Chapter 4

Dynamic programming

In this chapter, we focus on how to find optimal dynamic-programming algorithms. A particular attention is paid to problem size in order to avoid exponential-cost algorithms. The chapter is illustrated with two classical examples: the coin changing problem and the knapsack problem. These techniques are then further illustrated with a set of exercises in Section 4.4, with solutions found in Section 4.5.

4.1 The coin changing problem

The problem is the following: If we want to make change for S cents, and we have infinite supply of each coin in the set $Coins = \{v_1, v_2, \dots, v_n\}$, where v_i is the value of the i-th coin, what is the minimum number of coins required to reach the value S?

Greedy algorithm. We propose a greedy algorithm to solve the problem. First, we sort coins by nonincreasing values, then for each coin value we take as many coins as possible. The algorithm is formalized as Algorithm 4.1.

```
1 Sort elements of $Coins = \{v_1, \dots, v_n\}$ by nonincreasing values:
  $v_1 \geqslant v_2 \geqslant \cdots \geqslant v_n$
2 $R \leftarrow S$        { $R$ is the remaining sum to reach; it is initially $S$ }
3 for $i = 1$ to $n$ do
4     $c_i = \lfloor \frac{R}{v_i} \rfloor$        { $c_i$ is the number of coins of value $v_i$ that are taken }
5     $R \leftarrow R - c_i \times v_i$      { $R$ is updated }
```

ALGORITHM 4.1: Greedy algorithm for the coin changing problem.

We first assume that $Coins = \{10, 5, 2, 1\}$ (a typical European set of coins). In this case, we can prove that Algorithm 4.1 is optimal:

- An optimal solution returns, at most, one coin of value 5 (if there are two, it is better to use one single coin of value 10).
- An optimal solution returns, at most, one coin of value 1 (otherwise, we can use a coin of value 2).
- An optimal solution returns, at most, two coins of value 2 (otherwise, to obtain $6 = 2 + 2 + 2$, we would rather use $6 = 5 + 1$: one coin 5 and one coin 1).

Therefore, in the optimal solution, there cannot be more than four coins that are not of value 10, and $5 + 2 + 2 + 1 = 10$, so if there are four such coins, we would rather use a coin of 10. Thus, the optimal solution uses, at most, three coins that are not of value 10, and their total is at most 9. We can then conclude that the optimal number of coins of value 10 is $\lfloor \frac{S}{10} \rfloor$, which is the number selected by the greedy algorithm. It is then easy to conclude that the greedy algorithm always selects the optimal number of coins of each value.

Note, however, that the greedy algorithm is not optimal for any set of coins. For instance, if $Coins = \{6, 4, 1\}$ and $S = 8$, the greedy algorithm requires three coins $8 = 6 + 1 + 1$, while the optimal solution requires two coins of value 4. Still, U.S. readers will be pleased to know that the greedy algorithm is optimal for the set $Coins = \{25, 10, 5, 1\}$. The proof follows an ad hoc case analysis very similar to that conducted for European coins. Because the greedy algorithm is not always optimal, we explore another idea to solve the problem.

An optimal algorithm. The problem is to find the minimum number of coins required to reach sum S, with coins of value $\{v_1, \ldots, v_n\}$, which we denote as $z(S, n)$. Because the greedy algorithm may fail, we try to solve more subproblems so that we do not take a bad greedy choice as we did in the previous example. We also allow ourselves to come back to a choice already made and try another set of coins.

We investigate a way to solve the problem that is in appearance more complex than the initial problem. In other words, we artificially ask for more than requested, and aim at finding $z(T, i)$, the minimum number of coins required to reach sum $T \leqslant S$ with the first i coins, i.e., coins selected from the subset $\{v_1, \ldots, v_i\}$ (where $0 \leqslant i \leqslant n$). Instead of computing only $z(S, n)$, the original problem, we compute $S \times n$ values $z(T, i)$. But now, we have a recurrence relation to compute $z(T, i)$:

$$z(T, i) = \min \begin{cases} z(T, i-1) & i\text{-th coin not used;} \\ z(T - v_i, i) + 1 & i\text{-th coin used (at least) once.} \end{cases}$$

The recurrence must be properly initialized; values of i and T are decreasing, so we consider the cases $i = 0$ and $T \leqslant 0$:

- $z(T, 0) = +\infty$ for $T > 0$: There are no more coins, therefore, we cannot reach the sum $T > 0$ and this solution cannot be correct.
- $z(0, i) = 0$: We do not need any coin to reach the sum $T = 0$.

- $z(T,i) = +\infty$ for $T < 0$: We have exceeded the sum; this solution cannot be correct.

Thanks to the recurrence relation and the initialization conditions, we are now able to compute $z(S,n)$ and to solve the original problem. This kind of algorithm is called a *dynamic-programming* algorithm.

If the recurrence is applied without memoizing which values have already been computed, using a recursive algorithm, there will be an exponential number of computations. Note that the word *memoization* comes from "memo": the idea consists of memoizing the values so that we can look them up later.

However, we need to compute only $S \times n$ values of the function $z(T,i)$ ($1 \leqslant T \leqslant S$ and $1 \leqslant i \leqslant n$). This can be done either recursively, by memoizing the values that have already been computed, or iteratively, with, for instance, a loop with increasing i and then a loop with increasing T, so that we always have the values required to compute $z(T,i)$, i.e., $z(T,i-1)$ and $z(T-v_i,i)$, as shown in Algorithm 4.2. The precedence constraints are shown in Figure 4.1, and they are always enforced with this algorithm (for details about precedence constraints, see Section 6.4.4, p. 140). Note that we ensure that we never call the function with $T < 0$, and, therefore, we do not need the third initialization condition.

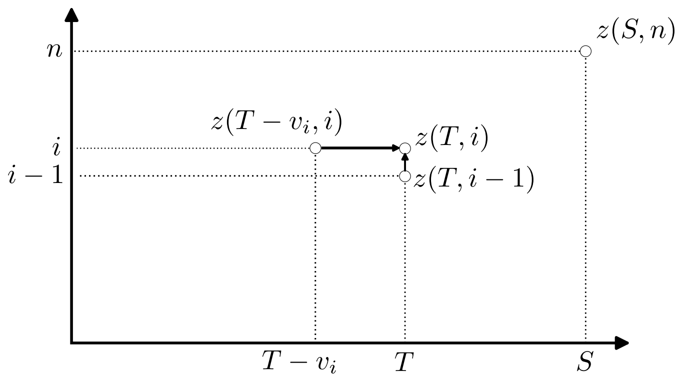

FIGURE 4.1: Precedence constraints for the coin-changing dynamic-programming algorithm.

The complexity of the dynamic-programming algorithm is $O(n \times S)$, while the greedy algorithm has a complexity in $O(n \log n)$ (the execution is linear, but sorting the coins requires a time in $O(n \log n)$).

Finally, note that characterizing the set of coins for which the greedy algorithm is optimal is still an open problem. It is easy to find sets that work. For instance, coins $\{1, B, B^2, B^3, \ldots\}$ with $B \geqslant 2$. However, the general case seems tricky. There are several variants of the coin changing problem, and

```
for $T = 1$ to $S$ do
    $z(T, 0) \leftarrow +\infty$    { Initialization: case $i = 0$ }
for $i = 0$ to $n$ do
    $z(0, i) \leftarrow 0$    { Initialization: case $T = 0$ }
for $i = 1$ to $n$ do
    for $T = 1$ to $S$ do
        $z(T, i) \leftarrow z(T, i-1)$
            { $z(T, i-1)$ computed at previous iteration, or case $i = 0$ }
        if $T - v_i \geqslant 0$ then
            $z(T, i) \leftarrow \min(z(T, i), z(T - v_i, i))$
                { $z(T - v_i, i)$ computed earlier in this loop, or case $T = 0$ }
```

ALGORITHM 4.2: Dynamic-programming algorithm for the coin changing problem.

many dynamic-programming algorithms to solve them. The interested reader may refer to the following papers: [85, 98]. We move in the next section to another classical problem: The knapsack problem.

4.2 The knapsack problem

We have a set of items, each with a weight and a value, and we want to determine the items to include in the collection so that the total weight does not exceed a given limit, and the total value is as large as possible. Formally, there are n items $I_1, \dots, I_n$, and item I_i has a weight w_i and a value c_i ($1 \leqslant i \leqslant n$). We are also given a maximum total weight W. The goal is to find a subset K of $\{1, \dots, n\}$ that maximizes $\sum_{i \in K} c_i$, under the constraint $\sum_{i \in K} w_i \leqslant W$. The analogy with the problem of packing the best items for a well-deserved vacation should be clear.

Greedy algorithm. Here again, we start by designing a greedy algorithm to solve the problem. The idea consists in selecting first those items that have a good value per unit of weight, $\frac{c_i}{w_i}$. Therefore, we sort items by nonincreasing $\frac{c_i}{w_i}$, and then we greedily add them in the knapsack as long as the total weight is not exceeded.

However, the algorithm is not optimal because items are not divisible. We cannot take only a fraction of an item, i.e., either we take it or we discard it. A counterexample for the greedy algorithm can be designed as follows, with three items. The first item with the greatest ratio c_1/w_1 is such that it fills up the knapsack by itself (no other item can fit in the knapsack once I_1 has been chosen, i.e., $w_1 + w_i > W$, for $i \geqslant 2$). Then, two more items are such that $w_2 + w_3 \leqslant W$ (they fit together in the knapsack), and $c_2 + c_3 > c_1$ (they have more value than the first item alone). If we are able to construct such an example, the greedy algorithm chooses the first item, while a better solution consists in choosing items 2 and 3. A possible set of items is the following, with $W = 10$: $(w_1 = 6, w_2 = 5, w_3 = 5)$ and $(c_1 = 7, c_2 = 5, c_3 = 5)$.

If we consider the problem of the fractional knapsack, in which it is possible to take only a fraction of an object, then the greedy algorithm is optimal. In the example, it would take the whole item 1, and then a fraction (4/5) of item 2 to fill the remaining space in the knapsack. The value would then be $c_1 + \frac{4}{5}c_2 = 7 + 4 = 11$, which is optimal. It is easy to prove the optimality of the greedy algorithm in this case. If an optimal solution is not making the greedy choice, we can always exchange a fraction of item of the optimal solution with the fraction of item of better value per weight unit that was not greedily chosen, and the total value can only increase.

Dynamic-programming algorithm. We come back to the integer knapsack problem, and since the greedy algorithm is not optimal, we try to solve a more complex problem, as in the coin changing problem, in order to be able to establish a recurrence. The two parameters are the total weight and the number of items considered. We want to compute $C(v, i)$, which is the maximum value that can be obtained when filling up a knapsack of maximum total weight v, using only some of the first i items $\{I_1, \dots, I_i\}$. The original problem is the value $C(W, n)$: The knapsack is of maximum total weight W, and we have the n items at our disposal.

To write the recurrence, we have two choices: (1) either we have chosen the last object, or (2) we have not, therefore leading to:

$$C(v,i) = \max \begin{cases} C(v, i-1) & \text{last object not chosen;} \\ C(v - w_i, i-1) + c_i & \text{last object chosen;} \end{cases}$$

with the initialization conditions:

- $C(v, i) = 0$ for $v = 0$ or $i = 0$;
- $C(v, i) = -\infty$ if $v < 0$ (capacity exceeded).

The optimal solutions of all subproblems that we solve allow us to compute the optimal solution of the original problem. Similarly to the coin changing problem, we need to carefully respect the precedence constraints of the computations, and we want to never compute twice the same value of the function $C(v, i)$. The algorithm is formalized in Algorithm 4.3. The precedence constraints are shown in Figure 4.2. Because the computation is done row by row, these constraints are always respected.

The complexity of the greedy algorithm is in $O(n \log n)$, because the n items must be sorted. However, the complexity of the dynamic programming algo-

```
for $i = 0$ to $n$ do
    $C(0, i) \leftarrow 0$    { Initialization: case $v = 0$ }
for $v = 1$ to $W$ do
    $C(v, 0) \leftarrow 0$    { Initialization: case $i = 0$ }
for $i = 1$ to $n$ do
    for $v = 1$ to $W$ do
        $C(v, i) \leftarrow C(v, i-1)$
        if $v - w_i \geqslant 0$ then
            $C(v, i) \leftarrow \max(C(v, i), C(v - w_i, i) + c_i)$
```

ALGORITHM 4.3: dynamic-programming algorithm for the knapsack problem.

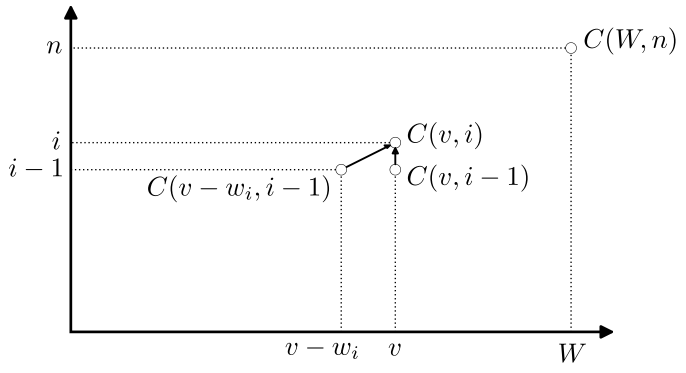

FIGURE 4.2: Precedence constraints for the knapsack dynamic-programming algorithm.

rithm is in $O(n \times W)$, because we need to compute $n \times W$ values of the function $C(v, i)$, and each computation takes constant time.

4.3 Designing dynamic-programming algorithms

In the previous two sections, we have given examples of dynamic-programming algorithms. The basic reasoning to obtain the optimal algorithm is similar in both cases:

1. Identify subproblems whose optimal solutions can be used to build an optimal solution to the original problem. Conversely, given an optimal

solution to the original problem, identify subparts of the solution that are optimal solutions for some subproblems. Usually, this step means that we identify a more complex problem derived from the original problem.

2. Write the recurrence.

3. Write the initial cases.

4. Write the algorithm, usually as an iterative algorithm, and taking care to enforce precedence constraints (use a figure to check that these constraints are indeed satisfied). A recursive algorithm may be used, but it requires tests to avoid redundant computations.

5. Study the complexity of the algorithm (usually straightforward from the iterative version of the algorithm).

Such an algorithm is *bottom-up*; we need results of the multiple subproblems to make a choice and compute the optimal solution, while the greedy algorithms were *top-down*, making a local choice at each step.

With dynamic-programming algorithms, one must be particularly cautious about the size of the data. It is not unusual to write nonpolynomial dynamic-programming algorithms. For instance, in the knapsack problem, the cost of the dynamic-programming algorithm is $O(nW)$. However, data can be encoded in $\sum_{i=1}^{n} \log w_i + \sum_{i=1}^{n} \log c_i \leqslant n(\log W + \log C)$, which means that W is in fact exponential in the problem size. This important encoding issue is related to weak NP-completeness and pseudo-polynomial algorithms, which we come back to in Section 6.6, p. 145.

4.4 Exercises

Exercise 4.1: Matrix chains (solution p. 90)

Consider n matrices $A_1, \dots, A_n$, where A_i is of size $P_{i-1} \times P_i$ $(1 \leqslant i \leqslant n)$. We want to compute $A_1 \times A_2 \times \cdots \times A_n$. The problem is to decide in which order the multiplications should be done and, therefore, to add parentheses to the expression, in order to minimize the number of operations. Note that it costs $P_a \times P_b \times P_c$ to multiply a matrix of size $P_a \times P_b$ by a matrix of size $P_b \times P_c$.

Propose a dynamic-programming algorithm to solve the problem and give its complexity. Be careful to define the initial conditions and the recurrence.