DP for the bounded knapsack problem

Outline

Instances and feasible solutions
    Encoding of feasible solutions

A DP algorithm
    Number of steps

Typical mistakes

In summary

Learning 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.

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 and the size of the input.

The problem of this chapter

The problem of this chapter is the

bounded knapsack problem (BKP).

Its definition and its assumptions are on its page.

Instances and feasible solutions

Introduction: instances and feasible solutions

- We fix the BKP instance that we use in all the examples of the chapter
- We see how the BKP differs from the 0/1 knapsack problem (KP) and from the unbounded knapsack problem (UKP)
- We count the possible solutions: enumerating them quickly becomes impractical, so we need a better method
- We describe a feasible solution as an integer vector: this is what the DP algorithm will compute

Example: 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 use this BKP instance in all the examples of the chapter.

From the KP to the BKP

- 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
- The BKP reduces to the unbounded knapsack problem (UKP) when every bound b_j is large enough not to be binding

How many solutions?

- 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

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

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

Encoding of feasible solutions

- For every item type $j \in \{1, 2, \ldots, n\}$, each component has the following meaning:

$$x_j = \text{number of copies of item type } j \text{ selected,}$$

$$0 \leq x_j \leq b_j.$$

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

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

- The total profit of a feasible solution is

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

Summary

Instances:

- The example instance: $c = 10$ and three item types, with bounds $b_1 = 2$, $b_2 = 2$, $b_3 = 1$
- Each item type may be selected up to b_j times: KP when $b_j = 1$, UKP when the bounds are not binding

Feasible solutions:

- At most $\prod_{j=1}^{n}(b_j + 1)$ candidate vectors: exponential in n
- A feasible solution is an integer vector $(x_1, x_2, \ldots, x_n)$ with $0 \leq x_j \leq b_j$ and $\sum_{j=1}^{n} w_j x_j \leq c$; its profit is $\sum_{j=1}^{n} p_j x_j$

A DP algorithm

Introduction: a DP algorithm

- **Step 1**: we define the subproblems $v(j, k)$ and a recurrence that tries every feasible number d of copies of item type j
- We check optimal substructure and overlapping subproblems, and we compare the top-down and the bottom-up approaches
- **Step 2**: we store the number of copies chosen in each subproblem and we reconstruct an optimal solution
- Finally, we count the steps: the DP algorithm runs in $\Theta(nc\, b_{\max})$ steps, a pseudo-polynomial number, while enumeration is exponential

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 recurrence

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

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

Subproblems and base cases

- The recursive decomposition process of the recurrence generates at most $b_j + 1$ **independent subproblems**
- They correspond 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;
  - $v(j, 0) = 0$, since no capacity is available.

Optimal substructure (1/2)

- 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 $1, 2, \ldots, j-1$

Optimal substructure (2/2)

- 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!**

Overlapping subproblems

- 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**
- Therefore, the same value $v(j, k)$ may need to be computed several times during the recursive evaluation of the recurrence

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

First method: the top-down approach

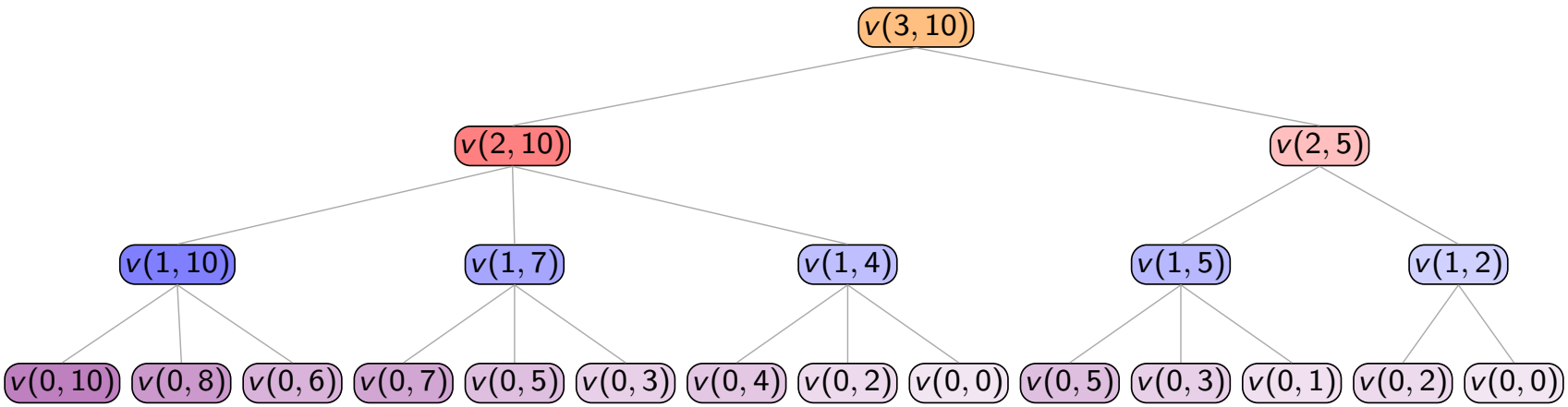
- A **first method** to compute the optimal value $v(n, c)$ of the BKP instance is the **top-down approach**
- The method starts from the call with $j = n$ and $k = c$ and evaluates the recurrence recursively
- Each call considers item type j and tries every feasible number d of copies of it, until no item type is left or the residual capacity is equal to 0

The recursion tree

- The **recursion tree** of the function v has one node for each recursive call, labeled with the function call it represents
- The root represents the initial call
- The children of a node represent the recursive calls made by that call
- The leaf nodes represent the base cases

Example: recursion tree

We consider the BKP instance of the example. 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: $v(0, 5)$, $v(0, 3)$, $v(0, 2)$ and $v(0, 0)$ are called 2 times each. All the other calls are made only once.

Order of computation

- 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

Let us see this order on the example instance.

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

We consider the BKP instance of the example 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 $k \geq 0$.

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

- 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 $\lfloor 4/2 \rfloor = 2$):

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

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

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

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

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

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

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

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

- 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 $\lfloor 5/3 \rfloor = 1$):

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

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

- 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 $b_3 = 1$):

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

The lookup table

- We can avoid solving the same subproblems repeatedly by storing the value of each subproblem as soon as it is computed
- The values are stored in a **lookup table**

Solving the subproblems in order

- 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

Second method: the bottom-up approach

- 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 the recurrence 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

Next: the pseudo-code of the **DP algorithm using the bottom-up approach**.

Algorithm: BKP_DP (1/2)

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
```

Algorithm: BKP_DP (2/2)

```
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]
18 end
```

How BKP_DP works (1/2)

- The matrix $v[0..n][0..c]$ is the **lookup table**: it stores the optimal values of all subproblems
- The first loop initializes the values $v[0][k] = 0$ for all capacities $k \in \{0, 1, \ldots, c\}$, corresponding to the case in which no item 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

How BKP_DP works (2/2)

- 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: execution of the DP algorithm (1/8)

We consider the BKP instance of the example 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 tables on the next slides show the values of j and k, the cell $v[j][k]$, and the operation performed.

Example: execution of the DP algorithm (2/8)

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	$v[0][10] \leftarrow 0$

Example: execution of the DP algorithm (3/8)

j	k	$v[j][k]$	Action
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	$\max\{v[0][5],\ p[1]+v[0][3],\ 2p[1]+v[0][1]\}$ $= \max\{0, 6, 12\} = 12$

Example: execution of the DP algorithm (4/8)

j	k	$v[j][k]$	Action
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	$\max\{v[0][10],\ p[1]+v[0][8],\ 2p[1]+v[0][6]\}$ $= \max\{0, 6, 12\} = 12$

Example: execution of the DP algorithm (5/8)

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	$\max\{v[1][5],\ p[2]+v[1][2]\} = \max\{12, 10+6\} = 16$

Example: execution of the DP algorithm (6/8)

j	k	$v[j][k]$	Action
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	$\max\{v[1][10],\ p[2]+v[1][7],\ 2p[2]+v[1][4]\}$ $= \max\{12, 22, 32\} = 32$

Example: execution of the DP algorithm (7/8)

j	k	$v[j][k]$	Action
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	$\max\{v[2][10],\ p[3]+v[2][5]\} = \max\{32, 12+16\} = 32$

Example: execution of the DP algorithm (8/8)

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

$v[j][k]$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$
$j=0$	0	0	0	0	0	0
$j=1$	0	0	6	6	12	12
$j=2$	0	0	6	10	12	16
$j=3$	0	0	6	10	12	16

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

Step 2: computation of an optimal solution

- 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 $k \in \{0, 1, \ldots, c\}$
- 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**

The matrix s of the numbers of copies

- 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 the recurrence when computing $v[j][k]$

Next: 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: BKP_DP_InfoSol (1/2)

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
```

Algorithm: BKP_DP_InfoSol (2/2)

```
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
18 end
```

Storing the decisions

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

We consider the BKP instance of the example 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 tables on the next slides show 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 $s[j][k]$

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

Two remarks on the tables:

- 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

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

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

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

j	k	$s[j][k]$	Values for $d = 0, 1, \ldots$
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

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

j	k	$s[j][k]$	Values for $d = 0, 1, \ldots$
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

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

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

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

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

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

For example:

- $s[1][4] = 2$, since the recurrence selects $d = 2$ copies of item type 1 when computing $v[1][4] = 12$
- $s[2][10] = 2$, 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

Reconstructing an optimal solution

- 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: 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
9 end
```

Example: computing an optimal solution (1/2)

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

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

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

Example: computing an optimal solution (2/2)

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 $v(3, 10) = 32$.

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

The innermost loop

- 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)
- We assume that the condition at line 10 is true at every iteration
- In this way $S(n, c, b_{\max})$ is an **upper bound** on the number of steps

On the next two slides we recall the pseudo-code of `BKP_DP`, with its line numbers.

Algorithm: BKP_DP (1/2)

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
```

Algorithm: BKP_DP (2/2)

```
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]
18 end
```

Counting the executions (1/2)

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.

Counting the executions (2/2)

The number of executions of the relevant lines is, at most:

- 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.

The total number of steps

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

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

Thus,

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

Hence,

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

Checking the order of growth

Indeed,

$$\lim_{n,c,b_{\max}\to\infty} \frac{3n(c+1)(b_{\max}+1)+3n(c+1)+n+2c+4}{nc\,b_{\max}}$$

$$= \lim_{n,c,b_{\max}\to\infty} \left(\frac{3(c+1)(b_{\max}+1)}{c\,b_{\max}} + \frac{3(c+1)}{c\,b_{\max}} + \frac{1}{c\,b_{\max}} + \frac{2}{n\,b_{\max}} + \frac{4}{nc\,b_{\max}} \right)$$

$$= 3 \in \mathbb{R}_{>0}.$$

Number of steps of the DP algorithm

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 $\Theta(nc\, b_{\max})$

When is the bound tight?

- 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

Summary (1/2)

Step 1: the optimal value:

- $v(j, k)$ tries every number $d \in \{0, 1, \ldots, \min\{b_j, \lfloor k/w_j \rfloor\}\}$ of copies of item type j; base cases $v(0, k) = v(j, 0) = 0$
- Optimal substructure and overlapping subproblems hold: the top-down approach calls v 22 times on the example, the bottom-up approach fills the lookup table row by row
- `BKP_DP` returns $v[n][c]$: on the example, $v(3, 10) = 32$

Step 2: an optimal solution:

- $s[j][k]$ stores the number of copies chosen (the smallest d in case of ties)
- Backwards from (n, c): $x_j \leftarrow s[j][k]$, $k \leftarrow k - s[j][k]\, w_j$; on the example, $(2, 2, 0)$

Summary (2/2)

Number of steps:

- In the worst case $S(n, c, b_{\max}) = 3n(c+1)(b_{\max}+1) + 3n(c+1) + n + 2c + 4 = \Theta(nc\, b_{\max})$
- `BKP_DP_InfoSol` has the same order of growth
- With $b_{\max} \leq c$ the bound is tight: all the weights equal to 1 and all the bounds equal to $b_{\max}$

Typical mistakes

Introduction: typical mistakes

- We list the mistakes that are made most often with the DP algorithm for the BKP
- They concern the range of the number of copies, the reconstruction, the count of the steps and the ties
- Knowing them helps to run the algorithm by hand correctly in the exercises

Observation 1: Typical mistakes (1/2)

- ***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.*

Observation: Typical mistakes (2/2)

- ***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.*

Summary

Typical mistakes:

- The number of copies is at most $\min\{b_j, \lfloor k/w_j \rfloor\}$: both the bound and the capacity matter
- In the reconstruction the capacity decreases by $s[j][k]\, w_j$, and j always decreases by one
- $b_{\max} + 1$ iterations per cell is only an upper bound
- Ties: m is updated only on a strictly larger value, so $s[j][k]$ keeps the smallest d

In summary

Introduction: in summary

- We collect the key ideas of the chapter
- We close with the dashboard of the example instance: the bounds found in this chapter

Summary (1/2)

- 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 $b_{\max} \leq c$.

Summary (2/2)

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

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