10. Dynamic programming

Dynamic programming is an algorithmic technique that can be used for efficiently solving many search problems. In this chapter, we will learn about dynamic programming through the following problem:

Task

You have an unlimited number coins with values given as a list. Each value is a positive integer and the smallest value is 1. The goal is to choose a set of coins summing up to x.

1. *Find the optimal solution*: How many coins at least is needed to achieve the sum x?
2. *Construct an optimal solution*: Give an example of a minimal set of coins summing up x.
3. *Count solutions*: How many different ways there are to achieve the sum x ?

For example, when the coin values are $[1, 2, 5]$ and $x = 13$, the answers are:

1. The smallest number of coins needed is 4.
2. A minimal solution is to choose the coins $[1, 2, 5, 5]$.
3. There are 634 ways to choose the coins (including $[1, 2, 5, 5]$, $[2, 2, 2, 2, 5]$ and $[1, 1, 1, 5, 5]$).

In the previous chapter, we saw an efficient solution using a greedy algorithm in the case where the coin values are $[1, 2, 5]$. However, the greedy algorithm does not work in all cases: For example, the algorithm may give a wrong answer when the coin values are $[1, 4, 5]$. Moreover, the greedy algorithm cannot be used for solving the part 3 of the problem, where we want to count all solutions.

We will next look at how dynamic programming can be used for solving all parts efficiently. The ideas and techniques developed here are useful for solving many other problems too.

Finding the optimal solution

The basic idea of dynamic programming is to solve a problem with the help of smaller cases of the same problem, called *subproblems*. For example, when we want to achieve the sum x with coins, the subprolems are the cases where the target sum is $0 \ldots x-1$.

Consider the example, where the coin values are $[1,2,5]$ and the target sum is $x = 13$. We start the solution by choosing one of the coins. There are three options:

- Choose a coin of value 1. Then the rest of the coins must sum up to $x = 12$.
- Choose a coin of value 2. Then the rest of the coins must sum up to $x = 11$.
- Choose a coin of value 5. Then the rest of the coins must sum up to $x = 8$.

If we already know the smallest number of coins to achieve the sums $x = 8$, $x = 11$ and $x = 12$, we can find the smallest number of coins summing up to $x = 13$ by taking the minimum of the three cases and adding one. Thus, in order to solve the case $x = 13$, we need to first solve the three subproblems $x = 8$, $x = 11$ ja $x = 12$. These subproblems can be solved similarly by dividing them into even smaller subproblems.

We can turn this idea into a practical dynamic programming algorithm by solving all the subproblems $x = 0, \ldots, x$ from the smallest to largest:

```
def min_coins(x, coins):
    result = {}

    result[0] = 0
    for s in range(1, x + 1):
        result[s] = s
        for c in coins:
            if s - c >= 0:
                result[s] = min(result[s], result[s - c] + 1)

    return result[x]
```

The function defines a dictionary `result` that stores the smallest number of coins needed for each sum $0 \ldots x$. The first step is to record the fact that the sum

0 needs 0 coins. Then the algorithm goes through all the other sums in a loop. For each sum, the smallest number of coins is computed as a minimum over the possible ways of choosing one coin.

Notice that, since one of the coin values is 1, all the sums $0 \ldots x$ can be formed using the coins. The initial value of `result[s]` is set to `s` which corresponds to choosing only coins of value 1.

The function can be used as follows:

```
print(min_coins(13, [1, 2, 5])) # 4
print(min_coins(13, [1, 4, 5])) # 3
print(min_coins(42, [1, 5, 6, 17])) # 5
```

When $x = 13$ and the coin values are $[1, 2, 5]$, the function computes the following answers:

Sum x	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Smallest number of coins	0	1	1	2	2	1	2	2	3	3	2	3	3	4

This shows that the sum $x = 13$ needs 4 coins and all smaller sums need at most 3 coins.

When $x = 13$ and the coin values are $[1, 4, 5]$, the results are:

Sum x	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Smallest number of coins	0	1	2	3	1	1	2	3	2	2	2	3	3	3

In this case, the greedy algorithm would produce a wrong answer, but the dynamic programming algorithm is correct, because it goes through all the ways of choosing the coins.

The time complexity of the algorithm is $O(nx)$, where n is the number of coin values and x is the target sum. This enables efficient computation even in fairly big cases that would be far too big for a brute force algorithm.

Constructing an optimal solution

Next we will modify the dynamic programming algorithm so that it produces the

actual set of coins corresponding to the minimal solution. For example, when $x = 13$ and the coin values are $[1, 2, 5]$, the algorithm should return the solution $[1, 2, 5, 5]$ rather than just reporting that 4 coins is needed.

We can implement the algorithm by storing for each sum $0 \ldots x$ the smallest set of coins as a list. The length of each such list is the smallest number of coins needed for that sum. This can be implemented as follows:

```
def min_coins(x, coins):
    result = {}

    result[0] = []
    for s in range(1, x + 1):
        result[s] = [1] * s
        for c in coins:
            if s - c >= 0:
                new_result = result[s - c] + [c]
                if len(new_result) < len(result[s]):
                    result[s] = new_result

    return sorted(result[x])
```

Inside the inner loop, the algorithm constructs a list `new_result` from the list corresponding to the sum $s - c$. Appending a coin of value c to the subproblem list produces the shortest list that sums up to s and has c as the last coin. If this list is shorter than any of the previous lists, it becomes the new minimal list for the sum s.

The function can be used as follows:

```
print(min_coins(13, [1, 2, 5])) # [1, 2, 5, 5]
print(min_coins(13, [1, 4, 5])) # [4, 4, 5]
print(min_coins(42, [1, 5, 6, 17])) # [1, 1, 6, 17, 17]
```

Counting solutions

Dynamic programming can also be used for counting the number of all solutions. For example, when the coin values are $[1, 2, 5]$, there are 634 different ways to choose a list of coins with the sum $x = 13$. Here are some of the possible

solutions:

- $[1, 2, 5, 5]$
- $[2, 2, 2, 2, 5]$
- $[1, 1, 1, 5, 5]$
- $[2, 2, 2, 2, 2, 2, 1]$
- $[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]$

The number of solutions can be computed in a similar way as the minimal number of coins. The search can be implemented as follows:

```
def count_coins(x, coins):
    result = {}

    result[0] = 1
    for s in range(1, x + 1):
        result[s] = 0
        for coin in coins:
            if s - coin >= 0:
                result[s] += result[s - coin]

    return result[x]
```

The difference to the earlier function `min_coins` is that the function takes a sum instead of a minimum. In the case $x = 0$, the result is 1, because there is exactly one way to achieve the sum 0 (an empty list). In other cases, the function goes through all the ways of choosing the last coin, and adds up the results.

The function can be used as follows:

```
print(count_coins(13, [1, 2, 5])) # 634
print(count_coins(13, [1, 4, 5])) # 88
print(count_coins(42, [1, 5, 6, 17])) # 1103532
```

Notice that the function counts each different ordering of the list separately. For example, the lists $[1, 2, 5, 5]$, $[1, 5, 2, 5]$ and $[1, 5, 5, 2]$ count as three different ways even though they consist of the same set of coins. Counting each combination of coins only once is a more difficult problem, but it too can be solved efficiently using dynamic programming.

Example: Subsequences

Task

You are given a list of n integers. Your task is to find how long is the *longest increasing subsequence*, i.e., how many numbers can we choose from the list going from left to right so that each number is larger than the previous number.

For example, in the list $[4, 1, 5, 6, 3, 4, 1, 8]$, a longest increasing subsequence is $[1, 3, 4, 8]$ and its length is 4.

We can use dynamic programming to solve this problem by computing for each position in the list the length of the longest increasing subsequence ending at that position. For the above example, the lengths are:

List position	0	1	2	3	4	5	6	7
Length of subsequence	1	1	2	3	2	3	1	4

For example, a longest increasing subsequence ending at the position 5 is $[1, 3, 4]$ and its length is 3.

The following function `find_longest` computes the length of the longest increasing subsequence by dynamic programming:

```
def find_longest(items):
    result = {}

    max_len = 0
    for i in range(len(items)):
        result[i] = 1
        for j in range(i):
            if items[j] < items[i]:
                result[i] = max(result[i], result[j] + 1)
        max_len = max(max_len, result[i])

    return max_len
```

The function defines a dictionary `result` and fills it with the subsequence lengths for each position. The variable `i` goes through the list positions and the

variable `j` goes through all the possible choices for the position of the previous number in the subsequence. If that previous number is smaller than the current number, the subsequence ending at the position `j` can be extended with the current number. The variable `max_len` keeps track of the length of the longest subsequence encountered so far.

The function can be used as follows:

```
print(find_longest([4, 1, 5, 6, 3, 4, 1, 8])) # 4
```

The time complexity of the algorithm is $O(n^2)$, because it has two nested loops. Since the number of all possible subsequences is $O(2^n)$, this algorithm is much faster than a brute force algorithm that iterates through all subsequences.

Example: Balanced parenthesis

Sometimes turning an inefficient brute force solution into an efficient dynamic programming solution is surprisingly easy. This is the case with the following problem:

Task

A balanced parenthesis sequence is a string consisting of the characters `(` and `)` so that it corresponds to a valid mathematical expression. Your task is to count the number of balanced parenthesis sequences of length n.

For example, when $n = 6$, the desired answer is 5, because the possible balanced parenthesis sequences are `((()))`, `(())()`, `()(())`, `(()())` and `()()()`.

In the previous chapter, we saw the following brute force solution:

```
def count_sequences(n, d=0):
    if d < 0 or d > n:
        return 0
    if n == 0:
        return 1
    return count_sequences(n - 1, d + 1) + \
           count_sequences(n - 1, d - 1)
```

This algorithm can be changed into an efficient dynamic programming algorithm by storing the function results into a dictionary:

```
def count_sequences(n, d=0, result={}):
    if d < 0 or d > n:
        return 0
    if n == 0:
        return 1
    if (n, d) not in result:
        result[(n, d)] = count_sequences(n - 1, d + 1) + \
                         count_sequences(n - 1, d - 1)
    return result[(n, d)]
```

In this case, the dynamic programming subproblems are *pairs* of the form (n, d), where n is the number of parenthesis symbols that still needs to be added, and d is the parenthesis depth of the current sequence. The dictionary `result` contains the answers for all parameter combinations that have already been computed by earlier calls of the function. The answer is computed by recursion only if it has not been computed previously. Otherwise the answer is found in the dictionary.

The brute force function is usable only for small cases, but the dynamic programming solution supports much bigger cases. For example, we can count balanced parenthesis sequences of length $n = 100$:

```
print(count_sequences(100)) # 1978261657756160653623774456
```

Since each call of the function needs $O(1)$ time and the number of parameter combinations is $O(n^2)$, the time complexity of the algorithm is $O(n^2)$.

There is another way of solving this problem using dynamic programming so that the function takes only one parameter n:

```
def count_sequences(n, result={}):
    if n == 0:
        return 1
    if n not in result:
        count = 0
        for i in range(2, n + 1, 2):
            count += count_sequences(i - 2) * \
```

```
                    count_sequences(n - i)
        result[n] = count
    return result[n]
```

The idea is to go through all possible ways for the position of the closing paranthesis `)` matching the first opening opening parenthesis `(`. For each such choice, both the sequence between the pair of parenthesis and the sequence after the closing parenthesis must be balanced parenthesis sequences. The sequence count for the choice can be obtained as a product of the sequence counts for the between and after sequences.

In this algorithm, each function call takes $O(n)$ time and the number of parameter combinations is $O(n)$. Thus the time complexity of this algorithm too is $O(n^2)$, even though the algorithm logic is quite different.

Nested recursion

Implementing a dynamic programming algorithm with recursion can lead to deeply nested recursive calls, which can cause problems when executing the code. For example, the following code does not work as desired:

```
print(count_sequences(2000)) # ?
```

The execution of the code causes an error ”RecursionError: maximum recursion depth exceeded in comparison”, which means that there are too many nested recursive calls.

In Python, the limit for nested recursive calls is fairly small, for example 1000. However, the limit can be raised using the function `setrecursionlimit` in the module `sys` as follows:

```
import sys
sys.setrecursionlimit(5000)
```

Now above code can be executed successfully:

```
print(count_sequences(2000)) # 204610552146802169264251998299
```

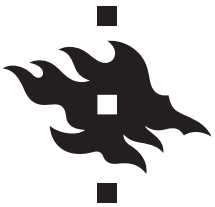
However, changing the recursion limit can be problematic, because it can cause issues with the Python execution environment. An alternative approach is to modify the dynamic programming implementation so that it uses loops instead of recursion. Consider the following recursive implementation:

```
def count_sequences(n, d=0, result={}):
    if d < 0 or d > n:
        return 0
    if n == 0:
        return 1
    if (n, d) not in result:
        result[(n, d)] = count_sequences(n - 1, d + 1) + \
                         count_sequences(n - 1, d - 1)
    return result[(n, d)]
```

We can remove the recursion by finding a suitable order for computing the subproblems so that when a subproblem result is needed, it has already been computed and stored in the dictionary `result`. In this case, we can implement the function as follows:

```
def count_sequences(n):
    result = {}
    result[(0, 0)] = 1
    for i in range(1, n + 1):
        result[(0, i)] = 0
    for i in range(1, n + 1):
        for j in range(0, n + 1):
            result[(i, j)] = 0
            if j + 1 <= n:
                result[(i, j)] += result[(i - 1, j + 1)]
            if j - 1 >= 0:
                result[(i, j)] += result[(i - 1, j - 1)]
    return result[(n, 0)]
```

The advantage of this implementation is that it does not use recursion and has no problems with the recursion limit. The disadvantage is that the code is slightly more complicated because the order of computing the subproblems must be designed and implemented explicitly rather than leave it for the recursion to handle.



HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI