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

Outline

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

Typical mistakes

In summary

Where we were

- In the first part, the recurrence $v(j, k)$ (the items $\{1, 2, \ldots, j\}$ and a knapsack of capacity k) gave the optimal value of the KP.
- Bottom-up, the matrix v is filled in $\Theta(nc)$ steps; a single vector is enough for the optimal value, and the matrix s of the decisions gives an optimal solution.
- The lists of non-dominated states (w, p) store only the states that can be reached.
- On the first instance ($c = 5$, profits $6, 7, 8$, weights $2, 3, 3$) the optimal value is 14, with the optimal solution $(1, 0, 1)$.
- **Now**: a second DP algorithm, which exchanges the roles of profits and weights.

A second DP algorithm

Introduction: a second DP algorithm

- We exchange the roles of the profits and of the weights: a subproblem is identified by a target profit q
- Its value $u(j, q)$ is the minimum total weight needed to obtain exactly the profit q
- We follow the same path as for the first DP algorithm: recurrence, top-down, bottom-up, single vector, optimal solution, number of steps and lists of states
- The number of steps becomes $\Theta(nP)$, where P is the total profit of the items: better when P is small

Step 1: computation of the optimal value

Exchanging profits and weights (1/2)

- 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

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

Exchanging profits and weights (2/2)

- 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 first instance, and smaller item profits.

Example: KP instance

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

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

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

The subproblems $u(j, q)$

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 recurrence of $u(j, q)$

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

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

Reading the recurrence of $u(j, q)$ (1/2)

- The recursive decomposition process of formula (1) 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.

Reading the recurrence of $u(j, q)$ (2/2)

- 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!**

Overlapping subproblems

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

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

From the values $u(n, q)$ to the optimal value

- 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

$$\max \left\{ q \in \{0, 1, \ldots, P\} \ : \ u(n, q) \leq c \right\}. \tag{2}$$

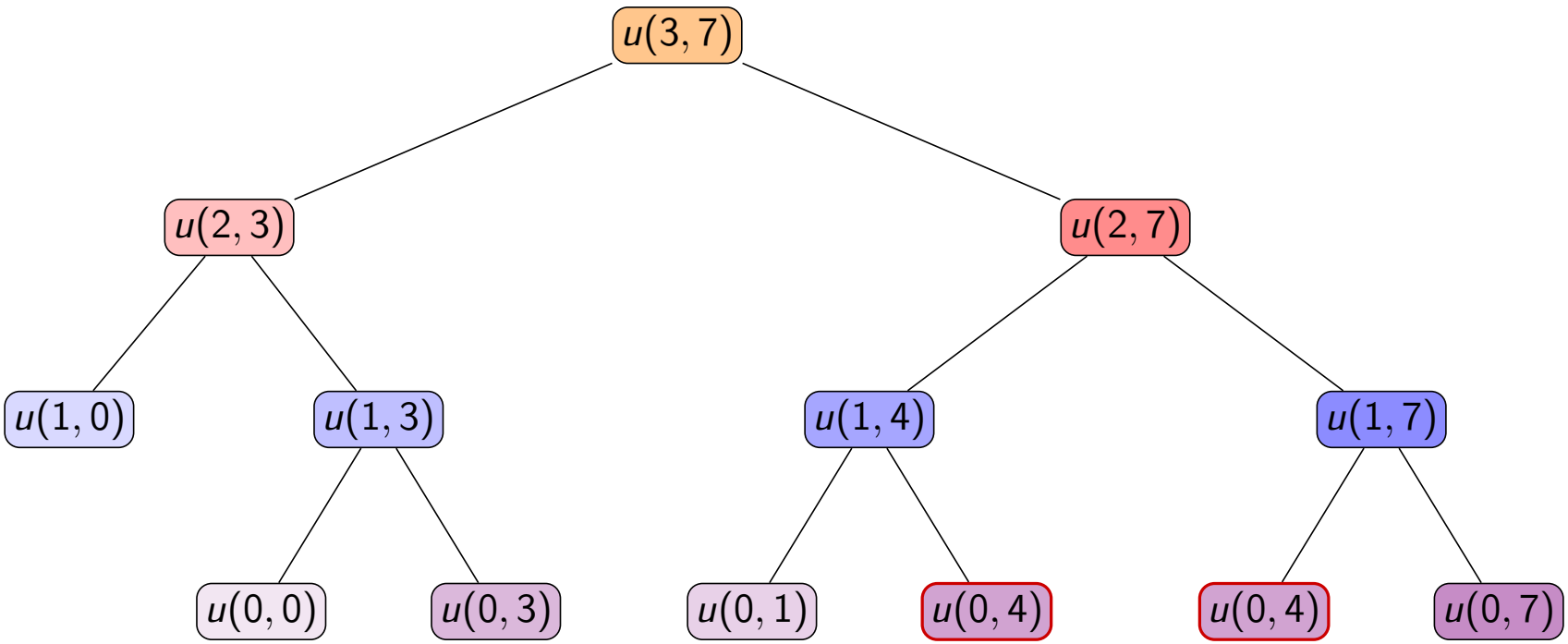
- The set in formula (2) is never empty, since $u(n, 0) = 0 \leq c$, and therefore the maximum is always well defined.

The top-down approach

- A **first method** to compute the optimal value of the KP instance is the **top-down approach**.
- The method evaluates formula (1) 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) 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: recursion tree (1/2)

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



Example: recursion tree (2/2)

- 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 $u(3, 7)$ of the previous slide.

Order of computation

- The value $u(j, q)$ can be computed once the values required by the right-hand side of formula (1) 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: computing the values of the u function (1/3)

We consider the second instance, 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 (1) ($q = 0$) gives the base case:

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

Example: computing the values of the u function (2/3)

- 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 $u(2,3)$:

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

Example: computing the values of the u function (3/3)

- 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 $c = 5$.

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

- We can avoid solving the same subproblems repeatedly by storing the value of each subproblem as soon as it is computed. The values are stored in a **lookup table**.
- Since solving any particular subproblem depends only on subproblems with a smaller value of j, we can solve the subproblems in increasing order of j and, for each fixed value of j, in increasing order of target profit.
- We start 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 build 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.

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

- 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 (1) 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).

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

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
```

Algorithm: DP2 (continued)

```
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
```

Algorithm: DP2 (continued)

```
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
31  end
```

How DP2 works (1/2)

- 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.

How DP2 works (2/2)

- 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).
- Finally, the optimal value of the KP instance is returned as z.

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

We consider the second instance, 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 tables of the next slides show the values of j and q, the cell $u[j][q]$, and the operation performed.

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

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	$+\infty$	$u[0][10] \leftarrow +\infty$

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

j	q	$u[j][q]$	Action
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	$+\infty$	$\min\{u[0][10], w[1] + u[0][7]\} = \min\{+\infty, +\infty\} = +\infty$

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

j	q	$u[j][q]$	Action
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	$+\infty$	$\min\{u[1][10], w[2] + u[1][7]\} = \min\{+\infty, +\infty\} = +\infty$

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

j	q	$u[j][q]$	Action
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	$\min\{u[2][10], w[3] + u[2][6]\} = \min\{+\infty, 3 + 5\} = 8$

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

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

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

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 $z = 7$.

From the matrix to a single vector

- 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.

A single vector (1/2)

- 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 $u(j - 1, q - p_j)$.

A single vector (2/2)

- 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: 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
```

Algorithm: DP2_Vector (continued)

```
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
```

Algorithm: DP2_Vector (continued)

```
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
25  end
```

How DP2_Vector works (1/2)

- 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.

How DP2_Vector works (2/2)

- 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).

Example: execution of DP2 using a single vector (1/5)

We consider the second instance, 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 tables of the next slides show the values of j and q, the cell $u[q]$ after the step, and the operation performed.

Example: execution of DP2 using a single vector (2/5)

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	$w[1] + u[0] = 2 < u[3] = +\infty$: $u[3] \leftarrow 2$

Example: execution of DP2 using a single vector (3/5)

j	q	$u[q]$	Action
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

Example: execution of DP2 using a single vector (4/5)

j	q	$u[q]$	Action
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	$w[3]+u[0]=3<u[4]=+\infty$: $u[4]\leftarrow 3$

Example: execution of DP2 using a single vector (5/5)

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 computed by the DP algorithm on the same instance. The last loop then scans the vector and gives the optimal value $z = 7$ of the KP instance, as with the matrix.

The vector computes only the optimal value

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

Step 2: computation of an optimal solution

The matrix s of the decisions

- To compute an optimal solution, i.e., the binary vector $(x_1, x_2, \ldots, x_n)$, it is necessary to store some additional information.
- For each item j and 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: 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
```

Algorithm: DP2_InfoSol (continued)

```
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
```

Algorithm: DP2_InfoSol (continued)

```
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
```

Algorithm: DP2_InfoSol (continued)

```
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
37  end
```

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

We consider the second instance, 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 tables of the next slides show the values of j and q, the cell $s[j][q]$, and the operation performed.

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

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	$s[0][10] \leftarrow 0$

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

j	q	$s[j][q]$	Action
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	$w[1] + u[0][7] = +\infty \geq u[0][10] = +\infty$: $s[1][10] \leftarrow 0$

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

j	q	$s[j][q]$	Action
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	$w[2] + u[1][7] = +\infty \geq u[1][10] = +\infty$: $s[2][10] \leftarrow 0$

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

j	q	$s[j][q]$	Action
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	$w[3] + u[2][6] = 3 + 5 < u[2][10] = +\infty$: $s[3][10] \leftarrow 1$

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

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

Reconstruction of an optimal solution

- Once the matrix s has been computed, an optimal solution can be reconstructed by scanning the items from $j = n$ down to $j = 1$, starting from 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: 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
```

Algorithm: DP2_Reconstruct (continued)

```
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
14  end
```

Example: computing an optimal solution (1/2)

After the execution of the DP algorithm on the second instance, 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

Example: computing an optimal solution (2/2)

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 $3 + 4 = 7$.

How many steps does the second DP algorithm execute?

We count how many times each line of the pseudo-code of `DP2` is executed.

Number of steps of DP2 (1/5)

- 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.

Number of steps of DP2 (2/5)

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.

Number of steps of DP2 (3/5)

- 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.

Number of steps of DP2 (4/5)

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,

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

Number of steps of DP2 (5/5)

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 number of steps of the three versions

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 $P + 1$.

An implementation with lists

Step 1: computation of the optimal value

Why lists?

- The DP algorithm of this section stores one value for every pair given by an item and a 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.
- We now see a **different implementation of the same DP algorithm**. The recursive decomposition and the optimal substructure are the ones of formula (1): what changes is the data structure.
- Instead of a whole row of the matrix, only the pairs 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.

States

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

- 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.

Dominated states

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

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

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

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

The recurrence of L_j

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:

$$L_j = \begin{cases} \{(0,0)\} & \text{if } j = 0, \\ \mathrm{nd}(L_{j-1} \cup M_j) & \text{if } 1 \leq j \leq n, \end{cases} \tag{3}$$

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

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

Reading the recurrence of L_j (1/2)

- The recursive decomposition process of formula (3) 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 (1), 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.

Reading the recurrence of L_j (2/2)

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

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

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

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

The optimal value from the last list

The knapsack capacity is used only at the end, exactly as in formula (2): 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

$$\max\left\{ p \ : \ (w,p) \in L_n, \ w \leq c \right\}. \tag{4}$$

- The set in formula (4) is never empty, since $(0,0) \in L_n$ and $0 \leq c$, and therefore the maximum is always well defined.

The lists and the rows of the matrix u

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

$$\min\left\{w \ : \ (w,p) \in L_j, \ p \geq q\right\} = \min_{q' \geq q} u[j][q'] \quad \forall q \in \{0,1,\ldots,P\}.$$

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

Example: the lists of states (1/3)

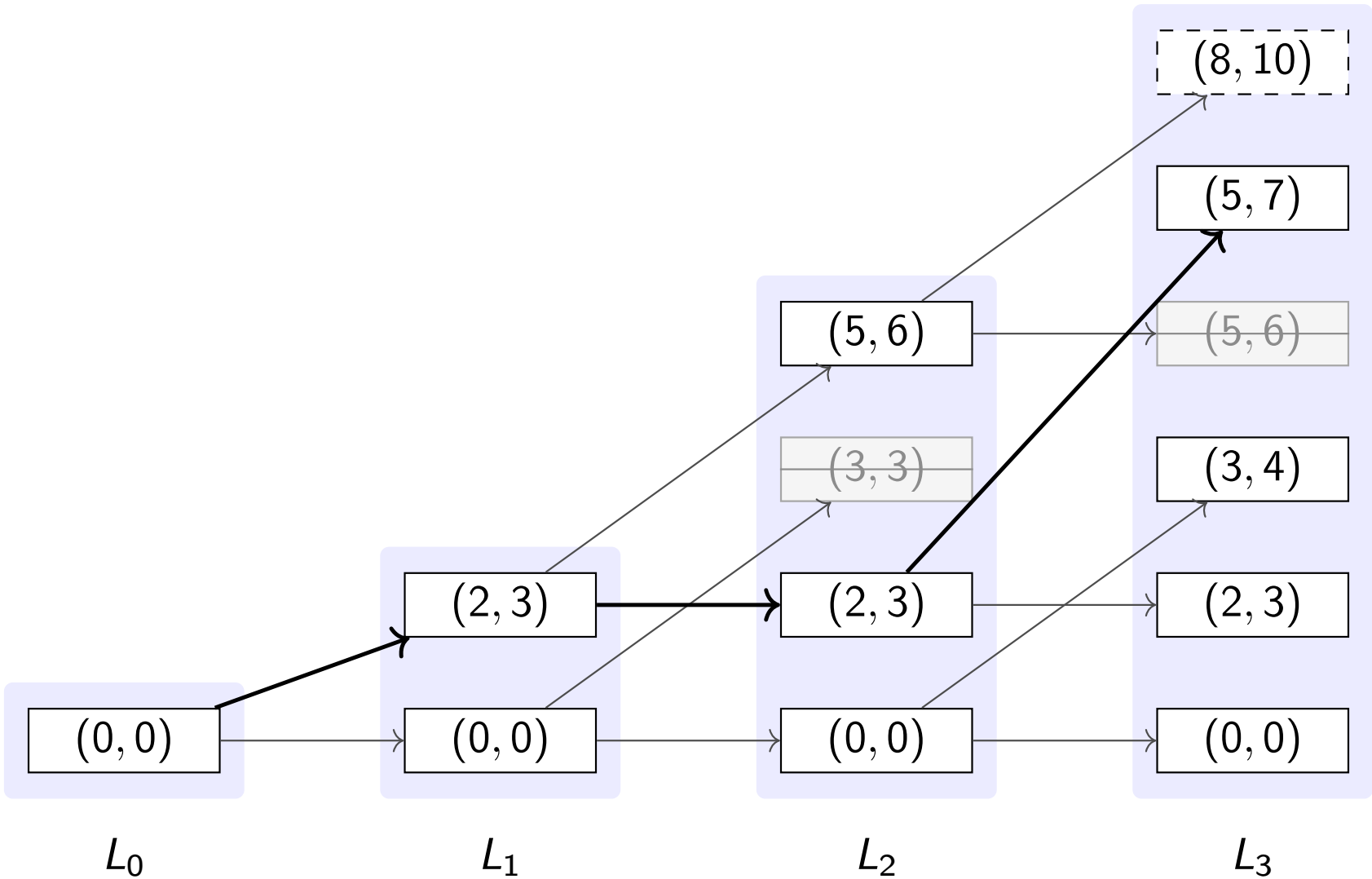
We consider the second instance, with a knapsack of capacity $c = 5$ and the following profits and weights:

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

In the figure of the next slide:

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

Example: the lists of states (2/3)



Example: the lists of states (3/3)

- The two crossed states 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 (4).

Computing the lists

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

The scanning order

To describe the scanning order we use the following convention.

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

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

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

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

Algorithm: 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
```

Algorithm: DP2_Lists (continued)

```
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
17  end
```

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

Algorithm: Merge

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

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

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

Algorithm: Merge (continued)

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

How DP2_Lists works (1/2)

- The list L is the **lookup table**: at the end of the iteration for item j, it stores the list L_j.
- The list L is initialized with the single state $(0, 0)$, which is the list L_0.
- The outer loop, over items $j \in \{1, 2, \ldots, n\}$, computes one list for each item.
- The inner loop builds the list M of the states obtained by selecting item j. No state is discarded here, because the knapsack capacity is not used during the recursion.

How DP2_Lists works (2/2)

- The procedure `Merge` scans the two lists at the same time and keeps a state only when its profit is strictly larger than the largest profit already appended, which is stored in the auxiliary variable π. The initial value $\pi = -1$ guarantees that the first scanned state is always appended.
- The last loop implements formula (4). 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.
- 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.

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

We consider the second instance, 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	$\{(3,4),(5,7),(8,10)\}$	$\{(0,0),(2,3),(3,4),(5,7),(8,10)\}$

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

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

j	state	from	Action
1	$(0, 0)$	L_0	$0 > \pi = -1$: append
1	$(2, 3)$	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

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

j	state	from	Action
3	$(0,0)$	L_2	$0 > \pi = -1$: append
3	$(2,3)$	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.

An implementation with lists

Step 2: computation of an optimal solution

The additional information for the lists (1/2)

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

The additional information for the lists (2/2)

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

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

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

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

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

j	h	$L_j[h]$	$\sigma_j[h]$	$\rho_j[h]$
3	1	$(0,0)$	0	1
3	2	$(2,3)$	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 $3 + p_3 = 3 + 4 = 7$.

Reconstruction with the lists

- The reconstruction starts from the position $\tilde{h}$ of the state that attains the maximum of formula (4), 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: 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
```

Algorithm: DP2_Lists_Reconstruct (continued)

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

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

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

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

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

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

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 $3 + 4 = 7$.

Number of steps of DP2_Lists

- 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

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

Lists and matrix

- 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 second instance 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

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

Summary (1/2)

The recurrence $u(j, q)$:

- $u(j, q)$ is the minimum total weight of a subset of $\{1, 2, \ldots, j\}$ with total profit exactly q
- $u(0, q) = +\infty$ for $q \geq 1$ rules out the infeasible cases
- The optimal value of the KP instance is the largest q with $u(n, q) \leq c$

Computing the optimal value:

- Top-down, bottom-up with the matrix $u[0..n][0..P]$, or with a single vector scanned with decreasing target profits
- $\Theta(nP)$ steps, where $P = \sum_{j=1}^{n} p_j$

Summary (2/2)

An optimal solution:

- The matrix s is read backwards from $j = n$ and $q = z$

Lists of states:

- The capacity is not used during the recursion: the states of L_n with $w > c$ are discarded only at the end
- The reconstruction starts from the position $\tilde{h}$ of the optimal state, not always the last one
- $\Theta(nP)$ steps and $|L_j| \leq \min\{2^j, P + 1\}$

Typical mistakes

Introduction: typical mistakes

- We list the mistakes that are made most often with the DP algorithms for the KP
- They concern the order of the updates, the base cases, the reconstruction of a solution, the number of steps, and the meaning of $u(n, q)$
- Knowing them helps to compute the tables by hand correctly

Observation 1: Typical mistakes (1/2)

- ***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.*

Observation: Typical mistakes (2/2)

- ***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 $u(n, q) \leq c$.*

Summary

Typical mistakes:

- Scan the capacities of the single vector from c downwards
- $v(0, k) = 0$ for every capacity k, not only for $k = 0$
- In the reconstruction, j decreases by one, k by w_j only if item j is selected
- $\Theta(nc)$ is pseudo-polynomial, not polynomial
- $u(n, q)$ is a minimum weight, not the optimal value

In summary

Introduction: in summary

- We collect the key ideas of the chapter
- We close with the dashboard of the first instance: the bounds found so far

Summary (1/2)

- 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.

Summary (2/2)

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

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