{2}$ times each.
  - Line 13 is executed 1 time.

  Therefore,
  $$S_c(m) = 2 + 4m + 4 \cdot \frac{m(m-1)}{2} = 2m^2 + 2m + 2.$$
- Let $S(m)$ denote the number of steps executed by DP_WW in the worst case. Line 2 executes CostMatrix, that is, $S_c(m)$ steps. In the worst case, the condition at line 8 is true at every iteration of the inner loop, and therefore line 9 is executed as often as line 8. The number of executions of the other lines is:
  - Line 3 is executed 1 time.
  - Line 4 is executed 1 time.
  - Line 5 is executed m times.
  - Line 6 is executed m times.
  - Line 7 is executed $\sum_{t=1}^{m} t$ times.
  - Line 8 is executed $\sum_{t=1}^{m} t$ times.
  - Line 9 is executed $\sum_{t=1}^{m} t$ times.
  - Line 12 is executed m times.
  - Line 14 is executed 1 time.

  Therefore, the total number of steps in the worst case is
  $$S(m) = S_c(m)+3+3m+3\sum_{t=1}^{m} t = 2m^2+2m+2+3+3m+3\cdot\frac{m(m+1)}{2} = \frac{7m^2+13m+10}{2}.$$
  Hence,
  $$S(m) = \Theta(m^2).$$
  Indeed,
  $$\lim_{m\to\infty} \frac{\frac{7m^2+13m+10}{2}}{m^2} = \lim_{m\to\infty} \frac{1}{2}\left(7 + \frac{13}{m} + \frac{10}{m^2}\right) = \frac{7}{2} \in \mathbb{R}_{>0}.$$

The DP algorithm executes

$$S(m) = \Theta(m^2)$$

steps in the worst case.

- DP_WW_InfoSol only adds a constant number of operations inside the same loops, and therefore its number of steps has the same order of growth.

10.3 Typical mistakes

Warning

- **Applying the zero-inventory ordering property with capacities.** The proof moves production between two periods without limits: with capacities a period can produce while units are in stock.
- **Computing the cost of a run from scratch.** Computing every $c_{j,t}$ from its definition costs $O(m^3)$ steps inside the DP algorithm; the incremental computation of CostMatrix keeps $\Theta(m^2)$.
- **Forgetting the empty final inventory.** The subinstance with periods $\{1, \dots, t\}$ ends with an empty inventory: this is why the run in period j produces exactly $D_{j,t}$ units.
- **Reading one optimal plan as the only one.** Example 10.1 has two optimal plans: the DP algorithm returns the one with the smallest period j, because it updates only on a strictly smaller value.

10.4 In summary

Summary

- If all costs are non-negative, the uncapacitated problem has an optimal plan in which a period produces only when the inventory is zero (Wagner and Whitin), so $x_j = D_{j,t}$ for every production run.
- The cost $c_{j,t}$ of every run is computed in $\Theta(m^2)$ steps, and the recurrence $v(t) = \min_j\{v(j-1) + c_{j,t}\}$ gives the optimal value in $\Theta(m^2)$ steps; it is a shortest path.
- The zero-inventory ordering property rests on production without limits: with capacities it fails.

Dashboard of the example.

Instance	Lower bound	Upper bound	Origin
Example 10.1 (uncapacitated, $m = 5$)	170	170	DP algorithm of Wagner and Whitin

Chapter 11

DP for the capacitated lot sizing problem

Prerequisites

- From this module, the chapter «DP for the uncapacitated lot sizing problem»: the total demands $D_{j,t}$, the encoding of a production plan and its network, the zero-inventory ordering property, and the instance of its Example 10.1.
- From the module *Combinatorial Optimization Problems*, the chapters «Lot sizing problems» and «Formulations of lot sizing problems»: the capacitated problem, its reference instance, and its ILP model with the variables x_t, s_t, y_t.
- From this module, the chapter «Principles of dynamic programming»: recurrence, base cases, lookup table and reconstruction of a solution.
- From the module *Algorithms*: pseudocode (Chapter 2), and the size of the input (Chapter 11).

Where we were

- **As a problem and as a model**: the capacitated lot sizing problem and its ILP formulations (chapters «Lot sizing problems» and «Formulations of lot sizing problems» of the module *Combinatorial Optimization Problems*).
- **As a table, without capacities**: the DP algorithm of Wagner and Whitin of Chapter 10 «DP for the uncapacitated lot sizing problem», over the production runs, in $\Theta(m^2)$ steps.
- **Now, with capacities**: a DP algorithm on the pairs (period, inventory), which is pseudo-polynomial.

Objectives

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

- explain why the zero-inventory ordering property fails with capacities, on an example;
- write the recurrence $v(t, k)$ on the pairs (period, final inventory), with its base cases;
- run the DP algorithm by hand, fill the matrices v and r, and reconstruct an optimal

plan;

- count the steps of the DP algorithm, and say why it is pseudo-polynomial;
- state the weaker property that holds when the capacity is the same in all periods.

The problem of this chapter is the capacitated lot sizing problem (CLS): its definition and its assumptions are on its page.

11.1 Notation, model and feasible solutions

- As in Chapter 10 «DP for the uncapacitated lot sizing problem», for any $j, t \in T$ we denote by
$$D_{j,t} = \sum_{k=j}^{t} d_k$$
the total demand of periods $j, j+1, \ldots, t$, with $D_{j,t} = 0$ if $j > t$.

- The ILP model is the ILP_{CLSP} model of the chapter «Formulations of lot sizing problems» of the module *Combinatorial Optimization Problems*, with the big-M value $\min\{cap_t, D_{t,m}\}$ of period t: it is the ILP_{ULS} model with the additional constraints $x_t \leq cap_t$, for all $t \in T$.

- A feasible solution is encoded by the same vector $(x_1, x_2, \ldots, x_m)$ as in the ULS, with the inventories $s_t = \sum_{j=1}^{t} x_j - D_{1,t}$, and with the additional requirement $x_t \leq cap_t$ for all $t \in T$.

- The **zero-inventory ordering property does not hold** for the CLS: a period may have to produce while units are still in stock, because the capacity does not allow to produce everything in the period where it would be cheapest.

- The property, and the DP algorithm over the production runs that rests on it, are those of Chapter 10 «DP for the uncapacitated lot sizing problem».

Example 11.1: Capacitated lot sizing

We consider the uncapacitated instance of Example 10.1 of Chapter 10 «DP for the uncapacitated lot sizing problem», measured in units of 10 products (hence the holding cost is 10 per unit), with a constant capacity of 4 units per period:

t	1	2	3	4	5
d_t	2	1	3	4	1
sc_t	50	50	50	50	50
vc_t	0	0	0	0	0
hc_t	10	10	10	10	10
cap_t	4	4	4	4	4

We refer to this CLS instance as Example 11.1: it is the reference instance of the chapter «Lot sizing problems» of the module *Combinatorial Optimization Problems*. The optimal solution of the uncapacitated instance, in units of 10, is the vector $(3, 0, 8, 0, 0)$, of cost 170: it is not feasible for Example 11.1, since period 3 produces more than 4 units.

11.2 A DP algorithm for the CLS

11.2.1 Step 1 and Step 2

- Step 1: computation of the optimal value
- For any period $t \in \{1, \dots, m\}$, the **production cost function** of period t is

$$f_t(x) = \begin{cases} 0 & \text{if } x = 0, \\ sc_t + vc_t\, x & \text{if } x \in \{1, 2, \dots, cap_t\}. \end{cases}$$

The inventory at the end of period t never exceeds the demand $D_{t+1,m}$ of the following periods, since the inventory must be empty at the end of period m.

For any $t \in \{1, 2, \dots, m\}$ and $k \in \{0, 1, \dots, D_{t+1,m}\}$, let $v(t, k)$ be the optimal value of the CLS subinstance given by:

1. a set $\{1, 2, \dots, t\}$ of t time periods,
2. for each period $j \in \{1, 2, \dots, t\}$, a demand d_j,
3. for each period $j \in \{1, 2, \dots, t\}$, a setup cost sc_j, a variable production cost vc_j, and a holding cost hc_j,
4. for each period $j \in \{1, 2, \dots, t\}$, a capacity cap_j,
5. an inventory equal to k at the end of period t.

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

$$v(t,k) = \begin{cases} \min\left\{v(t-1, k+d_t-x) + f_t(x) : x \in X_t(k)\right\} & \text{if } 1 \le t \le m, \\ \quad + hc_t\, k & \\ 0 & \text{if } t = 0 \text{ and } k = 0, \\ +\infty & \text{if } t = 0 \text{ and } k > 0, \end{cases} \tag{11.1}$$

where $X_t(k) = \{0, 1, \dots, \min\{cap_t,\, k + d_t\}\}$ is the set of the possible productions of period t.

- The pair (t, k) is the **state**: the last period of the subinstance and the inventory at its end. For $t = m$ and $k = 0$ we obtain the CLS instance itself, so its optimal value is $v(m, 0)$.

- The recursive decomposition process of formula (11.1) generates at most $cap_t + 1$ **independent subproblems**, each corresponding to a possible choice of the production x of the last period t.

- If x units are produced in period t and k units remain at its end, the inventory at the end of period $t-1$ is $k+d_t-x$: the residual subproblem is the CLS subinstance with periods $\{1,\dots,t-1\}$ and final inventory $k+d_t-x$.

- The production x is at most cap_t, and at most $k+d_t$ because the inventory at the end of period $t-1$ cannot be negative. Moreover $k+d_t-x \leq D_{t+1,m}+d_t = D_{t,m}$, so the residual subproblem is a state.

- The base case is $v(0,0)=0$: no period, no inventory, no cost.

- For $k>0$ we set $v(0,k)=+\infty$: the initial inventory is empty, so no feasible solution has a positive inventory before period 1. The value $+\infty$ rules out this case, and it propagates to every state that cannot be reached within the capacities, with the convention $+\infty + a = +\infty$.

- The optimal value $v(t,k)$ is obtained by trying all possible productions x of period t and selecting the one that minimizes the sum $v(t-1,k+d_t-x)+f_t(x)$ of the optimal value of the residual subproblem and the production cost of period t; the holding cost $hc_t\,k$ of the final inventory does not depend on x.

- Hence, an optimal solution of the subinstance of state (t,k) is built by combining the chosen production of period t with an optimal solution of the residual subproblem of state $(t-1,k+d_t-x)$.

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

- Different sequences of production decisions can lead to the same state. For instance, in Example 11.1, starting from the state $(5,0)$, the decisions $x_5=0$ and $x_4=4$ lead to the state $(3,1)$, and the decisions $x_5=1$ and $x_4=3$ lead to the same state $(3,1)$.

- Therefore the same value $v(t,k)$ may need to be computed several times during the recursive evaluation of formula (11.1).

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

- A **first method** to compute the optimal value $v(m,0)$ of the CLS instance is the **top-down approach**. The method starts from the call with $(t,k)=(m,0)$ and evaluates formula (11.1) recursively: each call for a state (t,k) triggers further calls for the states $(t-1,k+d_t-x)$, for all $x\in\{0,\dots,\min\{cap_t,k+d_t\}\}$, until one of the base cases $v(0,k)$ is reached.

- The value $v(t,k)$ can be computed once the values $v(t-1,k')$ are known for all $k'\in\{0,\dots,D_{t,m}\}$.

– We can avoid solving the same subproblems repeatedly by storing the value of each subproblem as soon as it is computed, in a **lookup table**: the matrix $v[0..m][0..D_{1,m}]$, whose cell $v[t][k]$ contains $v(t,k)$ for $k \leq D_{t+1,m}$ (the other cells are not used).

– A **second method** to compute the optimal value $v(m,0)$ of the CLS instance is the **bottom-up approach**. The method starts from the base cases $v(0,0) = 0$ and $v(0,k) = +\infty$ for all $k \in \{1,\ldots,D_{1,m}\}$.

– It evaluates formula (11.1) for increasing values of $t \in \{1,\ldots,m\}$ and, for each fixed t, for $k \in \{0,1,\ldots,D_{t+1,m}\}$: each value $v(t,k)$ uses already-computed values of row $t-1$, until $v(m,0)$ is obtained.

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

Algorithm

Algorithm: DP_CLS

Input: A CLS instance given by the number m of periods and the arrays $d[1..m]$, $sc[1..m]$, $vc[1..m]$, $hc[1..m]$, $cap[1..m]$

Output: The optimal value

```
1  Function DP_CLS(m, d, sc, vc, hc, cap)
2      R := new array[0..m]
3      R[m] ← 0
4      for t := m − 1 downto 0 do
5          R[t] ← R[t + 1] + d[t + 1]
6      end
7      v := new array[0..m][0..R[0]]
8      v[0][0] ← 0
9      for k := 1 to R[0] do
10         v[0][k] ← +∞
11     end
12     for t := 1 to m do
13         for k := 0 to R[t] do
14             b := +∞
15             for x := 0 to min{cap[t], k + d[t]} do
16                 if v[t − 1][k + d[t] − x] + f_t(x) + hc[t] · k < b then
17                     b ← v[t − 1][k + d[t] − x] + f_t(x) + hc[t] · k
18                 end
19             end
20             v[t][k] ← b
21         end
22     end
23     return v[m][0]
```

* The vector $R[0..m]$ contains the remaining demands $R[t] = D_{t+1,m}$, computed backwards from $R[m] = 0$: the states of period t are the pairs (t, k) with $k \leq R[t]$.

* The matrix v is the **lookup table**. The initialization sets $v[0][0] = 0$ and $v[0][k] = +\infty$ for all $k \in \{1, \ldots, R[0]\}$, since the initial inventory is empty.

* The outer loop, over $t \in \{1, 2, \ldots, m\}$, considers the periods in increasing order, and the second loop, over $k \in \{0, \ldots, R[t]\}$, considers all final inventories of period t.

* The inner loop, over the production x of period t, keeps in b the smallest value $v[t-1][k + d[t] - x] + f_t(x) + hc[t] \cdot k$; the value $f_t(x)$ is 0 for $x = 0$ and $sc[t] + vc[t] \cdot x$ otherwise. If all candidates are $+\infty$, the comparison is never true and $v[t][k] = +\infty$: the state (t, k) cannot be reached within the capacities.

* Finally, the optimal value $v[m][0]$ of the CLS instance is returned.

Example 11.2: Execution of the DP algorithm

We consider the CLS instance of Example 11.1:

t	1	2	3	4	5
d_t	2	1	3	4	1
sc_t	50	50	50	50	50
vc_t	0	0	0	0	0
hc_t	10	10	10	10	10
cap_t	4	4	4	4	4

The remaining demands are $(R[0], \ldots, R[5]) = (11, 9, 8, 5, 1, 0)$.

For each step of the DP algorithm, the table shows the values of t and k, the cell $v[t][k]$, and the operation performed. The candidates of the minimum are listed for $x = 0, 1, \ldots, \min\{cap_t, k + d_t\}$.

t	k	$v[t][k]$	Action
0	0	0	$v[0][0] \leftarrow 0$
0	1	$+\infty$	$v[0][1] \leftarrow +\infty$
0	2	$+\infty$	$v[0][2] \leftarrow +\infty$
0	3	$+\infty$	$v[0][3] \leftarrow +\infty$
0	4	$+\infty$	$v[0][4] \leftarrow +\infty$
0	5	$+\infty$	$v[0][5] \leftarrow +\infty$
0	6	$+\infty$	$v[0][6] \leftarrow +\infty$
0	7	$+\infty$	$v[0][7] \leftarrow +\infty$
0	8	$+\infty$	$v[0][8] \leftarrow +\infty$
0	9	$+\infty$	$v[0][9] \leftarrow +\infty$
0	10	$+\infty$	$v[0][10] \leftarrow +\infty$
0	11	$+\infty$	$v[0][11] \leftarrow +\infty$
1	0	50	$\min\{v[0][2]+f_1(0), v[0][1]+f_1(1), v[0][0]+f_1(2)\}+hc_1 \cdot 0 = \min\{+\infty, +\infty, 0+50\}+10 \cdot 0 = 50$

1	1	60	$\min\{v[0][3]+f_1(0), v[0][2]+f_1(1), v[0][1]+f_1(2), v[0][0]+f_1(3)\}+hc_1 \cdot 1 = \min\{+\infty, +\infty, +\infty, 0+50\}+10 \cdot 1 = 60$
1	2	70	$\min\{v[0][4]+f_1(0), v[0][3]+f_1(1), v[0][2]+f_1(2), v[0][1]+f_1(3), v[0][0]+f_1(4)\}+hc_1 \cdot 2 = \min\{+\infty, +\infty, +\infty, +\infty, 0+50\}+10 \cdot 2 = 70$
1	3	$+\infty$	the candidates use the cells $v[0][1], \ldots, v[0][5]$, which are all $+\infty$: $v[1][3] \leftarrow +\infty$
1	4	$+\infty$	the candidates use the cells $v[0][2], \ldots, v[0][6]$, which are all $+\infty$: $v[1][4] \leftarrow +\infty$
1	5	$+\infty$	the candidates use the cells $v[0][3], \ldots, v[0][7]$, which are all $+\infty$: $v[1][5] \leftarrow +\infty$
1	6	$+\infty$	the candidates use the cells $v[0][4], \ldots, v[0][8]$, which are all $+\infty$: $v[1][6] \leftarrow +\infty$
1	7	$+\infty$	the candidates use the cells $v[0][5], \ldots, v[0][9]$, which are all $+\infty$: $v[1][7] \leftarrow +\infty$
1	8	$+\infty$	the candidates use the cells $v[0][6], \ldots, v[0][10]$, which are all $+\infty$: $v[1][8] \leftarrow +\infty$
1	9	$+\infty$	the candidates use the cells $v[0][7], \ldots, v[0][11]$, which are all $+\infty$: $v[1][9] \leftarrow +\infty$
2	0	60	$\min\{v[1][1]+f_2(0), v[1][0]+f_2(1)\}+hc_2 \cdot 0 = \min\{60+0, 50+50\}+10 \cdot 0 = 60$
2	1	80	$\min\{v[1][2]+f_2(0), v[1][1]+f_2(1), v[1][0]+f_2(2)\}+hc_2 \cdot 1 = \min\{70+0, 60+50, 50+50\}+10 \cdot 1 = 80$
2	2	120	$\min\{v[1][3]+f_2(0), v[1][2]+f_2(1), v[1][1]+f_2(2), v[1][0]+f_2(3)\}+hc_2 \cdot 2 = \min\{+\infty, 70+50, 60+50, 50+50\}+10 \cdot 2 = 120$
2	3	130	$\min\{v[1][4]+f_2(0), v[1][3]+f_2(1), v[1][2]+f_2(2), v[1][1]+f_2(3), v[1][0]+f_2(4)\}+hc_2 \cdot 3 = \min\{+\infty, +\infty, 70+50, 60+50, 50+50\}+10 \cdot 3 = 130$
2	4	150	$\min\{v[1][5]+f_2(0), v[1][4]+f_2(1), v[1][3]+f_2(2), v[1][2]+f_2(3), v[1][1]+f_2(4)\}+hc_2 \cdot 4 = \min\{+\infty, +\infty, +\infty, 70+50, 60+50\}+10 \cdot 4 = 150$
2	5	170	$\min\{v[1][6]+f_2(0), v[1][5]+f_2(1), v[1][4]+f_2(2), v[1][3]+f_2(3), v[1][2]+f_2(4)\}+hc_2 \cdot 5 = \min\{+\infty, +\infty, +\infty, +\infty, 70+50\}+10 \cdot 5 = 170$
2	6	$+\infty$	the candidates use the cells $v[1][3], \ldots, v[1][7]$, which are all $+\infty$: $v[2][6] \leftarrow +\infty$
2	7	$+\infty$	the candidates use the cells $v[1][4], \ldots, v[1][8]$, which are all $+\infty$: $v[2][7] \leftarrow +\infty$
2	8	$+\infty$	the candidates use the cells $v[1][5], \ldots, v[1][9]$, which are all $+\infty$: $v[2][8] \leftarrow +\infty$
3	0	110	$\min\{v[2][3]+f_3(0), v[2][2]+f_3(1), v[2][1]+f_3(2), v[2][0]+f_3(3)\}+hc_3 \cdot 0 = \min\{130+0, 120+50, 80+50, 60+50\}+10 \cdot 0 = 110$
3	1	120	$\min\{v[2][4]+f_3(0), v[2][3]+f_3(1), v[2][2]+f_3(2), v[2][1]+f_3(3), v[2][0]+f_3(4)\}+hc_3 \cdot 1 = \min\{150+0, 130+50, 120+50, 80+50, 60+50\}+10 \cdot 1 = 120$
3	2	150	$\min\{v[2][5]+f_3(0), v[2][4]+f_3(1), v[2][3]+f_3(2), v[2][2]+f_3(3), v[2][1]+f_3(4)\}+hc_3 \cdot 2 = \min\{170+0, 150+50, 130+50, 120+50, 80+50\}+10 \cdot 2 = 150$
3	3	200	$\min\{v[2][6]+f_3(0), v[2][5]+f_3(1), v[2][4]+f_3(2), v[2][3]+f_3(3), v[2][2]+f_3(4)\}+hc_3 \cdot 3 = \min\{+\infty, 170+50, 150+50, 130+50, 120+50\}+10 \cdot 3 = 200$
3	4	220	$\min\{v[2][7]+f_3(0), v[2][6]+f_3(1), v[2][5]+f_3(2), v[2][4]+f_3(3), v[2][3]+f_3(4)\}+hc_3 \cdot 4 = \min\{+\infty, +\infty, 170+50, 150+50, 130+50\}+10 \cdot 4 = 220$
3	5	250	$\min\{v[2][8]+f_3(0), v[2][7]+f_3(1), v[2][6]+f_3(2), v[2][5]+f_3(3), v[2][4]+f_3(4)\}+hc_3 \cdot 5 = \min\{+\infty, +\infty, +\infty, 170+50, 150+50\}+10 \cdot 5 = 250$

4	0	160	$\min\{v[3][4]+f_4(0), v[3][3]+f_4(1), v[3][2]+f_4(2), v[3][1]+f_4(3), v[3][0]+f_4(4)\}+hc_4 \cdot 0 = \min\{220+0, 200+50, 150+50, 120+50, 110+50\}+10 \cdot 0 = 160$
4	1	180	$\min\{v[3][5]+f_4(0), v[3][4]+f_4(1), v[3][3]+f_4(2), v[3][2]+f_4(3), v[3][1]+f_4(4)\}+hc_4 \cdot 1 = \min\{250+0, 220+50, 200+50, 150+50, 120+50\}+10 \cdot 1 = 180$
5	0	180	$\min\{v[4][1]+f_5(0), v[4][0]+f_5(1)\}+hc_5 \cdot 0 = \min\{180+0, 160+50\}+10 \cdot 0 = 180$

This is the matrix v after the execution of the DP algorithm (a dot marks a pair (t, k) with $k > D_{t+1,m}$, which is not a state):

$v[t][k]$	0	1	2	3	4	5	6	7	8	9	10	11
$t=0$	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$
$t=1$	50	60	70	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$+\infty$	·	·
$t=2$	60	80	120	130	150	170	$+\infty$	$+\infty$	$+\infty$	·	·	·
$t=3$	110	120	150	200	220	250	·	·	·	·	·	·
$t=4$	160	180	·	·	·	·	·	·	·	·	·	·
$t=5$	180	·	·	·	·	·	·	·	·	·	·	·

The optimal value of the CLS instance is $v[5][0] = 180$.

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

- Step 2: computation of an optimal solution
  - To compute an optimal solution, it is necessary to store some additional information.
  - For each state (t, k), in addition to the optimal value $v[t][k]$, we store $r[t][k]$, the production of period t in an optimal solution of the subinstance of state (t, k); for the states with value $+\infty$ we store 0.
  - Starting from $(t, k) = (m, 0)$, the value $r[t][k]$ gives the production of period t; the inventory at the end of period $t-1$ is then $k + d_t - r[t][k]$, and the same rule can be applied again to period $t-1$, and so on, until $t = 0$.

    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

Algorithm: DP_CLS_InfoSol

Input: A CLS instance given by the number m of periods and the arrays $d[1..m]$, $sc[1..m]$, $vc[1..m]$, $hc[1..m]$, $cap[1..m]$

Output: The optimal value and the information matrix $r[0..m][0..D_{1,m}]$

```
1  Function DP_CLS_InfoSol(m, d, sc, vc, hc, cap)
2      R := new array[0..m]
3      R[m] ← 0
4      for t := m − 1 downto 0 do
5          R[t] ← R[t + 1] + d[t + 1]
6      end
7      v := new array[0..m][0..R[0]]
8      r := new array[0..m][0..R[0]]
9      v[0][0] ← 0
10     r[0][0] ← 0
11     for k := 1 to R[0] do
12         v[0][k] ← +∞
13         r[0][k] ← 0
14     end
15     for t := 1 to m do
16         for k := 0 to R[t] do
17             b := +∞
18             a := 0
19             for x := 0 to min{cap[t], k + d[t]} do
20                 if v[t − 1][k + d[t] − x] + f_t(x) + hc[t] · k < b then
21                     b ← v[t − 1][k + d[t] − x] + f_t(x) + hc[t] · k
22                     a ← x
23                 end
24             end
25             v[t][k] ← b
26             r[t][k] ← a
27         end
28     end
29     return v[m][0], r
```

∗ The DP algorithm uses the additional temporary variable a to keep track of the decision made when b is updated: if the production x is selected, $a \leftarrow x$ stores this decision, and at the end of the inner loop $r[t][k] \leftarrow a$ stores it, together with the optimal value $v[t][k] \leftarrow b$. Because of the strict comparison, in case of ties the smallest production is stored.

Example 11.3: Computation of the additional information

We consider the CLS instance of Example 11.1:

t	1	2	3	4	5
d_t	2	1	3	4	1
sc_t	50	50	50	50	50
vc_t	0	0	0	0	0
hc_t	10	10	10	10	10
cap_t	4	4	4	4	4

For each step of the DP algorithm, the table shows the values of t and k, the cell $r[t][k]$, and the production x for which the minimum of the corresponding row of Example 11.2 is attained first.

t	k	$r[t][k]$	Action
0	0	0	$r[0][0] \leftarrow 0$
0	1	0	$r[0][1] \leftarrow 0$
0	2	0	$r[0][2] \leftarrow 0$
0	3	0	$r[0][3] \leftarrow 0$
0	4	0	$r[0][4] \leftarrow 0$
0	5	0	$r[0][5] \leftarrow 0$
0	6	0	$r[0][6] \leftarrow 0$
0	7	0	$r[0][7] \leftarrow 0$
0	8	0	$r[0][8] \leftarrow 0$
0	9	0	$r[0][9] \leftarrow 0$
0	10	0	$r[0][10] \leftarrow 0$
0	11	0	$r[0][11] \leftarrow 0$
1	0	2	first minimum at $x = 2$, i.e., $v[0][0]+f_1(2)+hc_1 \cdot 0 = 0+50+0 = 50$: $r[1][0] \leftarrow 2$
1	1	3	first minimum at $x = 3$, i.e., $v[0][0]+f_1(3)+hc_1 \cdot 1 = 0+50+10 = 60$: $r[1][1] \leftarrow 3$
1	2	4	first minimum at $x = 4$, i.e., $v[0][0]+f_1(4)+hc_1 \cdot 2 = 0+50+20 = 70$: $r[1][2] \leftarrow 4$
1	3	0	all candidates equal $+\infty$: $r[1][3] \leftarrow 0$
1	4	0	all candidates equal $+\infty$: $r[1][4] \leftarrow 0$
1	5	0	all candidates equal $+\infty$: $r[1][5] \leftarrow 0$
1	6	0	all candidates equal $+\infty$: $r[1][6] \leftarrow 0$
1	7	0	all candidates equal $+\infty$: $r[1][7] \leftarrow 0$
1	8	0	all candidates equal $+\infty$: $r[1][8] \leftarrow 0$
1	9	0	all candidates equal $+\infty$: $r[1][9] \leftarrow 0$
2	0	0	first minimum at $x = 0$, i.e., $v[1][1]+f_2(0)+hc_2 \cdot 0 = 60+0+0 = 60$: $r[2][0] \leftarrow 0$
2	1	0	first minimum at $x = 0$, i.e., $v[1][2]+f_2(0)+hc_2 \cdot 1 = 70+0+10 = 80$: $r[2][1] \leftarrow 0$
2	2	3	first minimum at $x = 3$, i.e., $v[1][0]+f_2(3)+hc_2 \cdot 2 = 50+50+20 = 120$: $r[2][2] \leftarrow 3$
2	3	4	first minimum at $x = 4$, i.e., $v[1][0]+f_2(4)+hc_2 \cdot 3 = 50+50+30 = 130$: $r[2][3] \leftarrow 4$
2	4	4	first minimum at $x = 4$, i.e., $v[1][1]+f_2(4)+hc_2 \cdot 4 = 60+50+40 = 150$: $r[2][4] \leftarrow 4$
2	5	4	first minimum at $x = 4$, i.e., $v[1][2]+f_2(4)+hc_2 \cdot 5 = 70+50+50 = 170$: $r[2][5] \leftarrow 4$

2	6	0	all candidates equal $+\infty$: $r[2][6] \leftarrow 0$
2	7	0	all candidates equal $+\infty$: $r[2][7] \leftarrow 0$
2	8	0	all candidates equal $+\infty$: $r[2][8] \leftarrow 0$
3	0	3	first minimum at $x = 3$, i.e., $v[2][0]+f_3(3)+hc_3 \cdot 0 = 60+50+0 = 110$: $r[3][0] \leftarrow 3$
3	1	4	first minimum at $x = 4$, i.e., $v[2][0]+f_3(4)+hc_3 \cdot 1 = 60+50+10 = 120$: $r[3][1] \leftarrow 4$
3	2	4	first minimum at $x = 4$, i.e., $v[2][1]+f_3(4)+hc_3 \cdot 2 = 80+50+20 = 150$: $r[3][2] \leftarrow 4$
3	3	4	first minimum at $x = 4$, i.e., $v[2][2]+f_3(4)+hc_3 \cdot 3 = 120+50+30 = 200$: $r[3][3] \leftarrow 4$
3	4	4	first minimum at $x = 4$, i.e., $v[2][3]+f_3(4)+hc_3 \cdot 4 = 130+50+40 = 220$: $r[3][4] \leftarrow 4$
3	5	4	first minimum at $x = 4$, i.e., $v[2][4]+f_3(4)+hc_3 \cdot 5 = 150+50+50 = 250$: $r[3][5] \leftarrow 4$
4	0	4	first minimum at $x = 4$, i.e., $v[3][0]+f_4(4)+hc_4 \cdot 0 = 110+50+0 = 160$: $r[4][0] \leftarrow 4$
4	1	4	first minimum at $x = 4$, i.e., $v[3][1]+f_4(4)+hc_4 \cdot 1 = 120+50+10 = 180$: $r[4][1] \leftarrow 4$
5	0	0	first minimum at $x = 0$, i.e., $v[4][1]+f_5(0)+hc_5 \cdot 0 = 180+0+0 = 180$: $r[5][0] \leftarrow 0$

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

$r[t][k]$	0	1	2	3	4	5	6	7	8	9	10	11
$t = 0$	0	0	0	0	0	0	0	0	0	0	0	0
$t = 1$	2	3	4	0	0	0	0	0	0	0	·	·
$t = 2$	0	0	3	4	4	4	0	0	0	·	·	·
$t = 3$	3	4	4	4	4	4	·	·	·	·	·	·
$t = 4$	4	4	·	·	·	·	·	·	·	·	·	·
$t = 5$	0	·	·	·	·	·	·	·	·	·	·	·

– Once the matrix r has been computed, an optimal solution of the CLS instance can be computed by applying the following rule for $t = m, m-1, \ldots, 1$, starting from $k = 0$:

$$x_t \leftarrow r[t][k], \qquad k \leftarrow k + d_t - x_t.$$

This yields the vector $(x_1, x_2, \ldots, x_m)$, where each entry x_t is the number of units produced in period t in an optimal solution.

– The following pseudo-code implements the computation of an optimal solution, given the matrix r.

Algorithm

Algorithm: DP_CLS_SolComputation

Input: The number m of periods and the demands $d[1..m]$ of the CLS instance, and the matrix $r[0..m][0..D_{1,m}]$

Output: An optimal solution of the CLS instance

```
1 Function DP_CLS_SolComputation(m, d, r)
2     x := new array[1..m]
3     k := 0
4     for t := m downto 1 do
5         x[t] ← r[t][k]
6         k ← k + d[t] − x[t]
7     end
8     return x
```

Example 11.4: Computing an optimal solution

After the execution of the DP algorithm on the instance of Example 11.1, we have the matrix r of Example 11.3. We can compute an optimal solution as follows:

Iteration	t	k	$r[t][k]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$x[5]$	$k \leftarrow k + d[t] - x[t]$
–	–	–	–	0	0	0	0	0	$t = 5,\ k = 0$
1	5	0	0	0	0	0	0	0	$k = 0 + 1 - 0 = 1$
2	4	1	4	0	0	0	4	0	$k = 1 + 4 - 4 = 1$
3	3	1	4	0	0	4	4	0	$k = 1 + 3 - 4 = 0$
4	2	0	0	0	0	4	4	0	$k = 0 + 1 - 0 = 1$
5	1	1	3	3	0	4	4	0	$k = 1 + 2 - 3 = 0$

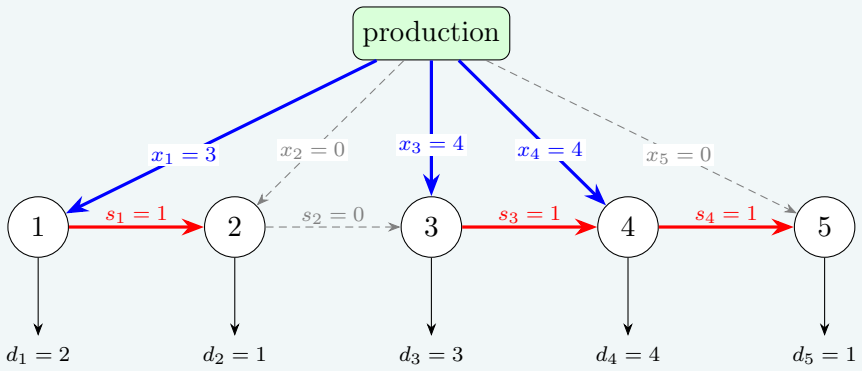
An optimal solution is the vector:

$$(3, 0, 4, 4, 0)$$

The setups are in periods 1, 3, 4, the inventories are $(s_1, \ldots, s_5) = (1, 0, 1, 1, 0)$, and the total cost is (there is no variable production cost)

$$\underbrace{50 + 50 + 50}_{\text{setup}} + \underbrace{10 \cdot 1 + 10 \cdot 0 + 10 \cdot 1 + 10 \cdot 1 + 10 \cdot 0}_{\text{holding}} = 150 + 30 = 180.$$

The corresponding optimal value is $v[5][0] = 180$, the same value given by Gurobi on the ILP_{CLSP} model; the capacity costs $180 - 170$ with respect to the uncapacitated instance. The network of the solution is:



Period 4 produces although one unit is in stock at the end of period 3: the solution does not have the zero-inventory ordering property (and it is the unique optimal solution with integer quantities).

11.2.2 Number of steps

- Let $S(m, D, C)$ denote the number of steps executed by DP_CLS in the worst case, where m is the number of periods, $D = D_{1,m}$ is the total demand, and $C = \max_{t \in T} cap_t$ is the largest capacity. For every $t \geq 1$ we have $R[t] + 1 \leq D$ (since $d_1 \geq 1$), and the inner loop has at most $C + 1$ iterations. In the worst case, the condition at line 16 is always true. The number of executions of the lines is:
  - Line 2, Line 3, Line 7, and Line 8 are executed 1 time each.
  - Line 4 and Line 5 are executed m times each.
  - Line 9 and Line 10 are executed D times each.
  - Line 12 is executed m times.
  - Line 13, Line 14, and Line 20 are executed $\sum_{t=1}^{m}(R[t] + 1) \leq mD$ times each.
  - Line 15, Line 16, and Line 17 are executed at most $mD(C + 1)$ times each.
  - Line 23 is executed 1 time.

  Therefore,

$$S(m, D, C) \leq 5 + 3m + 2D + 3mD + 3mD(C + 1) = 5 + 3m + 2D + 6mD + 3mDC,$$

  and hence

$$S(m, D, C) = O(mDC).$$

The DP algorithm executes

$$S(m, D, C) = O(m\,D\,C)$$

steps in the worst case. The number of steps depends on the values D and C, not only on the number m of periods: the DP algorithm is **pseudo-polynomial**.

- DP_CLS_InfoSol only adds a constant number of operations inside the same loops, and therefore its number of steps has the same order of growth.

- *Example.* Consider the capacitated instance with 6 periods, demands $(50, 30, 10, 10, 40, 10)$, setup cost 40, holding cost 1, and capacity 80. Measured in units of 10, it has 35 states with $t \geq 1$.

- The DP algorithm gives the optimal value 160, with the production plan $(50, 50, 0, 0, 50, 0)$ (in the original units): the same value given by Gurobi on the ILP_{CLSP} model.

11.2.3 The case of constant capacity

- With general capacities cap_t the CLS is NP-hard, so a polynomial algorithm is not expected and the pseudo-polynomial DP algorithm is a natural exact method for small values of D and C.

- When the capacity is the same in all periods, $cap_t = C$ for all $t \in T$, the CLS can be solved in polynomial time, because the zero-inventory ordering property is replaced by a weaker one:

If $cap_t = C$ for all $t \in T$, there is an optimal solution in which, between two consecutive periods with empty inventory, at most one period produces a quantity strictly between 0 and C; all the other periods produce 0 or C.

- The CLS with constant capacity has its own page in the catalogue of problems.

- Example 11.1 has constant capacity $C = 4$. Its optimal solution $(3, 0, 4, 4, 0)$ has inventories $(1, 0, 1, 1, 0)$, empty at the end of periods 0, 2, and 5.

- In periods 1–2 only period 1 produces a quantity strictly between 0 and 4, and in periods 3–5 every period produces 0 or 4, as stated by the property.

11.3 Typical mistakes

Warning

- **Applying the zero-inventory ordering property with capacities.** With capacities a period can produce while units are in stock: in Example 11.1 the unique optimal plan violates the property.

- **Forgetting the base cases with value $+\infty$.** The initial inventory is empty, so $v(0, k) = +\infty$ for $k > 0$; this value marks every state that cannot be reached within the capacities.

- **Allowing too large an inventory.** At the end of period t the inventory is at most $D_{t+1,m}$, since it must be empty at the end of period m: the other pairs (t, k) are not states.

- **Putting the holding cost inside the choice.** The cost $hc_t\, k$ of the final inventory

is paid in the state (t, k) whatever the production x: it is added outside the minimum.

- **Calling $O(mDC)$ polynomial.** The running time contains the total demand D and the largest capacity C, which are numbers of the input: the DP algorithm is pseudo-polynomial.

11.4 In summary

Summary

- With capacities the zero-inventory ordering property fails: a period may produce while units are in stock, and the DP algorithm over the production runs does not apply.
- The state is the pair (t, k) of the period and the final inventory, with $k \leq D_{t+1,m}$; the recurrence $v(t, k)$ chooses the production x of period t, and the optimal value is $v(m, 0)$.
- The matrix r of the chosen productions, read backwards from the state $(m, 0)$, gives an optimal plan.
- The DP algorithm takes $O(mDC)$ steps, which is pseudo-polynomial.
- With a constant capacity a weaker property holds, and the problem is polynomial; with general capacities it is NP-hard.

Dashboard of the example.

Instance	Lower bound	Upper bound	Origin
Example 11.1 (capacitated, $C = 4$)	180	180	DP algorithm on (period, inventory)

Miscellaneous problems

Chapter 12

DP for the coin change problem

Prerequisites

- From this module, the chapter «DP for the unbounded knapsack problem»: unlimited supply of every item type, 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 (Chapter 2) and the size of the input (Chapter 11).

Objectives

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

- explain why the CCP asks for an exact amount and why an instance can be infeasible;
- write the recurrence $v(k)$ with the value $+\infty$ for the amounts that cannot be returned;
- run the DP algorithm by hand, following the comparisons with the running best value m;
- reconstruct an optimal solution as the vector $(x_1, \ldots, x_n)$ when the instance is feasible;
- count the steps of the DP algorithm.

The problem of this chapter is the coin change problem (CCP): its definition is on its page.

12.1 Instances and feasible solutions

Example 12.1: CCP instance

We consider an amount of change $c = 10$ to return, and $n = 3$ denominations:

$$w = (2, 3, 5).$$

We refer to this CCP instance as Example 12.1.

Unlike the 0/1 knapsack problem (KP), where the total weight of the selected items must not exceed the capacity, the CCP requires the total value of the selected coins to be *exactly equal* to c. Moreover, since every denomination is available in unlimited supply, the number of coins of a given denomination used in a solution is not bounded a priori.

12.1.1 Encoding of feasible solutions

A *feasible solution* of the CCP is a choice of the number of coins of each denomination whose total value is exactly equal to the amount of change c.

- A CCP feasible solution can be described by the integer vector

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

  where each component has the following meaning:

  $$x_j = \text{number of coins of denomination } j \text{ used}, \qquad \forall j \in \{1, 2, \ldots, n\}, \quad x_j \in \mathbb{Z}_{\geq 0}.$$

  Since the solution is feasible, the total value of the selected coins must be exactly equal to c:

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

  The total number of coins of a feasible solution is

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

- Since $\sum_{j=1}^{n} w_j x_j = c$, every component x_j of a feasible solution satisfies $x_j \leq \lfloor c/w_j \rfloor$. Therefore, the number of feasible solutions is finite, but it grows quickly with c and n, and enumerating all of them to find the one using the fewest coins quickly becomes impractical.

12.2 A 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 CCP subinstance given by:

1. an integer amount of change k to return,
2. the same set $\{1, 2, \ldots, n\}$ of denominations, each available in unlimited supply.

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

$$v(k) = \begin{cases} 0 & \text{if } k = 0, \\ +\infty & \text{if } k \geq 1 \text{ and } w_j > k \text{ for every } j \in \{1, 2, \ldots, n\}, \\ \min\limits_{j\,:\,w_j \leq k} \{1 + v(k - w_j)\} & \text{if } k \geq 1 \text{ and } w_j \leq k \text{ for some } j \in \{1, 2, \ldots, n\}. \end{cases} \tag{12.1}$$

  - The recursive decomposition process of formula (12.1) generates at most n **independent subproblems**, corresponding to the possible choices of a denomination j, with $w_j \leq k$, used as one of the coins returning the amount of change k.
  - The base case is $v(0) = 0$ since no coin is needed to return a null amount of change, and $v(k) = +\infty$ if no denomination is small enough to be used, since in this case the amount of change k cannot be returned at all.
  - The optimal value $v(k)$ is obtained by trying every denomination j with $w_j \leq k$ and selecting the one that minimizes the sum $1 + v(k - w_j)$ of the single coin of denomination j and the optimal value of the residual subproblem, returning the remaining amount of change $k - w_j$.
  - Hence, an optimal solution for the subinstance with amount of change k is built by combining the chosen coin of denomination j with an optimal solution of the corresponding residual subproblem, returning the amount of change $k - w_j$.

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

  - Different choices of the denomination used as the first coin, possibly made for different values of k, can lead to the same residual amount of change, and therefore the same value $v(k)$ may need to be computed several times during the recursive evaluation of formula (12.1).

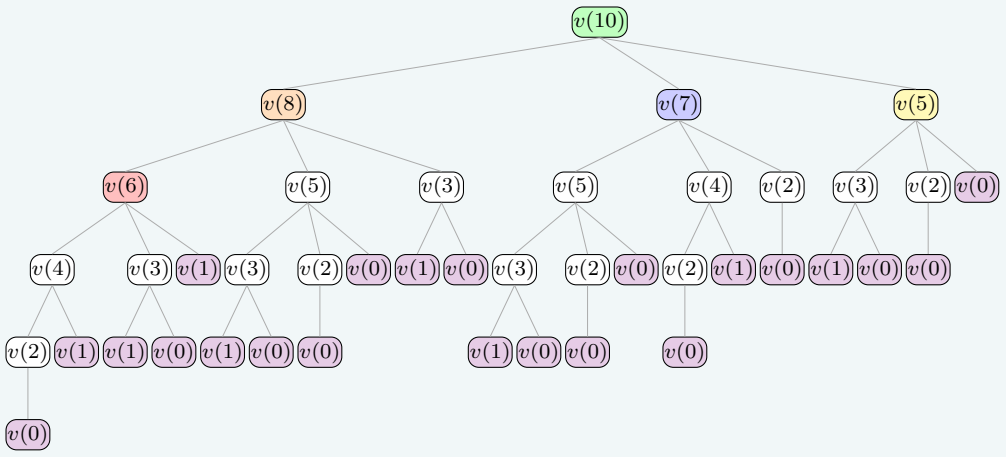
    $\rightsquigarrow$ **Overlapping Subproblems verified!**

– A **first method** to compute the optimal value $v(c)$ of the CCP instance is the **top-down approach**. The method starts from the call with $k = c$ and evaluates formula (12.1) recursively: each call tries every denomination j with $w_j \leq k$, until the amount of change is equal to 0 or no denomination is small enough to be used.

– 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 12.2: Recursion tree

We consider the CCP instance of Example 12.1. This is the recursion tree of the top-down approach to compute $v(10)$:



The function v is called 42 times, and the same subproblems are solved multiple times: $v(0)$ is called 14 times, $v(1)$ is called 8 times, $v(2)$ is called 6 times, $v(3)$ is called 5 times, $v(4)$ is called 2 times, and $v(5)$ is called 3 times. All the other calls are made only once.

– The value $v(k)$ can be computed once the values $v(k')$, for every smaller amount of change $k' < k$, are known. Therefore, all values $v(k)$ can be computed by considering the amount of change k in increasing order.

Example 12.3: Computing the values of the v function

We consider the CCP instance of Example 12.1, with amount of change $c = 10$ to return and denominations $w = (2, 3, 5)$.
The values of the function v are computed as follows:

* The base case gives $v(0) = 0$.
* Once we have computed $v(0)$, we can compute $v(1)$: since no denomination satisfies $w_j \leq 1$, we obtain
$$v(1) = +\infty.$$

* Once we have computed $v(0)$, we can compute $v(2)$:

$$v(2) = 1 + v(0) = 1 + 0 = 1.$$

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

$$v(3) = \min\{1 + v(1),\ 1 + v(0)\} = \min\{1 + \infty,\ 1 + 0\} = 1.$$

* Once we have computed $v(1)$ and $v(2)$, we can compute $v(4)$, trying every denomination j with $w_j \leq 4$:

$$v(4) = \min\{1 + v(2),\ 1 + v(1)\} = \min\{1 + 1,\ 1 + \infty\} = 2.$$

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

$$v(5) = \min\{1 + v(3),\ 1 + v(2),\ 1 + v(0)\} = \min\{1 + 1,\ 1 + 1,\ 1 + 0\} = 1.$$

* Once we have computed $v(1)$, $v(3)$, and $v(4)$, we can compute $v(6)$:

$$v(6) = \min\{1 + v(4),\ 1 + v(3),\ 1 + v(1)\} = \min\{1 + 2,\ 1 + 1,\ 1 + \infty\} = 2.$$

* Once we have computed $v(2)$, $v(4)$, and $v(5)$, we can compute $v(7)$, trying every denomination j with $w_j \leq 7$:

$$v(7) = \min\{1 + v(5),\ 1 + v(4),\ 1 + v(2)\} = \min\{1 + 1,\ 1 + 2,\ 1 + 1\} = 2.$$

  Here, the denominations $w_1 = 2$ and $w_3 = 5$ are tied: both give value 2. Since the bottom-up algorithm only updates the optimal value when a strictly smaller value is found, the first denomination achieving the minimum, $w_1 = 2$, is the one recorded.

* Once we have computed $v(3)$, $v(5)$, and $v(6)$, we can compute $v(8)$:

$$v(8) = \min\{1 + v(6),\ 1 + v(5),\ 1 + v(3)\} = \min\{1 + 2,\ 1 + 1,\ 1 + 1\} = 2.$$

* Finally, once we have computed $v(5)$, $v(7)$, and $v(8)$, we can compute $v(10)$, trying every denomination j with $w_j \leq 10$:

$$v(10) = \min\{1 + v(8),\ 1 + v(7),\ 1 + v(5)\} = \min\{1 + 2,\ 1 + 2,\ 1 + 1\} = 2.$$

– 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 amount of change, we can solve the subproblems in increasing order 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 CCP instance.

– A **second method** to compute the optimal value $v(c)$ of the CCP instance is the **bottom-up approach**. The method starts from the base case and evaluates formula (12.1) for increasing values of $k \in \{1, 2, \ldots, c\}$: each value $v(k)$ is computed using already-computed values, until the value $v(c)$ of the CCP instance is obtained.

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

Algorithm

Algorithm: CCP_DP

Input: A CCP instance given by n, the array $w[1..n]$ of denominations, and the amount of change c

Output: The optimal value

```
1  Function CCP_DP(n, w, c)
2      v := new array[0..c]
3      v[0] ← 0
4      for k := 1 to c do
5          m := +∞
6          for j := 1 to n do
7              if w[j] ≤ k and 1 + v[k − w[j]] < m then
8                  m ← 1 + v[k − w[j]]
9              end
10         end
11         v[k] ← m
12     end
13     return v[c]
```

* The array $v[0..c]$ is the **lookup table**: it stores the optimal values of all subproblems.
* The instruction $v[0] \leftarrow 0$ initializes the base case, corresponding to a null amount of change.
* The outer loop, over $k \in \{1, 2, \ldots, c\}$, computes the entries of the array in increasing order of amount of change.
* The inner loop, over $j \in \{1, 2, \ldots, n\}$, tries every denomination and keeps in m the smallest value found so far; m is initialized to $+\infty$, so that $v[k]$ remains $+\infty$ if no denomination j with $w[j] \leq k$ is found.
* At the end of the inner loop, $v[k] \leftarrow m$ stores the optimal value $v[k]$.
* Finally, the optimal value of the CCP instance is returned as $v[c]$.

Example 12.4: Execution of the DP algorithm

We consider the CCP instance of Example 12.1, with amount of change $c = 10$ to return and denominations $w = (2, 3, 5)$.

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

k	j	$v[k]$	m	Action
–	–	0	–	$v[0] \leftarrow 0$
	–	–	$+\infty$	$m := +\infty$
1	1	–	$+\infty$	$w[1] = 2 > 1$: skip
	2	–	$+\infty$	$w[2] = 3 > 1$: skip
	3	–	$+\infty$	$w[3] = 5 > 1$: skip
	–	$+\infty$	$+\infty$	$v[1] \leftarrow m = +\infty$
	–	–	$+\infty$	$m := +\infty$
2	1	–	1	$1+v[0] = 1 < m$: $m \leftarrow 1$
	2	–	1	$w[2] = 3 > 2$: skip
	3	–	1	$w[3] = 5 > 2$: skip
	–	1	1	$v[2] \leftarrow m = 1$
	–	–	$+\infty$	$m := +\infty$
3	1	–	$+\infty$	$1+v[1] = +\infty \not< m = +\infty$: no update
	2	–	1	$1+v[0] = 1 < m$: $m \leftarrow 1$
	3	–	1	$w[3] = 5 > 3$: skip
	–	1	1	$v[3] \leftarrow m = 1$
	–	–	$+\infty$	$m := +\infty$
4	1	–	2	$1+v[2] = 2 < m$: $m \leftarrow 2$
	2	–	2	$1+v[1] = +\infty \not< m = 2$: no update
	3	–	2	$w[3] = 5 > 4$: skip
	–	2	2	$v[4] \leftarrow m = 2$
	–	–	$+\infty$	$m := +\infty$
5	1	–	2	$1+v[3] = 2 < m$: $m \leftarrow 2$
	2	–	2	$1+v[2] = 2 \not< m = 2$: no update
	3	–	1	$1+v[0] = 1 < m$: $m \leftarrow 1$
	–	1	1	$v[5] \leftarrow m = 1$
	–	–	$+\infty$	$m := +\infty$
6	1	–	3	$1+v[4] = 3 < m$: $m \leftarrow 3$
	2	–	2	$1+v[3] = 2 < m$: $m \leftarrow 2$
	3	–	2	$1+v[1] = +\infty \not< m = 2$: no update
	–	2	2	$v[6] \leftarrow m = 2$
	–	–	$+\infty$	$m := +\infty$
7	1	–	2	$1+v[5] = 2 < m$: $m \leftarrow 2$
	2	–	2	$1+v[4] = 3 \not< m = 2$: no update
	3	–	2	$1+v[2] = 2 \not< m = 2$: no update
	–	2	2	$v[7] \leftarrow m = 2$
	–	–	$+\infty$	$m := +\infty$
8	1	–	3	$1+v[6] = 3 < m$: $m \leftarrow 3$
	2	–	2	$1+v[5] = 2 < m$: $m \leftarrow 2$
	3	–	2	$1+v[3] = 2 \not< m = 2$: no update
	–	2	2	$v[8] \leftarrow m = 2$
	–	–	$+\infty$	$m := +\infty$
9	1	–	3	$1+v[7] = 3 < m$: $m \leftarrow 3$
	2	–	3	$1+v[6] = 3 \not< m = 3$: no update
	3	–	3	$1+v[4] = 3 \not< m = 3$: no update
	–	3	3	$v[9] \leftarrow m = 3$
	–	–	$+\infty$	$m := +\infty$
10	1			$: m \leftarrow 3$
	2			$= 3$: no update
	3			$: m \leftarrow 2$
	–	2	2	$v[10] \leftarrow m = 2$

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

k	0	1	2	3	4	5	6	7	8	9	10
$v[k]$	0	$+\infty$	1	1	2	1	2	2	2	3	2

- After the execution of the DP algorithm, the array v contains the optimal value $v[k]$ for every amount of change $k \in \{0, 1, \ldots, c\}$, where $v[c]$ is the minimum number of coins needed to return the amount of change c.

- Step 2: computation of an optimal solution
  - To compute an optimal solution, i.e., the integer vector $(x_1, x_2, \ldots, x_n)$, it is necessary to store some additional information.
  - For each amount of change k, in addition to the optimal value $v[k]$, we store $s[k]$, the index of the denomination of the coin used to obtain the optimal value $v[k]$ in formula (12.1).
  - More precisely, $s[k]$ is equal to the index j of the denomination w_j that minimizes the right-hand side of formula (12.1) when computing $v[k]$.

  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

Algorithm: CCP_DP_InfoSol

Input: A CCP instance given by n, the array $w[1..n]$ of denominations, and the amount of change c

Output: The optimal value and the array $s[0..c]$ of the indices of the denominations used

```
1  Function CCP_DP_InfoSol(n, w, c)
2      v := new array[0..c]; s := new array[0..c]
3      v[0] ← 0; s[0] ← 0
4      for k := 1 to c do
5          m := +∞; t := 0
6          for j := 1 to n do
7              if w[j] ≤ k and 1 + v[k − w[j]] < m then
8                  m ← 1 + v[k − w[j]]; t ← j
9              end
10         end
11         v[k] ← m; s[k] ← t
12     end
13     return v[c], s
```

* The DP algorithm uses the additional temporary variable t to keep track of the decision made when m is updated. If the denomination $w[j]$ is used, $t \leftarrow j$ stores its index. At the end of the inner loop, $s[k] \leftarrow t$ stores the index of the denomination of the coin used, together with the optimal value $v[k] \leftarrow m$.

Example 12.5: Computation of the additional information

We consider the CCP instance of Example 12.1, with amount of change $c = 10$ to return and denominations $w = (2, 3, 5)$.

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

k	0	1	2	3	4	5	6	7	8	9	10
$s[k]$	0	0	1	2	1	3	2	1	2	1	3

For example, $s[5] = 3$ since formula (12.1) selects the denomination $w_3 = 5$ when computing $v[5] = 1$, and $s[10] = 3$ since the denomination $w_3 = 5$ is also selected when computing $v[10] = 2$. The entry $s[1]$ is left undefined (0, by convention), since $v[1] = +\infty$ is infeasible.

– Once the array s has been computed, and provided that $v[c] < +\infty$, an optimal solution can be reconstructed by repeatedly using one coin of denomination $s[k]$ and moving to the residual amount of change $k - w_{s[k]}$, starting from $k = c$ and $x_j = 0$ for all $j \in \{1, 2, \ldots, n\}$, until $k = 0$:

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

This yields the vector $(x_1, x_2, \ldots, x_n)$, where each entry x_j is the number of coins of denomination j in an optimal solution.

– The condition $v[c] < +\infty$ is necessary: if $v[c] = +\infty$ the instance is infeasible, $s[c] = 0$ and there is no solution to reconstruct. If $v[c] < +\infty$, then $s[k] \geq 1$ for every amount of change k visited by the procedure.

Algorithm

Algorithm: CCP_Reconstruct

Input: The number n of denominations, the array $w[1..n]$, the amount of change c with $v[c] < +\infty$, and the array $s[0..c]$

Output: An optimal solution

```
1  Function CCP_Reconstruct(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 do
8          x[s[k]] ← x[s[k]] + 1
9          k ← k − w[s[k]]
10     end
11     return x
```

Example 12.6: Computing an optimal solution

After the execution of the DP algorithm on the instance of Example 12.1, we have the following array s:

k	0	1	2	3	4	5	6	7	8	9	10
$s[k]$	0	0	1	2	1	3	2	1	2	1	3

We can reconstruct an optimal solution by starting from $k = 10$ as follows:

Iteration	k	$s[k]$	$x[1]$	$x[2]$	$x[3]$	$k \leftarrow k - w_{s[k]}$
–	–	–	0	0	0	$k = 10$
1	10	3	0	0	1	$k = 10 - 5 = 5$
2	5	3	0	0	2	$k = 5 - 5 = 0$

Thus, the vector

$$(0, 0, 2)$$

is an optimal solution: it uses two coins of denomination $w_3 = 5$. The corresponding total value is $2 \cdot 5 = 10 = c$. The corresponding optimal value is $v(10) = 2$.

12.2.1 Number of steps

- Let $S(n, c)$ denote the number of steps executed by `CCP_DP` in the worst case, where n is the number of denominations and c is the amount of change. We compute $S(n, c)$ by counting how many times the lines of the pseudo-code of `CCP_DP` are executed.

- In the worst case, the condition at line 7 is true at every iteration of the inner loop, and therefore line 8 is executed as often as line 7. 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 11 is executed c times.
  - Line 13 is executed 1 time.

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

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

  Thus,

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

  Hence,

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

  Indeed,

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

  The DP algorithm executes

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

  steps in the worst case.

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

12.3 Typical mistakes

Warning

- **Using 0 instead of $+\infty$.** An amount that cannot be returned has $v(k) = +\infty$; with the value 0 the DP algorithm would claim that no coin is needed.
- **Reconstructing an infeasible instance.** If $v[c] = +\infty$ then $s[c] = 0$, and the reconstruction would read $x[0]$ and $w[0]$, outside the arrays: it must first check that $v[c] < +\infty$.
- **Storing the value of the coin instead of its index.** The vector x is indexed by the denominations: $s[k]$ stores the index j, and the coin value is $w_{s[k]}$.
- **Confusing the CCP with the UKP.** The CCP asks for exactly c and minimizes the number of coins; the UKP asks for at most c and maximizes the profit.

12.4 In summary

Summary

- The CCP asks for the minimum number of coins whose total value is exactly c; the instance is infeasible when $v(c) = +\infty$.
- The recurrence $v(k) = \min_{j:\, w_j \le k}\{1 + v(k - w_j)\}$ is computed bottom-up in $\Theta(nc)$ steps, which is pseudo-polynomial.
- An optimal solution is read backwards from $s[c]$, the index of the denomination of the first coin chosen for the amount c, and only if $v[c] < +\infty$.

Dashboard of the CCP example. Example 12.1 ($c = 10$, $w = (2, 3, 5)$):

Method	Lower bound	Upper bound	Origin
DP algorithm	2	2	minimum number of coins, solution $(0, 0, 2)$

Chapter 13

DP for the rod cutting problem

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 this module, the chapter «DP for the 0/1 knapsack problem»: the pairs (item, capacity) and the reconstruction of a solution, which return here.
- From the module *Algorithms*: how to read pseudocode (Chapter 2) and how to count the steps of an algorithm (Chapter 12 «Running time»).

Objectives

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

- state the RCP, encode a solution as a vector of numbers of pieces, and explain why cutting patterns and vectors are different objects;
- write the recurrence $v(\ell)$ and explain it, base case included;
- run the DP algorithm by hand with the top-down and the bottom-up approach, and with the running best value m;
- reconstruct an optimal solution from the vector s, for the rod and for every shorter rod;
- write the second recurrence $v(j, \ell)$ on pairs (largest piece length, rod length) and count the steps of the DP algorithm.

The problem of this chapter is the rod cutting problem (RCP): its definition is on its page.

13.1 Instances and feasible solutions

Example 13.1: Rod cutting problem

We consider a rod of length $n = 4$ and a set $\{1, 2, 3, 4\}$ of 4 possible piece lengths with the following profits:

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
p_ℓ	1	5	8	9

We refer to this RCP instance as Example 13.1.

The number of possible solutions of the RCP is 2^{n-1} since we can cut a rod of length n in 2^{n-1} different ways, with an independent option of cutting (or not) at each possible cutting position.

- The number of possible RCP solutions grows exponentially with the length n of the rod:

$$2^{n-1} = O(2^n).$$

Therefore, enumerating all possible solutions quickly becomes impractical as n increases.

13.1.1 Encoding of feasible solutions

A *feasible solution* of the RCP is a way of cutting the rod into pieces, such that the total length of the pieces is equal to the rod length.

- An RCP feasible solution can be described by an integer solution vector

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

where each component has the following meaning:

$$x_\ell = \text{number of pieces of length } \ell, \qquad \forall \ell \in \{1, 2, \ldots, n\};$$

The number of pieces is

$$\sum_{\ell=1}^{n} x_\ell.$$

Since the solution is feasible, the total length of the pieces must be equal to the rod length:

$$\sum_{\ell=1}^{n} \ell\, x_\ell = n.$$

The total profit of a feasible solution is

$$\sum_{\ell=1}^{n} p_\ell x_\ell.$$

- Two or more cutting patterns may have the same vector representation: the vector records how many pieces of each length are obtained, but it does not specify the order in which the pieces are cut.

- Cutting patterns with the same vector representation have the same total profit, since the order in which the pieces are cut does not affect the obtained pieces and therefore does not affect the total profit.

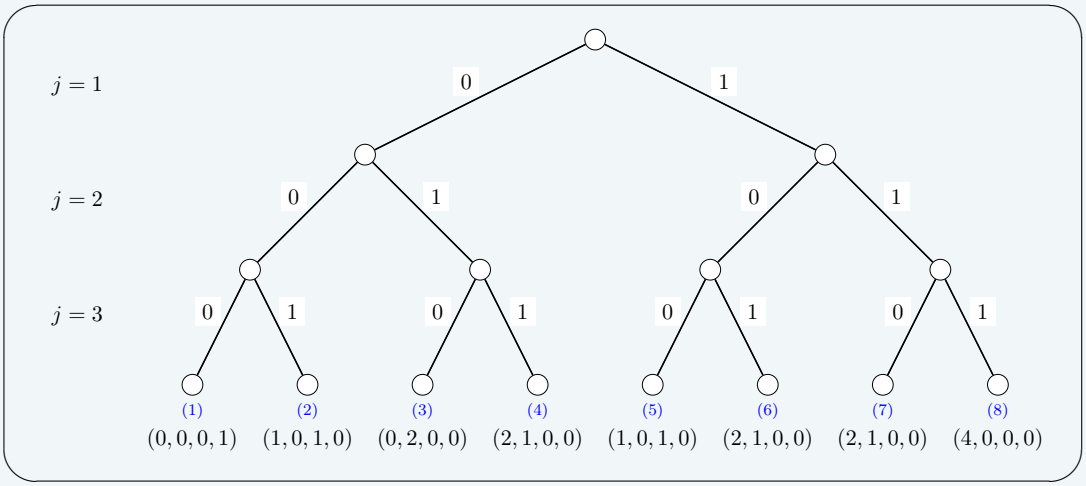
Example 13.2: Enumeration of all possible RCP solutions

We consider the RCP instance of Example 13.1 with a rod of length $n = 4$ and the profits associated with the possible piece lengths are:

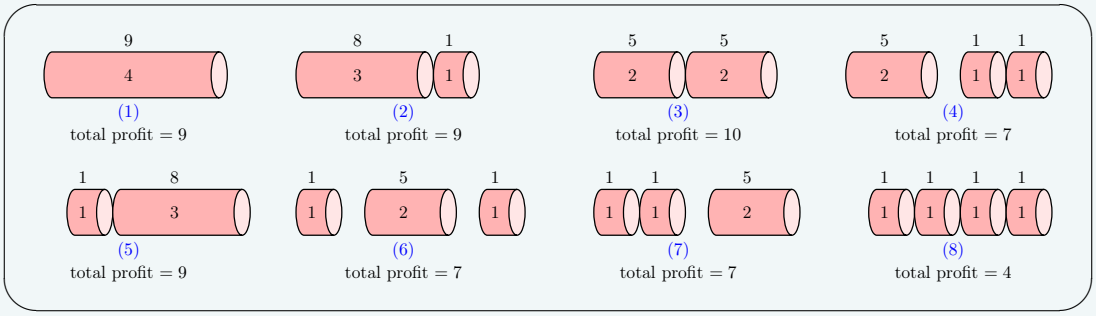
	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
p_ℓ	1	5	8	9

There are $2^{4-1} = 8$ feasible solutions, corresponding to the different ways of cutting the rod.

In the following binary tree, each level corresponds to a possible cutting position: an edge labeled 1 means that the rod is cut at that length, while an edge labeled 0 means that it is not cut at that length. Each leaf represents one complete cutting pattern and reports the corresponding vector.



The figure below shows all possible cutting patterns associated with the leaves of the previous tree, through the labels $(1), (2), \ldots, (8)$. For each obtained piece, the number written inside the piece is its length, while the number above it is its profit.



Label	Solution vector	Number of pieces	Total profit
(1)	$(0,0,0,1)$	1	9
(2)	$(1,0,1,0)$	2	$8+1=9$
(3)	$(0,2,0,0)$	2	$5+5=10$
(4)	$(2,1,0,0)$	3	$5+1+1=7$
(5)	$(1,0,1,0)$	2	$1+8=9$
(6)	$(2,1,0,0)$	3	$1+5+1=7$
(7)	$(2,1,0,0)$	3	$1+1+5=7$
(8)	$(4,0,0,0)$	4	$1+1+1+1=4$

13.2 A first DP algorithm

- Step 1: computation of the optimal value

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

1. a rod of length ℓ,
2. a set $\{1,2,\ldots,\ell\}$ of ℓ possible piece lengths,
3. for each piece length $j \in \{1,2,\ldots,\ell\}$, a profit p_j.

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

$$v(\ell) = \begin{cases} \max\left\{p_j + v(\ell-j) : j \in \{1,2,\ldots,\ell\}\right\} & \text{if } 1 \le \ell \le n, \\ 0 & \text{if } \ell = 0. \end{cases} \tag{13.1}$$

  - The recursive decomposition process of formula (13.1) generates ℓ **independent subproblems**, each corresponding to a possible choice of cutting a first piece of length j from a rod of length ℓ. Here, $v(0) = 0$ since a rod of length 0 cannot be cut and has profit 0.

– The optimal value $v(\ell)$ is obtained by trying all possible first-piece lengths and selecting the one that maximizes the sum $p_j + v(\ell - j)$ of the profit of the first piece and the optimal value of the residual subproblem.

– Hence, an optimal solution for a rod of length ℓ is built by combining the chosen first piece of length j with an optimal solution of the residual subproblem of length $\ell - j$.

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

– Different choices of the first-piece length, possibly made for different values of ℓ, can lead to the same residual length, and therefore the same value $v(\ell)$ may need to be computed several times during the recursive evaluation of formula (13.1).

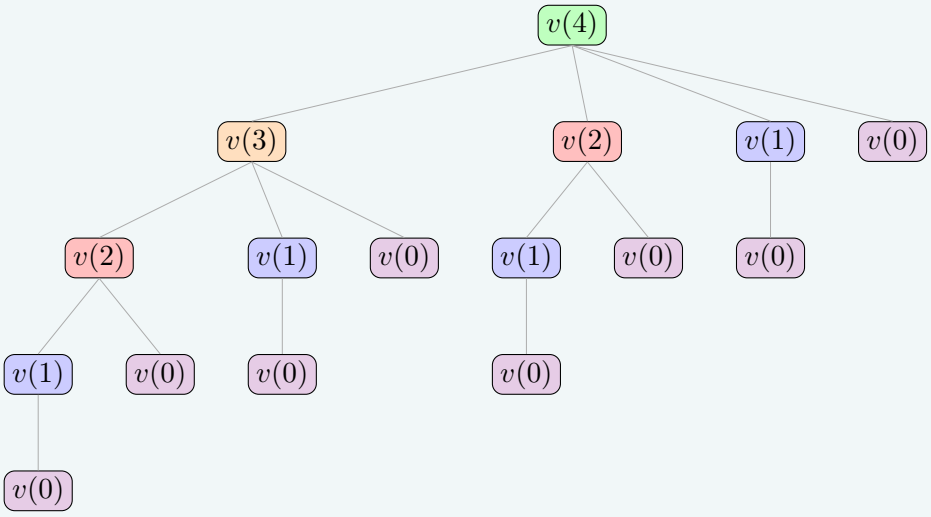
$\rightsquigarrow$ **Overlapping Subproblems verified!**

– A **first method** to compute the optimal value $v(n)$ of the RCP instance is the **top-down approach**. The method starts from the call with $\ell = n$ and evaluates formula (13.1) recursively: each call for a rod of length ℓ triggers further calls for rods of length $\ell - j$, for all $j \in \{1, 2, \ldots, \ell\}$, until the base case $v(0)$ 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 13.3: Recursion tree

We consider the RCP instance of Example 13.1. This is the recursion tree of the top-down approach to compute $v(4)$:



The function v is called 16 times, and the same subproblems are solved multiple times: $v(2)$ is called 2 times, $v(1)$ is called 4 times, and $v(0)$ is called 8 times.

All the other calls are made only once.

– The value $v(\ell)$, for a generic rod length ℓ, can be computed once the values $v(j)$ are known for all smaller lengths $j < \ell$.

Example 13.4: Computing the values of the v function

We consider the RCP instance of Example 13.1 with a rod of length $n = 4$ and the profits associated with the possible piece lengths are:

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
p_ℓ	1	5	8	9

The value of the function v can be computed by starting from the base case $v(0) = 0$ and then computing the values of $v(1)$, $v(2)$, $v(3)$, and $v(4)$ in order, as follows:

* Since we know the value of $v(0)$, we can compute $v(1)$, as follows:

$$v(1) = p_1 + v(0) = 1 + 0 = 1$$

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

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

* Once we have computed $v(2)$ and $v(1)$, we can compute $v(3)$, as follows:

$$v(3) = \max\{p_1 + v(2), p_2 + v(1), p_3 + v(0)\} = \max\{1 + 5, 5 + 1, 8 + 0\} = 8$$

* Once we have computed $v(3)$, $v(2)$ and $v(1)$, we can compute $v(4)$, as follows:

$$\begin{aligned} v(4) &= \max\{p_1 + v(3), p_2 + v(2), p_3 + v(1), p_4 + v(0)\} \\ &= \max\{1 + 8, 5 + 5, 8 + 1, 9 + 0\} = 10 \end{aligned}$$

– 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 $v(\ell)$ depends only on subproblems with values $v(j)$, with $j < \ell$, we can solve the subproblems for increasing values of ℓ, starting from the base case $v(0) = 0$ and building up to $v(n)$: this observation naturally leads to a **much more efficient method** to compute the optimal value $v(n)$ of the RCP instance.

– A **second method** to compute the optimal value $v(n)$ of the RCP instance is the **bottom-up approach**. The method starts from the base case $v(0) = 0$ and evaluates formula (13.1) for increasing rod lengths $\ell \in \{1, 2, \ldots, n\}$: each value $v(\ell)$ is

computed using the already-computed values $v(0), v(1), \ldots, v(\ell-1)$, until the value $v(n)$ of the RCP instance is obtained.

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

Algorithm

Algorithm: DP1

Input: An RCP instance given by the length n of the rod and an array $p[1..n]$ of piece profits

Output: The optimal value of the RCP instance

```
1  Function DP1(n, p)
2      v := new array[0..n]
3      v[0] ← 0
4      for ℓ := 1 to n do
5          m := 0
6          for j := 1 to ℓ do
7              if m < p[j] + v[ℓ − j] then
8                  m ← p[j] + v[ℓ − j]
9              end
10         end
11         v[ℓ] ← m
12     end
13     return v[n]
```

* The vector $v[0..n]$ is the **lookup table**: it stores the optimal values of all sub-problems, with the base case $v[0] = 0$.
* The outer loop, over $\ell \in \{1, 2, \ldots, n\}$, computes $v[\ell]$ in increasing order of rod length.
* The inner loop, over $j \in \{1, 2, \ldots, \ell\}$, evaluates all possible first-piece lengths and keeps in m the largest value $p[j] + v[\ell - j]$.
* At the end of the inner loop, $v[\ell] \leftarrow m$ stores the optimal value $v[\ell]$.
* Finally, the optimal value $v[n]$ of the RCP instance is returned.

Example 13.5: Execution of the DP algorithm

We consider the RCP instance of Example 13.1 with a rod of length $n = 4$ and the profits associated with the possible piece lengths are:

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
p_ℓ	1	5	8	9

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

ℓ	j	$v[\ell]$	m	Action
–	–	0	–	$v[0] \leftarrow 0$
	–	–	0	$m \leftarrow 0$
1	1	–	1	$p[1]+v[0] = 1+0 = 1 > 0$: $m \leftarrow 1$
	–	1	1	$v[1] \leftarrow m = 1$
	–	–	0	$m \leftarrow 0$
2	1	–	2	$p[1]+v[1] = 1+1 = 2 > 0$: $m \leftarrow 2$
	2	–	5	$p[2]+v[0] = 5+0 = 5 > 2$: $m \leftarrow 5$
	–	5	5	$v[2] \leftarrow m = 5$
	–	–	0	$m \leftarrow 0$
3	1	–	6	$p[1]+v[2] = 1+5 = 6 > 0$: $m \leftarrow 6$
	2	–	6	$p[2]+v[1] = 5+1 = 6 = m$: no update
	3	–	8	$p[3]+v[0] = 8+0 = 8 > 6$: $m \leftarrow 8$
	–	8	8	$v[3] \leftarrow m = 8$
	–	–	0	$m \leftarrow 0$
4	1	–	9	$p[1]+v[3] = 1+8 = 9 > 0$: $m \leftarrow 9$
	2	–	10	$p[2]+v[2] = 5+5 = 10 > 9$: $m \leftarrow 10$
	3	–	10	$p[3]+v[1] = 8+1 = 9 < 10$: no update
	4	–	10	$p[4]+v[0] = 9+0 = 9 < 10$: no update
	–	10	10	$v[4] \leftarrow m = 10$

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

	$\ell = 0$	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
$v[\ell]$	0	1	5	8	10

  - After the execution of the DP algorithm, the vector v contains the optimal value $v[\ell]$ for every rod length.

- Step 2: computation of an optimal solution
  - To compute an optimal solution, it is necessary to store some additional information.
  - For each rod length ℓ, in addition to the optimal value $v[\ell]$, we store $s[\ell]$, the size of the first piece to cut off in an optimal solution for a rod of length ℓ.
  - Starting from $\ell = n$, the value $s[\ell]$ gives the length of the first piece to cut; the remaining rod has length $\ell - s[\ell]$, and the same rule can be applied again, and so on, until the residual length reaches 0.

    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

Algorithm: DP1_InfoSol

Input: An RCP instance given by the length n of the rod and an array $p[1..n]$ of piece profits

Output: The optimal value of the RCP instance and the information vector $s[1..n]$

```
1  Function DP1_InfoSol(n, p)
2      v := new array[0..n]
3      s := new array[1..n]
4      v[0] ← 0
5      for ℓ := 1 to n do
6          m := 0
7          t := 0
8          for j := 1 to ℓ do
9              if m < p[j] + v[ℓ − j] then
10                 m ← p[j] + v[ℓ − j]
11                 t ← j
12             end
13         end
14         v[ℓ] ← m
15         s[ℓ] ← t
16     end
17     return v[n], s
```

∗ The DP algorithm uses the additional temporary variable t to keep track of the decision made when m is updated. If a first piece of length j is selected, $t \leftarrow j$ stores this decision. At the end of the inner loop, $s[\ell] \leftarrow t$ stores the length of the selected first piece, together with the optimal value $v[\ell] \leftarrow m$.

Example 13.6: Computation of the additional information

We consider the RCP instance of Example 13.1 with a rod of length $n = 4$ and the profits associated with the possible piece lengths are:

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
p_ℓ	1	5	8	9

For each step of the DP algorithm, the table shows the values of ℓ and j, the cell $s[\ell]$, 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 $s[\ell]$ are highlighted in blue when updated.

ℓ	j	$s[\ell]$	m	t	Action
–	–	–	–	–	Initial state
	–	–	0	0	$m \leftarrow 0,\ t \leftarrow 0$
1	1	–	1	1	$1{+}0 = 1 > 0$: $m \leftarrow 1,\ t \leftarrow 1$
	–	1	1	1	$v[1] \leftarrow m = 1,\ s[1] \leftarrow t = 1$
	–	–	0	0	$m \leftarrow 0,\ t \leftarrow 0$
2	1	–	2	1	$1{+}1 = 2 > 0$: $m \leftarrow 2,\ t \leftarrow 1$
	2	–	5	2	$5{+}0 = 5 > 2$: $m \leftarrow 5,\ t \leftarrow 2$
	–	2	5	2	$v[2] \leftarrow m = 5,\ s[2] \leftarrow t = 2$
	–	–	0	0	$m \leftarrow 0,\ t \leftarrow 0$
3	1	–	6	1	$1{+}5 = 6 > 0$: $m \leftarrow 6,\ t \leftarrow 1$
	2	–	6	1	$5{+}1 = 6 = m$: no update
	3	–	8	3	$8{+}0 = 8 > 6$: $m \leftarrow 8,\ t \leftarrow 3$
	–	3	8	3	$v[3] \leftarrow m = 8,\ s[3] \leftarrow t = 3$
	–	–	0	0	$m \leftarrow 0,\ t \leftarrow 0$
4	1	–	9	1	$1{+}8 = 9 > 0$: $m \leftarrow 9,\ t \leftarrow 1$
	2	–	10	2	$5{+}5 = 10 > 9$: $m \leftarrow 10,\ t \leftarrow 2$
	3	–	10	2	$8{+}1 = 9 < 10$: no update
	4	–	10	2	$9{+}0 = 9 < 10$: no update
	–	2	10	2	$v[4] \leftarrow m = 10,\ s[4] \leftarrow t = 2$

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

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
$s[\ell]$	1	2	3	2

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

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

until $j = 0$. This yields the vector $(x_1, x_2, \ldots, x_n)$, where each entry x_ℓ is the number of pieces of length ℓ in an optimal solution.

– The following pseudo-code implements the computation of an optimal solution, given the vector s.

Algorithm

Algorithm: DP1_SolComputation

Input: The rod length n of the RCP instance and the vector $s[1..n]$
Output: An optimal solution of the RCP instance

```
1  Function DP1_SolComputation(n, s)
2      x := new array[1..n]
3      for ℓ := 1 to n do
4          x[ℓ] ← 0
5      end
6      j := n
7      while j > 0 do
8          x[s[j]] ← x[s[j]] + 1
9          j ← j − s[j]
10     end
11     return x
```

Example 13.7: Computing an optimal solution

After the execution of the DP algorithm on the instance of Example 13.1, we have the following vector s:

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
$s[\ell]$	1	2	3	2

We can compute an optimal solution for a rod of length $n = 4$ as follows:

Iteration	j	$s[j]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$j \leftarrow j - s[j]$
–	–	–	0	0	0	0	$j = 4$
1	4	2	0	1	0	0	$j = 4 - 2 = 2$
2	2	2	0	2	0	0	$j = 2 - 2 = 0$

An optimal solution is the vector:

$$(0, 2, 0, 0)$$

– The same computation procedure, based on the vector s, can be applied starting from any rod length ℓ.

Example 13.8: Computing optimal solutions for rod lengths 3 and 2

After the execution of the DP algorithm on the instance of Example 13.1, we have the following vector s:

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
$s[\ell]$	1	2	3	2

We can compute an optimal solution for a rod of length 3 as follows:

Iteration	j	$s[j]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$j \leftarrow j - s[j]$
–	–	–	0	0	0	0	$j = 3$
1	3	3	0	0	1	0	$j = 3 - 3 = 0$

An optimal solution is the vector:

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

We can compute an optimal solution for a rod of length 2 as follows:

Iteration	j	$s[j]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	$j \leftarrow j - s[j]$
–	–	–	0	0	0	0	$j = 2$
1	2	2	0	1	0	0	$j = 2 - 2 = 0$

An optimal solution is the vector:

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

13.2.1 Number of steps

- Let $S(n)$ denote the number of steps executed by `DP1` in the worst case, where n is the rod length. We compute $S(n)$ by counting how many times Line 2–Line 13 are executed.

- In the worst case, the condition at line 7 is true at every iteration of the inner loop, and therefore line 8 is executed as often as line 7. 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 n times.
  - Line 5 is executed n times.
  - Line 6 is executed $\sum_{\ell=1}^{n} \ell$ times.
  - Line 7 is executed $\sum_{\ell=1}^{n} \ell$ times.
  - Line 8 is executed $\sum_{\ell=1}^{n} \ell$ times.

- Line 11 is executed n times.
- Line 13 is executed 1 time.

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

$$1+1+n+n+\sum_{\ell=1}^{n}\ell+\sum_{\ell=1}^{n}\ell+\sum_{\ell=1}^{n}\ell+n+1=3+3n+3\sum_{\ell=1}^{n}\ell.$$

Using $\sum_{\ell=1}^{n}\ell=\dfrac{n(n+1)}{2}$, we obtain

$$S(n)=3+3n+3\cdot\frac{n(n+1)}{2}=\frac{3n^2+9n+6}{2}.$$

Hence,

$$S(n)=\Theta(n^2).$$

Indeed,

$$\lim_{n\to\infty}\frac{\frac{3n^2+9n+6}{2}}{n^2}=\lim_{n\to\infty}\frac{1}{2}\left(3+\frac{9}{n}+\frac{6}{n^2}\right)=\frac{3}{2}\in\mathbb{R}_{>0}.$$

The DP algorithm executes

$$S(n)=\Theta(n^2)$$

steps in the worst case.

13.3 A second DP algorithm

- Step 1: computation of the optimal value

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

1. a rod of length ℓ,
2. a set $\{1,2,\ldots,j\}$ of j possible piece lengths,
3. for each piece length $k\in\{1,2,\ldots,j\}$, a profit p_k.

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

$$v(j,\ell)=\begin{cases}\max\left\{v(j-1,\ell),\ p_j+v(j,\ell-j)\right\} & \text{if } 1\le j\le\ell,\\ v(j-1,\ell) & \text{if } 1\le\ell<j,\\ 0 & \text{if } \ell=0,\\ -\infty & \text{if } j=0 \text{ and } \ell>0.\end{cases}\tag{13.2}$$

– The recursive decomposition process of formula (13.2) generates at most 2 **independent subproblems**, corresponding to whether or not to cut a piece of length j from a rod of length ℓ. If $j > \ell$, a piece of length j cannot be cut from the rod, and therefore the only feasible option is not to cut it.

– Here, $v(j, 0) = 0$ since a rod of length 0 cannot be cut and has profit 0.

– Moreover, $v(0, \ell) = -\infty$ for $\ell > 0$: in this case no positive piece length is available, so no feasible solution exists for a rod of positive length. The value $-\infty$ is used to rule out this infeasible case.

– If $1 \leq j \leq \ell$, the optimal value $v(j, \ell)$ is obtained by trying the two possible options, cutting or not cutting a first piece of length j, and selecting the one that maximizes the profit. The corresponding values are $p_j + v(j, \ell - j)$ and $v(j - 1, \ell)$, respectively. If $1 \leq \ell < j$, piece length j cannot be used, and therefore $v(j, \ell) = v(j - 1, \ell)$.

– Hence, an optimal solution for a rod of length ℓ with largest available piece length j is built by combining the chosen decision on piece length j with an optimal solution of the corresponding residual subproblem.

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

– Different sequences of decisions on the piece lengths, possibly made for different values of j and ℓ, can lead to the same subproblem, and therefore the same value $v(j, \ell)$ may need to be computed several times during the recursive evaluation of formula (13.2).

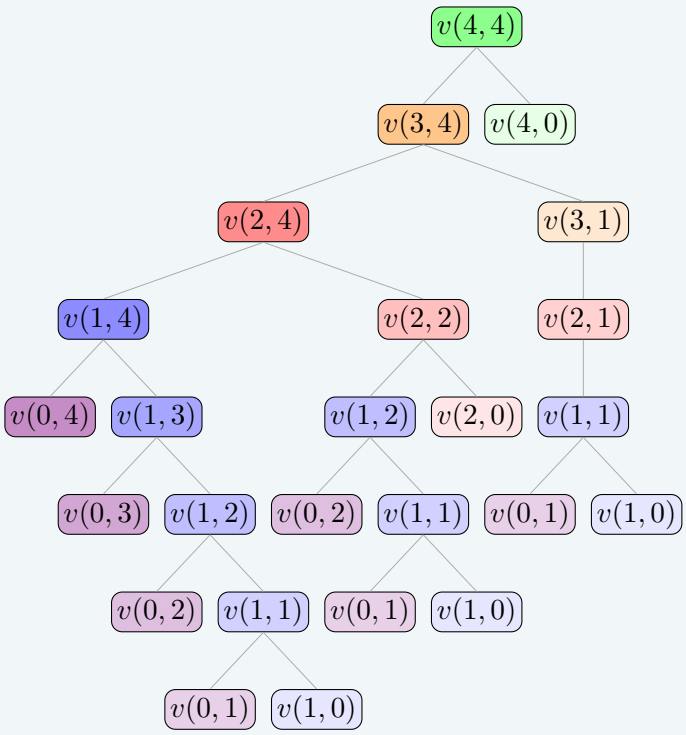
  $\rightsquigarrow$ **Overlapping Subproblems verified!**

– A **first method** to compute the optimal value $v(n, n)$ of the RCP instance is the **top-down approach**. The method starts from the call with $(j, \ell) = (n, n)$ and evaluates formula (13.2) recursively: each call for a rod of length ℓ and largest available piece length j triggers further calls corresponding to the feasible decisions on piece length j, until one of the base cases $v(j, 0)$ or $v(0, \ell)$ 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 13.9: Recursion tree

We consider the RCP instance of Example 13.1. This is the recursion tree of the top-down approach to compute $v(4, 4)$:



The function v is called 25 times, and the same subproblems are solved multiple times: $v(1,2)$ is called 2 times, $v(0,2)$ is called 2 times, $v(1,1)$ is called 3 times, $v(0,1)$ is called 3 times, and $v(1,0)$ is called 3 times. All the other calls are made only once.

– The value $v(j,\ell)$ can be computed once $v(j-1,\ell)$ is known and, when $j \leq \ell$, $v(j,\ell-j)$ is also known. Therefore, for a fixed value of j, all values $v(j,\ell)$ can be computed by considering the values of ℓ in increasing order.

– Starting from the base cases $v(j,0) = 0$ and $v(0,\ell) = -\infty$, the values of the function v can be computed row by row, moving to the next value of j only after all values $v(j,\ell)$ for the current value of j have been computed.

Example 13.10: Computing the values of the v function

We consider the RCP instance of Example 13.1 with a rod of length $n = 4$ and the profits associated with the possible piece lengths are:

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
p_ℓ	1	5	8	9

The values of the function v are computed as follows:

∗ For $j = 1$, only pieces of length 1 can be used. Therefore:

$$v(1,0) = 0,$$

$$v(1,1) = \max\{v(0,1), p_1 + v(1,0)\} = \max\{-\infty, 1+0\} = 1,$$

$$v(1,2) = \max\{v(0,2), p_1 + v(1,1)\} = \max\{-\infty, 1+1\} = 2,$$
$$v(1,3) = \max\{v(0,3), p_1 + v(1,2)\} = \max\{-\infty, 1+2\} = 3,$$
$$v(1,4) = \max\{v(0,4), p_1 + v(1,3)\} = \max\{-\infty, 1+3\} = 4.$$

* For $j = 2$, pieces of length 1 and 2 can be used. We obtain:

$$v(2,0) = 0, \qquad v(2,1) = v(1,1) = 1,$$
$$v(2,2) = \max\{v(1,2), p_2 + v(2,0)\} = \max\{2, 5+0\} = 5,$$
$$v(2,3) = \max\{v(1,3), p_2 + v(2,1)\} = \max\{3, 5+1\} = 6,$$
$$v(2,4) = \max\{v(1,4), p_2 + v(2,2)\} = \max\{4, 5+5\} = 10.$$

* For $j = 3$, pieces of length 1, 2, and 3 can be used. We obtain:

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

* For $j = 4$, all piece lengths are available. We obtain:

$$v(4,0) = 0, \qquad v(4,1) = v(3,1) = 1,$$
$$v(4,2) = v(3,2) = 5, \qquad v(4,3) = v(3,3) = 8,$$
$$v(4,4) = \max\{v(3,4), p_4 + v(4,0)\} = \max\{10, 9+0\} = 10.$$

– 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 rod length, we can solve the subproblems in increasing order of j and, for each fixed value of j, in increasing order of rod length, starting from the base cases $v(0,0) = 0$ and $v(0,\ell) = -\infty$ for all $\ell \in \{1, \ldots, n\}$ and building up to $v(n,n)$.

– This observation naturally leads to a **much more efficient method** to compute the optimal value $v(n,n)$ of the RCP instance.

– A **second method** to compute the optimal value $v(n,n)$ of the RCP instance is the **bottom-up approach**. The method starts from the base cases $v(0,0) = 0$ and $v(0,\ell) = -\infty$ for all $\ell \in \{1, \ldots, n\}$ and evaluates formula (13.2) for increasing values of $j \in \{1, 2, \ldots, n\}$ and $\ell \in \{0, 1, \ldots, n\}$: each value $v(j,\ell)$ is computed using already-computed values, until the value $v(n,n)$ of the RCP instance is obtained.

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

Algorithm

Algorithm: DP2

Input: An RCP instance given by the length n of the rod and an array $p[1..n]$ of piece profits

Output: The optimal value of the RCP instance

```
1  Function DP2(n, p)
2      v := new array[0..n][0..n]
3      v[0][0] ← 0
4      for ℓ := 1 to n do
5          v[0][ℓ] ← −∞
6      end
7      for j := 1 to n do
8          for ℓ := 0 to n do
9              if j > ℓ then
10                 v[j][ℓ] ← v[j − 1][ℓ]
11             else
12                 if p[j] + v[j][ℓ − j] > v[j − 1][ℓ] then
13                     v[j][ℓ] ← p[j] + v[j][ℓ − j]
14                 else
15                     v[j][ℓ] ← v[j − 1][ℓ]
16                 end
17             end
18         end
19     end
20     return v[n][n]
```

* The matrix $v[0..n][0..n]$ is the **lookup table**: it stores the optimal values of all subproblems.
* The initialization sets $v[0][0] = 0$ and $v[0][\ell] = -\infty$ for all rod lengths $\ell \in \{1, \ldots, n\}$, since a positive length cannot be filled if no piece length is available.
* The outer loop, over $j \in \{1, 2, \ldots, n\}$, considers the available piece lengths in increasing order.
* The inner loop, over $\ell \in \{0, 1, \ldots, n\}$, computes the best value for each rod length using only pieces of length at most j.
* If $j > \ell$, piece length j cannot be used, and therefore $v[j][\ell] \leftarrow v[j-1][\ell]$.
* Otherwise, the DP algorithm compares the value obtained by not using piece length j, namely $v[j-1][\ell]$, with the value obtained by using one piece of length j, namely $p[j] + v[j][\ell - j]$.
* Finally, the optimal value of the RCP instance is returned as $v[n][n]$.

Example 13.11: Execution of the DP algorithm

We consider the RCP instance of Example 13.1 with a rod of length $n = 4$ and the profits associated with the possible piece lengths are:

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
p_ℓ	1	5	8	9

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

j	ℓ	$v[j][\ell]$	Action
0	0	0	$v[0][0] \leftarrow 0$
0	1	$-\infty$	$v[0][1] \leftarrow -\infty$
0	2	$-\infty$	$v[0][2] \leftarrow -\infty$
0	3	$-\infty$	$v[0][3] \leftarrow -\infty$
0	4	$-\infty$	$v[0][4] \leftarrow -\infty$
1	0	0	$v[1][0] \leftarrow v[0][0] = 0$
1	1	1	$\max\{v[0][1], p[1] + v[1][0]\} = \max\{-\infty, 1 + 0\} = 1$
1	2	2	$\max\{v[0][2], p[1] + v[1][1]\} = \max\{-\infty, 1 + 1\} = 2$
1	3	3	$\max\{v[0][3], p[1] + v[1][2]\} = \max\{-\infty, 1 + 2\} = 3$
1	4	4	$\max\{v[0][4], p[1] + v[1][3]\} = \max\{-\infty, 1 + 3\} = 4$
2	0	0	$v[2][0] \leftarrow v[1][0] = 0$
2	1	1	$2 > 1$: $v[2][1] \leftarrow v[1][1] = 1$
2	2	5	$\max\{v[1][2], p[2] + v[2][0]\} = \max\{2, 5 + 0\} = 5$
2	3	6	$\max\{v[1][3], p[2] + v[2][1]\} = \max\{3, 5 + 1\} = 6$
2	4	10	$\max\{v[1][4], p[2] + v[2][2]\} = \max\{4, 5 + 5\} = 10$
3	0	0	$v[3][0] \leftarrow v[2][0] = 0$
3	1	1	$3 > 1$: $v[3][1] \leftarrow v[2][1] = 1$
3	2	5	$3 > 2$: $v[3][2] \leftarrow v[2][2] = 5$
3	3	8	$\max\{v[2][3], p[3] + v[3][0]\} = \max\{6, 8 + 0\} = 8$
3	4	10	$\max\{v[2][4], p[3] + v[3][1]\} = \max\{10, 8 + 1\} = 10$
4	0	0	$v[4][0] \leftarrow v[3][0] = 0$
4	1	1	$4 > 1$: $v[4][1] \leftarrow v[3][1] = 1$
4	2	5	$4 > 2$: $v[4][2] \leftarrow v[3][2] = 5$
4	3	8	$4 > 3$: $v[4][3] \leftarrow v[3][3] = 8$
4	4	10	$\max\{v[3][4], p[4] + v[4][0]\} = \max\{10, 9 + 0\} = 10$

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

$v[j][\ell]$	$\ell = 0$	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
$j = 0$	0	$-\infty$	$-\infty$	$-\infty$	$-\infty$
$j = 1$	0	1	2	3	4
$j = 2$	0	1	5	6	10
$j = 3$	0	1	5	8	10
$j = 4$	0	1	5	8	10

– After the execution of the DP algorithm, the matrix v contains the optimal value

$v[j][\ell]$ for every subproblem (j, ℓ).

- Step 2: computation of an optimal solution
  - To compute an optimal solution, it is necessary to store some additional information.
  - For each value j of the largest available piece length and each rod length ℓ, in addition to the optimal value $v[j][\ell]$, we store $s[j][\ell]$, which is equal to 1 if one piece of length j is used and to 0 otherwise.
  - Starting from $j = n$ and $\ell = n$, if $s[j][\ell] = 0$, then piece length j is not used and j is decreased by 1. If $s[j][\ell] = 1$, then one piece of length j is used and the residual rod length becomes $\ell - j$. The procedure stops when the residual length reaches 0.

    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

Algorithm: DP2_InfoSol

Input: An RCP instance given by the length n of the rod and an array $p[1..n]$ of piece profits

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

```
1  Function DP2_InfoSol(n, p)
2      v := new array[0..n][0..n]
3      s := new array[0..n][0..n]
4      v[0][0] ← 0
5      s[0][0] ← 0
6      for ℓ := 1 to n do
7          v[0][ℓ] ← −∞
8          s[0][ℓ] ← 0
9      end
10     for j := 1 to n do
11         for ℓ := 0 to n do
12             if j > ℓ then
13                 v[j][ℓ] ← v[j − 1][ℓ]
14                 s[j][ℓ] ← 0
15             else
16                 if p[j] + v[j][ℓ − j] > v[j − 1][ℓ] then
17                     v[j][ℓ] ← p[j] + v[j][ℓ − j]
18                     s[j][ℓ] ← 1
19                 else
20                     v[j][ℓ] ← v[j − 1][ℓ]
21                     s[j][ℓ] ← 0
22                 end
23             end
24         end
25     end
26     return v[n][n], s
```

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

Example 13.12: Computation of the additional information

We consider the RCP instance of Example 13.1 with a rod of length $n = 4$ and the profits associated with the possible piece lengths are:

	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
p_ℓ	1	5	8	9

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

j	ℓ	$s[j][\ell]$	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	$s[0][4] \leftarrow 0$
1	0	0	$1 > 0$: $s[1][0] \leftarrow 0$
1	1	1	$p[1] + v[1][0] = 1 + 0 > v[0][1] = -\infty$: $s[1][1] \leftarrow 1$
1	2	1	$p[1] + v[1][1] = 1 + 1 > v[0][2] = -\infty$: $s[1][2] \leftarrow 1$
1	3	1	$p[1] + v[1][2] = 1 + 2 > v[0][3] = -\infty$: $s[1][3] \leftarrow 1$
1	4	1	$p[1] + v[1][3] = 1 + 3 > v[0][4] = -\infty$: $s[1][4] \leftarrow 1$
2	0	0	$2 > 0$: $s[2][0] \leftarrow 0$
2	1	0	$2 > 1$: $s[2][1] \leftarrow 0$
2	2	1	$p[2] + v[2][0] = 5 + 0 > v[1][2] = 2$: $s[2][2] \leftarrow 1$
2	3	1	$p[2] + v[2][1] = 5 + 1 > v[1][3] = 3$: $s[2][3] \leftarrow 1$
2	4	1	$p[2] + v[2][2] = 5 + 5 > v[1][4] = 4$: $s[2][4] \leftarrow 1$
3	0	0	$3 > 0$: $s[3][0] \leftarrow 0$
3	1	0	$3 > 1$: $s[3][1] \leftarrow 0$
3	2	0	$3 > 2$: $s[3][2] \leftarrow 0$
3	3	1	$p[3] + v[3][0] = 8 + 0 > v[2][3] = 6$: $s[3][3] \leftarrow 1$
3	4	0	$p[3] + v[3][1] = 8 + 1 \leq v[2][4] = 10$: $s[3][4] \leftarrow 0$
4	0	0	$4 > 0$: $s[4][0] \leftarrow 0$
4	1	0	$4 > 1$: $s[4][1] \leftarrow 0$
4	2	0	$4 > 2$: $s[4][2] \leftarrow 0$
4	3	0	$4 > 3$: $s[4][3] \leftarrow 0$
4	4	0	$p[4] + v[4][0] = 9 + 0 \leq v[3][4] = 10$: $s[4][4] \leftarrow 0$

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

$s[j][\ell]$	$\ell = 0$	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
$j = 0$	0	0	0	0	0
$j = 1$	0	1	1	1	1
$j = 2$	0	0	1	1	1
$j = 3$	0	0	0	1	0
$j = 4$	0	0	0	0	0

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

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

The procedure stops when $\ell = 0$. This yields the vector $(x_1, x_2, \ldots, x_n)$, where each entry x_h is the number of pieces of length h in an optimal solution.

– The following pseudo-code implements the computation of an optimal solution, given the matrix s.

Algorithm

Algorithm: DP2_SolComputation

Input: The rod length n and the matrix $s[0..n][0..n]$
Output: An optimal solution of the RCP instance

```
1  Function DP2_SolComputation(n, s)
2      x := new array[1..n]
3      for h := 1 to n do
4          x[h] ← 0
5      end
6      j := n
7      ℓ := n
8      while ℓ > 0 do
9          if s[j][ℓ] = 0 then
10             j ← j − 1
11         else
12             x[j] ← x[j] + 1
13             ℓ ← ℓ − j
14         end
15     end
16     return x
```

Example 13.13: Computing an optimal solution

After the execution of the DP algorithm on the instance of Example 13.1, we have the following matrix s:

$s[j][\ell]$	$\ell=0$	$\ell=1$	$\ell=2$	$\ell=3$	$\ell=4$
$j=0$	0	0	0	0	0
$j=1$	0	1	1	1	1
$j=2$	0	0	1	1	1
$j=3$	0	0	0	1	0
$j=4$	0	0	0	0	0

We can compute an optimal solution for a rod of length $n=4$ as follows:

Iteration	j	ℓ	$s[j][\ell]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	Action
1	4	4	0	0	0	0	0	$j \leftarrow 3$
2	3	4	0	0	0	0	0	$j \leftarrow 2$
3	2	4	1	0	1	0	0	$x[2] \leftarrow 1$, $\ell \leftarrow 4-2=2$
4	2	2	1	0	2	0	0	$x[2] \leftarrow 2$, $\ell \leftarrow 2-2=0$

An optimal solution is the vector:

$$(0, 2, 0, 0)$$

– The same computation procedure, based on the matrix s, can be applied to any rod length $\ell \leq n$ by starting from $j = n$.

Example 13.14: Computing optimal solutions

After the execution of the DP algorithm on the instance of Example 13.1, we have the following matrix s:

$s[j][\ell]$	$\ell = 0$	$\ell = 1$	$\ell = 2$	$\ell = 3$	$\ell = 4$
$j = 0$	0	0	0	0	0
$j = 1$	0	1	1	1	1
$j = 2$	0	0	1	1	1
$j = 3$	0	0	0	1	0
$j = 4$	0	0	0	0	0

We can compute an optimal solution for a rod of length $\ell = 3$ as follows:

Iteration	j	ℓ	$s[j][\ell]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	Action
1	4	3	0	0	0	0	0	$j \leftarrow 3$
2	3	3	1	0	0	1	0	$x[3] \leftarrow 1$, $\ell \leftarrow 3 - 3 = 0$

An optimal solution is the vector:

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

We can compute an optimal solution for a rod of length $\ell = 2$ as follows:

Iteration	j	ℓ	$s[j][\ell]$	$x[1]$	$x[2]$	$x[3]$	$x[4]$	Action
1	4	2	0	0	0	0	0	$j \leftarrow 3$
2	3	2	0	0	0	0	0	$j \leftarrow 2$
3	2	2	1	0	1	0	0	$x[2] \leftarrow 1$, $\ell \leftarrow 2 - 2 = 0$

An optimal solution is the vector:

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

13.3.1 Number of steps

- Let $S(n)$ denote the number of steps executed by `DP2` in the worst case, where n is the rod length. We compute $S(n)$ by counting how many times the lines of the pseudo-code of `DP2` are executed.

- The number of times each line of code is executed is:
  - Line 2 is executed 1 time.
  - Line 3 is executed 1 time.
  - Line 4 is executed n times.
  - Line 5 is executed n times.
  - Line 7 is executed n times.
  - Line 8 is executed $n(n+1)$ times.
  - Line 9 is executed $n(n+1)$ times.
  - Line 12 is checked only when $j \leq \ell$, that is
  $$\sum_{j=1}^{n}(n-j+1) = \sum_{j=1}^{n} j = \frac{n(n+1)}{2}$$
  times.
  - One among Line 10, Line 13, and Line 15 is executed for each pair (j, ℓ), and therefore these assignment lines are executed $n(n+1)$ times in total.
  - Line 20 is executed 1 time.

  Therefore, the total number of steps in the worst case is
  $$1+1+n+n+n+n(n+1)+n(n+1)+\frac{n(n+1)}{2}+n(n+1)+1.$$
  Thus,
  $$S(n) = 3+3n+3n(n+1)+\frac{n(n+1)}{2} = \frac{7n^2+13n+6}{2}.$$
  Hence,
  $$S(n) = \Theta(n^2).$$
  Indeed,
  $$\lim_{n\to\infty} \frac{\frac{7n^2+13n+6}{2}}{n^2} = \lim_{n\to\infty} \frac{1}{2}\left(7+\frac{13}{n}+\frac{6}{n^2}\right) = \frac{7}{2} \in \mathbb{R}_{>0}.$$

The DP algorithm executes
$$S(n) = \Theta(n^2)$$
steps in the worst case.

13.4 Typical mistakes

Warning

- **Counting vectors instead of cutting patterns.** The 2^{n-1} cutting patterns of a rod of length n are more than the solution vectors when $n \geq 3$, because different patterns can give the same vector.
- **Forgetting the base case.** $v(0) = 0$ is used when the first piece has the length of the whole rod: $v(\ell)$ contains the term $p_\ell + v(0)$.
- **Using row $j-1$ in the second recurrence.** The term $p_j + v(j, \ell - j)$ stays in row j: otherwise a piece length could be cut only once.
- **Stopping the reconstruction early.** The rule $x_{s[\ell]} \leftarrow x_{s[\ell]} + 1$, $\ell \leftarrow \ell - s[\ell]$ is repeated until $\ell = 0$; the last piece completes the rod.

13.5 In summary

Summary

- The RCP asks how to cut a rod of length n into pieces, with each length available any number of times, to maximize the total profit.
- The recurrence $v(\ell) = \max_{j \leq \ell}\{p_j + v(\ell - j)\}$ has the optimal substructure and overlapping subproblems, and $v(0) = 0$.
- The vector v is filled bottom-up in $\Theta(n^2)$ steps, which is polynomial because the input has n profits.
- The vector s of the first pieces gives an optimal solution for every rod length at no extra cost.

Dashboard of the RCP example. Example 13.1 (rod of length 4):

Method	Lower bound	Upper bound	Origin
DP algorithm	10	10	optimal value, two pieces of length 2