The idea behind dynamic programming is to build an exhaustive table of optimal solutions. We start with trivial subproblems. We build optimal solutions for increasingly larger problems by constructing them from the tabulated solutions to smaller problems.

Again, we use the knapsack problem as an example. Define $P(i, C)$ as the maximum profit possible when only items 1 to i can be put in the knapsack and the total weight is at most C. Our goal is to compute $P(n, M)$. We start with trivial cases and work our way up. The trivial cases are "no items" and "total weight zero". In both cases, the maximum profit it zero. So

$$P(0, C) = 0 \text{ for all } C \quad \text{and} \quad P(i, 0) = 0 \ .$$

Consider next the case $i > 0$ and $C > 0$. In the solution maximizing the profit we either use item i or we do not use it. In the latter case, the maximum achievable profit is $P(i-1, C)$. In the former case, the maximum achievable profit is $P(i-1, C-w_i)+p_i$ since we obtain profit p_i for item i and must use a solution of total weight at most $C-w_i$ for the first $i-1$ items. Of course, the former alternative is only feasible if $C \geq w_i$. We summarize the discussion in the following recurrence for $P(i, C)$:

$$P(i, C) = \begin{cases} \max(P(i-1, C), P(i-1, C-w_i) + p_i) & \text{if } w_i \leq C \\ P(i-1, C) & \text{if } w_i > C \end{cases} \tag{12.1}$$

Exercise 216. Show that the case distinction in the definition of $P(i, C)$ can be avoided by defining $P(i, C) = -\infty$ for $C < 0$.

Using the recurrence, we can compute $P(n, M)$ by filling a table $P(i, C)$ with one column for each possible capacity C and one row for each item i. Table 12.1 gives an example. There are many ways to fill this table, for example row by row. In order to reconstruct a solution from this table, we work our way backwards starting at the bottom right-hand corner of the table. Set $i = n$ and $C = M$. If $P(i, C) = P(i-1, C)$ we set $x_i = 0$ and continue in row $i-1$ with column C. Otherwise, we set $x_i = 1$. We have $P(i, C) = P(i-1, C-w_i) + p_i$ and therefore continue in row $i-1$ with column $C-w_i$. We continue with this procedure until we arrive at row 0 by which time the solution $(x_1, \ldots, x_n)$ has been completed.

Exercise 217. Dynamic programming, as described above, needs to store a table with $\Theta(nM)$ integers. Give a more space-efficient solution that stores only a single *bit* in each table entry except for two rows of $P(i, C)$ values at a time. What information is stored in this bit? How is it used to reconstruct a solution? How can you get down to *one* row of stored values? Hint: exploit your freedom in the order of filling in table values.

We will next describe an important optimization. It uses less space and is also faster. Instead of computing $P(i, C)$ for all i and all C, it only computes *Pareto-optimal* solutions. A solution x is Pareto-optimal if there is no solution that *dominates* it, i.e., has greater profit and no greater cost or the same profit and less cost. In

$i \setminus C$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	**0**	10	10	10	10	10
2	0	10	10	**20**	30	30
3	0	10	15	25	30	**35**
4	0	10	15	25	30	**35**

Table 12.1. A dynamic programming table for the knapsack instance with $p = (10, 20, 15, 20)$, $w = (1, 3, 2, 4)$, and $M = 5$. Bold face entries contribute to the optimal solution.

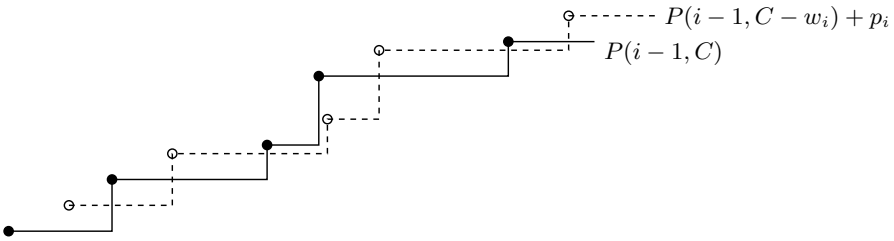

Fig. 12.4. The solid step function shows $C \mapsto P(i-1, C)$ and the dashed step function shows $C \mapsto P(i-1, C-w_i)+p_i$. $P(i, C)$ is the point-wise maximum of both functions. The solid step function is stored as the sequence of solid points. The representation of the dashed step function is obtained by adding (w_i, p_i) to every solid point. The representation of $C \mapsto P(i, C)$ is obtained by merging both representations and deleting all dominated elements.

other words, since $P(i, C)$ is an increasing function of C, we only need to remember those pairs $(C, P(i, C))$ where $P(i, C) > P(i, C-1)$. We store these pairs in a list L_i sorted by C value. So $L_0 = \langle (0, 0) \rangle$ indicating $P(0, C) = 0$ for all $C \geq 0$ and $L_1 = \langle (0, 0), (w_1, p_1) \rangle$ indicating that $P(1, C) = 0$ for $0 \leq C < w_1$ and $P(i, C) = p_1$ for $C \geq w_1$.

How can we go from L_{i-1} to L_i? The recurrence for $P(i, C)$ paves the way, see Figure 12.4. We have the list representation L_{i-1} for the function $C \mapsto P(i-1, C)$. We obtain the representation L'_{i-1} for $C \mapsto P(i-1, C-w_i)+p_i$ by shifting every point in L_{i-1} by (w_i, p_i). We merge L_{i-1} and L'_{i-1} into a single list by order of first component and delete all elements that are dominated by another value, i.e., we delete all elements that are preceded by an element with higher second component and for each fixed value of C, we keep only the element with largest second component.

Exercise 218. Give pseudo-code for the merge. Show that the merge can be carried out in time $|L_{i-1}|$. Conclude that the running time of the algorithm is proportional to the number of Pareto-optimal solutions.

The basic dynamic programming algorithm for the knapsack problem and also its optimization requires $\Theta(nM)$ worst case time. This is quite good if M is not

too large. Since the running time is polynomial in n and M, the algorithm is called *pseudo-polynomial*. The "pseudo" means that it is not necessarily polynomial in the *input size* measured in bits; however, it is polynomial in the natural parameters n and M. There is, however, an important difference between the basic and the refined approach. The basic approach has best case running time $\Theta(nM)$. The best case for the refined approach is $O(n)$. The *average case* complexity of the refined algorithm is polynomial in n, independent of M. This even holds if the averaging is only done over perturbations of an arbitrary instance by small random noise. We refer the reader to [15] for details.

Exercise 219 (Dynamic Programming by Profit). Define $W(i, P)$ to be the smallest weight needed to achieve a profit of at least P using knapsack items $1..i$.

1. Show that $W(i, P) = \min\{W(i-1, P), W(i-1, P-p_i) + w_i\}$.
2. Develop a table-based dynamic programming algorithm using the above recurrence, that computes optimal solutions of the knapsack problem in time $\mathcal{O}(np^*)$ where p^* is the profit of the optimal solution. Hint: assume first that p^* is known or at least a good upper bound for it. Then remove this assumption.

Exercise 220 (Making Change). Suppose you have to program a vending machine that should give exact change using a minimum number of coins.

1. Develop an optimal greedy algorithm that works in the Euro zone with coins worth 1, 2, 5, 10, 20, 50, 100, and 200 cents and in the Dollar zone with coins worth 1, 5, 10, 25, 50, and 100 cents.
2. Show that this algorithm would not be optimal if there were a 4 cent coin.
3. Develop a dynamic programming algorithm that gives optimal change for any currency system.

Exercise 221 (Chained Matrix Products). We want to compute the matrix product $M_1 M_2 \cdots M_n$ where M_i is a $k_{i-1} \times k_i$ matrix. Assume that a pairwise matrix product is computed in the straight-forward way using mks element multiplications for the product of an $m \times k$ matrix with a $k \times s$ matrix. Exploit the associativity of matrix product to minimize the number of element multiplications needed. Use dynamic programming to find an optimal evaluation order in time $\mathcal{O}(n^3)$. For example, the product between a 4×5 matrix M_1, a 5×2 matrix M_2, and a 2×8 matrix M_3 can be computed in two ways. Computing $M_1(M_2 M_3)$ takes $5 \cdot 2 \cdot 8 + 4 \cdot 5 \cdot 8 = 240$ multiplications whereas computing $(M_1 M_2)M_3$ takes only $4 \cdot 5 \cdot 2 + 4 \cdot 2 \cdot 8 = 104$ multiplications.

Exercise 222 (Minimum Edit Distance). The *minimum edit distance* (or *Levenshtein distance*) $L(s, t)$ between two strings s and t is the minimum number of character deletions, insertions, and replacements applied to s that produces string t. For example, $L(\texttt{graph}, \texttt{group}) = 3$. (delete `h`, replace `a` by `o`, insert `h` before `p`). Define $d(i, j) = L(\langle s_1, \ldots, s_i\rangle, \langle t_1, \ldots, t_j\rangle)$. Show that

$$d(i, j) = \min\{d(i-1, j) + 1, d(i, j-1) + 1, d(i-1, j-1) + [s_i = t_j]\}$$

where $[s_i = t_j]$ is one if s_i is equal to t_j and is zero otherwise.

```
Function bbKnapsack((p_1,...,p_n),(w_1,...,w_n),M) : L
    assert p_1/w_1 ≥ p_2/w_2 ≥ ··· ≥ p_n/w_n        // assume input sorted by profit density
    x̂ = heuristicKnapsack((p_1,...,p_n),(w_1,...,w_n),M) : L        // best solution so far
    x : L                                                          // current partial solution
    recurse(1, M, 0)
    return x̂

    // Find solutions assuming x_1,...,x_{i-1} are fixed, M' = M − Σ_{k<i} x_i w_i, P = Σ_{k<i} x_i p_i.
    Procedure recurse(i, M', P : ℕ)
        u := P + upperBound((p_i,...,p_n),(w_i,...,w_n),M')
        if u > p · x̂ then
            if i > n then x̂ := x
            else                                                   // Branch on variable x_i
                if w_i ≤ M' then x_i := 1; recurse(i + 1, M' − w_i, P + p_i)
                if u > p · x̂ then x_i := 0; recurse(i + 1, M', P)
```

Fig. 12.5. A branch-and-bound algorithm for the knapsack problem. A first feasible solution is constructed by Function $heuristicKnapsack$ using some heuristic algorithm. Function $upperBound$ computes an upper bound for the possible profit.

Exercise 223. Does the principle of optimality hold for minimum spanning trees? Check the following three possibilities for definitions of subproblems: subsets of nodes, arbitrary subsets of edges, and prefixes of the sorted sequence of edges.

Exercise 224 (Constrained Shortest Path). Consider a directed graph $G = (V, E)$ where edges $e \in E$ have a *length* $\ell(e)$ and a *cost* $c(e)$. We want to find a path from node s to node t that minimizes the total length subject to the constraint that the total cost of the path is at most C. Show that subpaths $\langle s', t' \rangle$ of optimal solutions are *not* necessarily shortest paths from s' to t'.

12.4 Systematic Search — If in Doubt, Use Brute Force

In many optimization problems, the universe $\mathcal{U}$ of possible solutions is finite so that we can in principle solve the optimization problem by trying all possibilities. Naive application of this idea does not lead very far. However, we can frequently restrict the search to *promising* candidates and then the concept carries a lot further.

We will explain the concept of systematic search using the knapsack problem and a specific approach to systematic search known as *branch-and-bound*. In Exercises 226 and 227 we outline systematic search routines following a somewhat different pattern.

Figure 12.5 gives pseudo-code for a systematic search routine $bbKnapsack$ for the knapsack problem. *Branching* is the most fundamental ingredient of systematic search routines. All sensible values for some piece of the solution are tried. For each of these values, the resulting problem is solved recursively. Within the recursive call,