

# Chapter 7

## Dynamic programming

**Dynamic programming** is a technique that combines the correctness of complete search and the efficiency of greedy algorithms. Dynamic programming can be applied if the problem can be divided into overlapping subproblems that can be solved independently.

There are two uses for dynamic programming:

- **Finding an optimal solution:** We want to find a solution that is as large as possible or as small as possible.
- **Counting the number of solutions:** We want to calculate the total number of possible solutions.

We will first see how dynamic programming can be used to find an optimal solution, and then we will use the same idea for counting the solutions.

Understanding dynamic programming is a milestone in every competitive programmer's career. While the basic idea is simple, the challenge is how to apply dynamic programming to different problems. This chapter introduces a set of classic problems that are a good starting point.

### Coin problem

We first focus on a problem that we have already seen in Chapter 6: Given a set of coin values  $\text{coins} = \{c_1, c_2, \dots, c_k\}$  and a target sum of money  $n$ , our task is to form the sum  $n$  using as few coins as possible.

In Chapter 6, we solved the problem using a greedy algorithm that always chooses the largest possible coin. The greedy algorithm works, for example, when the coins are the euro coins, but in the general case the greedy algorithm does not necessarily produce an optimal solution.

Now is time to solve the problem efficiently using dynamic programming, so that the algorithm works for any coin set. The dynamic programming algorithm is based on a recursive function that goes through all possibilities how to form the sum, like a brute force algorithm. However, the dynamic programming algorithm is efficient because it uses *memoization* and calculates the answer to each subproblem only once.

## Recursive formulation

The idea in dynamic programming is to formulate the problem recursively so that the solution to the problem can be calculated from solutions to smaller subproblems. In the coin problem, a natural recursive problem is as follows: what is the smallest number of coins required to form a sum  $x$ ?

Let  $\text{solve}(x)$  denote the minimum number of coins required for a sum  $x$ . The values of the function depend on the values of the coins. For example, if  $\text{coins} = \{1, 3, 4\}$ , the first values of the function are as follows:

$\text{solve}(0)$	$=$	0
$\text{solve}(1)$	$=$	1
$\text{solve}(2)$	$=$	2
$\text{solve}(3)$	$=$	1
$\text{solve}(4)$	$=$	1
$\text{solve}(5)$	$=$	2
$\text{solve}(6)$	$=$	2
$\text{solve}(7)$	$=$	2
$\text{solve}(8)$	$=$	2
$\text{solve}(9)$	$=$	3
$\text{solve}(10)$	$=$	3

For example,  $\text{solve}(10) = 3$ , because at least 3 coins are needed to form the sum 10. The optimal solution is  $3 + 3 + 4 = 10$ .

The essential property of  $\text{solve}$  is that its values can be recursively calculated from its smaller values. The idea is to focus on the *first* coin that we choose for the sum. For example, in the above scenario, the first coin can be either 1, 3 or 4. If we first choose coin 1, the remaining task is to form the sum 9 using the minimum number of coins, which is a subproblem of the original problem. Of course, the same applies to coins 3 and 4. Thus, we can use the following recursive formula to calculate the minimum number of coins:

$$\begin{aligned}\text{solve}(x) = \min(&\text{solve}(x-1) + 1, \\ &\text{solve}(x-3) + 1, \\ &\text{solve}(x-4) + 1).\end{aligned}$$

The base case of the recursion is  $\text{solve}(0) = 0$ , because no coins are needed to form an empty sum. For example,

$$\text{solve}(10) = \text{solve}(7) + 1 = \text{solve}(4) + 2 = \text{solve}(0) + 3 = 3.$$

Now we are ready to give a general recursive function that calculates the minimum number of coins needed to form a sum  $x$ :

$$\text{solve}(x) = \begin{cases} \infty & x < 0 \\ 0 & x = 0 \\ \min_{c \in \text{coins}} \text{solve}(x - c) + 1 & x > 0 \end{cases}$$

First, if  $x < 0$ , the value is  $\infty$ , because it is impossible to form a negative sum of money. Then, if  $x = 0$ , the value is 0, because no coins are needed to form an

empty sum. Finally, if  $x > 0$ , the variable  $c$  goes through all possibilities how to choose the first coin of the sum.

Once a recursive function that solves the problem has been found, we can directly implement a solution in C++ (the constant INF denotes infinity):

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    return best;
}
```

Still, this function is not efficient, because there may be an exponential number of ways to construct the sum. However, next we will see how to make the function efficient using a technique called memoization.

## Using memoization

The idea of dynamic programming is to use **memoization** to efficiently calculate values of a recursive function. This means that the values of the function are stored in an array after calculating them. For each parameter, the value of the function is calculated recursively only once, and after this, the value can be directly retrieved from the array.

In this problem, we use arrays

```
bool ready[N];
int value[N];
```

where  $ready[x]$  indicates whether the value of  $solve(x)$  has been calculated, and if it is,  $value[x]$  contains this value. The constant  $N$  has been chosen so that all required values fit in the arrays.

Now the function can be efficiently implemented as follows:

```
int solve(int x) {
    if (x < 0) return INF;
    if (x == 0) return 0;
    if (ready[x]) return value[x];
    int best = INF;
    for (auto c : coins) {
        best = min(best, solve(x-c)+1);
    }
    value[x] = best;
    ready[x] = true;
    return best;
}
```

The function handles the base cases  $x < 0$  and  $x = 0$  as previously. Then the function checks from `ready[x]` if `solve(x)` has already been stored in `value[x]`, and if it is, the function directly returns it. Otherwise the function calculates the value of `solve(x)` recursively and stores it in `value[x]`.

This function works efficiently, because the answer for each parameter  $x$  is calculated recursively only once. After a value of `solve(x)` has been stored in `value[x]`, it can be efficiently retrieved whenever the function will be called again with the parameter  $x$ . The time complexity of the algorithm is  $O(nk)$ , where  $n$  is the target sum and  $k$  is the number of coins.

Note that we can also *iteratively* construct the array `value` using a loop that simply calculates all the values of `solve` for parameters  $0 \dots n$ :

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0) {
            value[x] = min(value[x], value[x-c]+1);
        }
    }
}
```

In fact, most competitive programmers prefer this implementation, because it is shorter and has lower constant factors. From now on, we also use iterative implementations in our examples. Still, it is often easier to think about dynamic programming solutions in terms of recursive functions.

## Constructing a solution

Sometimes we are asked both to find the value of an optimal solution and to give an example how such a solution can be constructed. In the coin problem, for example, we can declare another array that indicates for each sum of money the first coin in an optimal solution:

```
int first[N];
```

Then, we can modify the algorithm as follows:

```
value[0] = 0;
for (int x = 1; x <= n; x++) {
    value[x] = INF;
    for (auto c : coins) {
        if (x-c >= 0 && value[x-c]+1 < value[x]) {
            value[x] = value[x-c]+1;
            first[x] = c;
        }
    }
}
```

After this, the following code can be used to print the coins that appear in an optimal solution for the sum  $n$ :

```
while (n > 0) {
    cout << first[n] << "\n";
    n -= first[n];
}
```

## Counting the number of solutions

Let us now consider another version of the coin problem where our task is to calculate the total number of ways to produce a sum  $x$  using the coins. For example, if  $\text{coins} = \{1, 3, 4\}$  and  $x = 5$ , there are a total of 6 ways:

- $1 + 1 + 1 + 1 + 1$
- $1 + 1 + 3$
- $1 + 3 + 1$
- $3 + 1 + 1$
- $1 + 4$
- $4 + 1$

Again, we can solve the problem recursively. Let  $\text{solve}(x)$  denote the number of ways we can form the sum  $x$ . For example, if  $\text{coins} = \{1, 3, 4\}$ , then  $\text{solve}(5) = 6$  and the recursive formula is

$$\begin{aligned} \text{solve}(x) = & \text{solve}(x-1) + \\ & \text{solve}(x-3) + \\ & \text{solve}(x-4). \end{aligned}$$

Then, the general recursive function is as follows:

$$\text{solve}(x) = \begin{cases} 0 & x < 0 \\ 1 & x = 0 \\ \sum_{c \in \text{coins}} \text{solve}(x-c) & x > 0 \end{cases}$$

If  $x < 0$ , the value is 0, because there are no solutions. If  $x = 0$ , the value is 1, because there is only one way to form an empty sum. Otherwise we calculate the sum of all values of the form  $\text{solve}(x-c)$  where  $c$  is in coins.

The following code constructs an array  $\text{count}$  such that  $\text{count}[x]$  equals the value of  $\text{solve}(x)$  for  $0 \leq x \leq n$ :

```
count[0] = 1;
for (int x = 1; x <= n; x++) {
    for (auto c : coins) {
        if (x-c >= 0) {
            count[x] += count[x-c];
        }
    }
}
```

Often the number of solutions is so large that it is not required to calculate the exact number but it is enough to give the answer modulo  $m$  where, for example,  $m = 10^9 + 7$ . This can be done by changing the code so that all calculations are done modulo  $m$ . In the above code, it suffices to add the line

```
count[x] %= m;
```

after the line

```
count[x] += count[x-c];
```

Now we have discussed all basic ideas of dynamic programming. Since dynamic programming can be used in many different situations, we will now go through a set of problems that show further examples about the possibilities of dynamic programming.


## Longest increasing subsequence

Our first problem is to find the **longest increasing subsequence** in an array of  $n$  elements. This is a maximum-length sequence of array elements that goes from left to right, and each element in the sequence is larger than the previous element. For example, in the array

0	1	2	3	4	5	6	7
6	2	5	1	7	4	8	3

the longest increasing subsequence contains 4 elements:

0	1	2	3	4	5	6	7
6	2	5	1	7	4	8	3



Let  $\text{length}(k)$  denote the length of the longest increasing subsequence that ends at position  $k$ . Thus, if we calculate all values of  $\text{length}(k)$  where  $0 \leq k \leq n-1$ , we will find out the length of the longest increasing subsequence. For example, the values of the function for the above array are as follows:

```
length(0) = 1
length(1) = 1
length(2) = 2
length(3) = 1
length(4) = 3
length(5) = 2
length(6) = 4
length(7) = 2
```

For example,  $\text{length}(6) = 4$ , because the longest increasing subsequence that ends at position 6 consists of 4 elements.

To calculate a value of  $\text{length}(k)$ , we should find a position  $i < k$  for which  $\text{array}[i] < \text{array}[k]$  and  $\text{length}(i)$  is as large as possible. Then we know that  $\text{length}(k) = \text{length}(i) + 1$ , because this is an optimal way to add  $\text{array}[k]$  to a subsequence. However, if there is no such position  $i$ , then  $\text{length}(k) = 1$ , which means that the subsequence only contains  $\text{array}[k]$ .

Since all values of the function can be calculated from its smaller values, we can use dynamic programming. In the following code, the values of the function will be stored in an array `length`.

```
for (int k = 0; k < n; k++) {
    length[k] = 1;
    for (int i = 0; i < k; i++) {
        if (array[i] < array[k]) {
            length[k] = max(length[k], length[i]+1);
        }
    }
}
```

This code works in  $O(n^2)$  time, because it consists of two nested loops. However, it is also possible to implement the dynamic programming calculation more efficiently in  $O(n \log n)$  time. Can you find a way to do this?

## Paths in a grid

Our next problem is to find a path from the upper-left corner to the lower-right corner of an  $n \times n$  grid, such that we only move down and right. Each square contains a positive integer, and the path should be constructed so that the sum of the values along the path is as large as possible.

The following picture shows an optimal path in a grid:

3	7	9	2	7
9	8	3	5	5
1	7	9	8	5
3	8	6	4	10
6	3	9	7	8

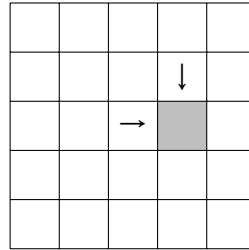
The sum of the values on the path is 67, and this is the largest possible sum on a path from the upper-left corner to the lower-right corner.

Assume that the rows and columns of the grid are numbered from 1 to  $n$ , and  $\text{value}[y][x]$  equals the value of square  $(y, x)$ . Let  $\text{sum}(y, x)$  denote the maximum sum on a path from the upper-left corner to square  $(y, x)$ . Now  $\text{sum}(n, n)$  tells us the maximum sum from the upper-left corner to the lower-right corner. For example, in the above grid,  $\text{sum}(5, 5) = 67$ .

We can recursively calculate the sums as follows:

$$\text{sum}(y, x) = \max(\text{sum}(y, x-1), \text{sum}(y-1, x)) + \text{value}[y][x]$$

The recursive formula is based on the observation that a path that ends at square  $(y, x)$  can come either from square  $(y, x - 1)$  or square  $(y - 1, x)$ :



Thus, we select the direction that maximizes the sum. We assume that  $\text{sum}(y, x) = 0$  if  $y = 0$  or  $x = 0$  (because no such paths exist), so the recursive formula also works when  $y = 1$  or  $x = 1$ .

Since the function  $\text{sum}$  has two parameters, the dynamic programming array also has two dimensions. For example, we can use an array

```
int sum[N][N];
```

and calculate the sums as follows:

```
for (int y = 1; y <= n; y++) {
    for (int x = 1; x <= n; x++) {
        sum[y][x] = max(sum[y][x-1], sum[y-1][x]) + value[y][x];
    }
}
```

The time complexity of the algorithm is  $O(n^2)$ .

## Knapsack problems

The term **knapsack** refers to problems where a set of objects is given, and subsets with some properties have to be found. Knapsack problems can often be solved using dynamic programming.

In this section, we focus on the following problem: Given a list of weights  $[w_1, w_2, \dots, w_n]$ , determine all sums that can be constructed using the weights. For example, if the weights are  $[1, 3, 3, 5]$ , the following sums are possible:

0	1	2	3	4	5	6	7	8	9	10	11	12
X	X		X	X	X	X	X	X	X		X	X

In this case, all sums between  $0 \dots 12$  are possible, except 2 and 10. For example, the sum 7 is possible because we can select the weights  $[1, 3, 3]$ .

To solve the problem, we focus on subproblems where we only use the first  $k$  weights to construct sums. Let  $\text{possible}(x, k) = \text{true}$  if we can construct a sum  $x$  using the first  $k$  weights, and otherwise  $\text{possible}(x, k) = \text{false}$ . The values of the function can be recursively calculated as follows:

$$\text{possible}(x, k) = \text{possible}(x - w_k, k - 1) \vee \text{possible}(x, k - 1)$$



The formula is based on the fact that we can either use or not use the weight  $w_k$  in the sum. If we use  $w_k$ , the remaining task is to form the sum  $x - w_k$  using the first  $k - 1$  weights, and if we do not use  $w_k$ , the remaining task is to form the sum  $x$  using the first  $k - 1$  weights. As the base cases,

$$\text{possible}(x,0) = \begin{cases} \text{true} & x = 0 \\ \text{false} & x \neq 0 \end{cases}$$

because if no weights are used, we can only form the sum 0.

The following table shows all values of the function for the weights [1,3,3,5] (the symbol "X" indicates the true values):

$k \backslash x$	0	1	2	3	4	5	6	7	8	9	10	11	12
0	X												
1	X	X											
2	X	X		X	X								
3	X	X		X	X		X	X					
4	X	X		X	X	X	X	X	X	X		X	X

After calculating those values,  $\text{possible}(x,n)$  tells us whether we can construct a sum  $x$  using *all* weights.

Let  $W$  denote the total sum of the weights. The following  $O(nW)$  time dynamic programming solution corresponds to the recursive function:

```
possible[0][0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = 0; x <= W; x++) {
        if (x-w[k] >= 0) possible[x][k] |= possible[x-w[k]][k-1];
        possible[x][k] |= possible[x][k-1];
    }
}
```

However, here is a better implementation that only uses a one-dimensional array  $\text{possible}[x]$  that indicates whether we can construct a subset with sum  $x$ . The trick is to update the array from right to left for each new weight:

```
possible[0] = true;
for (int k = 1; k <= n; k++) {
    for (int x = W; x >= 0; x--) {
        if (possible[x]) possible[x+w[k]] = true;
    }
}
```

Note that the general idea presented here can be used in many knapsack problems. For example, if we are given objects with weights and values, we can determine for each weight sum the maximum value sum of a subset.

## Edit distance

The **edit distance** or **Levenshtein distance**<sup>1</sup> is the minimum number of editing operations needed to transform a string into another string. The allowed editing operations are as follows:

- insert a character (e.g. ABC  $\rightarrow$  ABCA)
- remove a character (e.g. ABC  $\rightarrow$  AC)
- modify a character (e.g. ABC  $\rightarrow$  ADC)

For example, the edit distance between LOVE and MOVIE is 2, because we can first perform the operation LOVE  $\rightarrow$  MOVE (modify) and then the operation MOVE  $\rightarrow$  MOVIE (insert). This is the smallest possible number of operations, because it is clear that only one operation is not enough.

Suppose that we are given a string  $x$  of length  $n$  and a string  $y$  of length  $m$ , and we want to calculate the edit distance between  $x$  and  $y$ . To solve the problem, we define a function  $\text{distance}(a, b)$  that gives the edit distance between prefixes  $x[0 \dots a]$  and  $y[0 \dots b]$ . Thus, using this function, the edit distance between  $x$  and  $y$  equals  $\text{distance}(n-1, m-1)$ .

We can calculate values of distance as follows:

$$\begin{aligned} \text{distance}(a, b) = \min(&\text{distance}(a, b-1) + 1, \\ &\text{distance}(a-1, b) + 1, \\ &\text{distance}(a-1, b-1) + \text{cost}(a, b)). \end{aligned}$$

Here  $\text{cost}(a, b) = 0$  if  $x[a] = y[b]$ , and otherwise  $\text{cost}(a, b) = 1$ . The formula considers the following ways to edit the string  $x$ :

- $\text{distance}(a, b-1)$ : insert a character at the end of  $x$
- $\text{distance}(a-1, b)$ : remove the last character from  $x$
- $\text{distance}(a-1, b-1)$ : match or modify the last character of  $x$

In the two first cases, one editing operation is needed (insert or remove). In the last case, if  $x[a] = y[b]$ , we can match the last characters without editing, and otherwise one editing operation is needed (modify).

The following table shows the values of distance in the example case:

		M	O	V	I	E
L O V E	0	1	2	3	4	5
	1	1	2	3	4	5
	2	2	1	2	3	4
	3	3	2	1	2	3
	4	4	3	2	2	2

<sup>1</sup>The distance is named after V. I. Levenshtein who studied it in connection with binary codes [49].

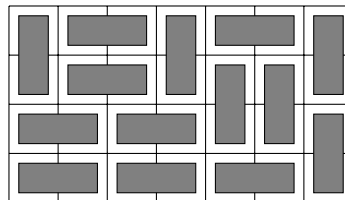
The lower-right corner of the table tells us that the edit distance between LOVE and MOVIE is 2. The table also shows how to construct the shortest sequence of editing operations. In this case the path is as follows:

		M	O	V	I	E
L	0	1	2	3	4	5
O	1	1	2	3	4	5
V	2	2	1	2	3	4
E	3	3	2	1	2	3
	4	4	3	2	2	2

The last characters of LOVE and MOVIE are equal, so the edit distance between them equals the edit distance between LOV and MOVI. We can use one editing operation to remove the character I from MOVI. Thus, the edit distance is one larger than the edit distance between LOV and MOV, etc.

## Counting tilings

Sometimes the states of a dynamic programming solution are more complex than fixed combinations of numbers. As an example, consider the problem of calculating the number of distinct ways to fill an  $n \times m$  grid using  $1 \times 2$  and  $2 \times 1$  size tiles. For example, one valid solution for the  $4 \times 7$  grid is



and the total number of solutions is 781.

The problem can be solved using dynamic programming by going through the grid row by row. Each row in a solution can be represented as a string that contains  $m$  characters from the set  $\{\sqcap, \sqcup, \sqsubset, \sqsupset\}$ . For example, the above solution consists of four rows that correspond to the following strings:

- $\sqcap \sqsubset \sqcap \sqsubset \sqcap \sqsubset \sqcap$
- $\sqcup \sqsubset \sqcup \sqcap \sqcap \sqcup \sqcup$
- $\sqsubset \sqsubset \sqsubset \sqcup \sqcup \sqcap \sqcap$
- $\sqsubset \sqsubset \sqsubset \sqsubset \sqsubset \sqcup \sqcup$

Let  $\text{count}(k, x)$  denote the number of ways to construct a solution for rows  $1 \dots k$  of the grid such that string  $x$  corresponds to row  $k$ . It is possible to use dynamic programming here, because the state of a row is constrained only by the state of the previous row.

A solution is valid if row 1 does not contain the character  $\sqcup$ , row  $n$  does not contain the character  $\sqcap$ , and all consecutive rows are *compatible*. For example, the rows  $\sqcup \sqcup \sqcup \sqcup \sqcap \sqcap \sqcup$  and  $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcap$  are compatible, while the rows  $\sqcap \sqcup \sqcup \sqcap \sqcup \sqcup \sqcap$  and  $\sqcup \sqcup \sqcup \sqcup \sqcup \sqcup \sqcup$  are not compatible.

Since a row consists of  $m$  characters and there are four choices for each character, the number of distinct rows is at most  $4^m$ . Thus, the time complexity of the solution is  $O(n4^{2m})$  because we can go through the  $O(4^m)$  possible states for each row, and for each state, there are  $O(4^m)$  possible states for the previous row. In practice, it is a good idea to rotate the grid so that the shorter side has length  $m$ , because the factor  $4^{2m}$  dominates the time complexity.

It is possible to make the solution more efficient by using a more compact representation for the rows. It turns out that it is sufficient to know which columns of the previous row contain the upper square of a vertical tile. Thus, we can represent a row using only characters  $\sqcap$  and  $\square$ , where  $\square$  is a combination of characters  $\sqcup$ ,  $\sqcap$  and  $\sqcup$ . Using this representation, there are only  $2^m$  distinct rows and the time complexity is  $O(n2^{2m})$ .

As a final note, there is also a surprising direct formula for calculating the number of tilings<sup>2</sup>:

$$\prod_{a=1}^{\lceil n/2 \rceil} \prod_{b=1}^{\lceil m/2 \rceil} 4 \cdot \left( \cos^2 \frac{\pi a}{n+1} + \cos^2 \frac{\pi b}{m+1} \right)$$

This formula is very efficient, because it calculates the number of tilings in  $O(nm)$  time, but since the answer is a product of real numbers, a problem when using the formula is how to store the intermediate results accurately.

---

<sup>2</sup>Surprisingly, this formula was discovered in 1961 by two research teams [43, 67] that worked independently.