Codility_

# Chapter 17

# Dynamic programming

Dynamic programming is a method by which a solution is determined based on solving successively similar but smaller problems. This technique is used in algorithmic tasks in which the solution of a bigger problem is relatively easy to find, if we have solutions for its sub-problems.

## 17.1. The Coin Changing problem

For a given set of denominations, you are asked to find the minimum number of coins with which a given amount of money can be paid. Assume that you can use as many coins of a particular denomination as necessary. The greedy algorithmic approach is always to select the largest denomination not exceeding the remaining amount of money to be paid. As long as the remaining amount is greater than zero, the process is repeated. However, this algorithm may return a suboptimal result. For instance, for an amount of 6 and coins of values $1, 3, 4$, we get $6 = 4 + 1 + 1$, but the optimal solution here is $6 = 3 + 3$.

A dynamic algorithm finds solutions to this problem for all amounts not exceeding the given amount, and for increasing sets of denominations. For the example data, it would consider all the amounts from 0 to 6, and the following sets of denominations: $\varnothing, \{1\}, \{1, 3\}$ and $\{1, 3, 4\}$. Let $dp[i, j]$ be the minimum number of coins needed to pay the amount $j$ if we use the set containing the $i$ smallest denominations. The number of coins needed must satisfy the following rules:

- no coins are needed to pay a zero amount: $dp[i, 0] = 0$ (for all $i$);
- if there are no denominations and the amount is positive, there is no solution, so for convenience the result can be infinite in this case: $dp[0, j] = \infty$ (for all $j > 0$);
- if the amount to be paid is smaller than the highest denomination $c_i$, this denomination can be discarded: $dp[i, j] = dp[i - 1, j]$ (for all $i > 0$ and all $j$ such that $c_i > j$);
- otherwise, we should consider two options and choose the one requiring fewer coins: either we use a coin of the highest denomination, and a smaller amount to be paid remains, or we don't use coins of the highest denomination (and the denomination can thus be discarded): $dp[i, j] = \min(dp[i, j - c_i] + 1, dp[i - 1, j])$ (for all $i > 0$ and all $j$ such that $c_i \leqslant j$).

The following table shows all the solutions to sub-problems considered for the example data.

| $dp[i,j]$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| $\varnothing$ | 0 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |
| $\{1\}$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| $\{1,3\}$ | 0 | 1 | 2 | 1 | 2 | 3 | 2 |
| $\{1,3,4\}$ | 0 | 1 | 2 | 1 | 1 | 2 | 2 |

### Implementation

Consider $n$ denominations, $0 < c_0 \leqslant c_1 \leqslant \ldots \leqslant c_{n-1}$. The algorithm processes the respective denominations and calculates the minimum number of coins needed to pay every amount from 0 to $k$. When considering each successive denomination, we use the previously calculated results for the smaller amounts.

**17.1: The dynamic algorithm for finding change.**

```
def dynamic_coin_changing(C, k):
    n = len(C)
    # create two-dimensional array with all zeros
    dp = [[0] * (k + 1) for i in xrange(n + 1)]
    dp[0] = [0] + [MAX_INT] * k
    for i in xrange(1, n + 1):
        for j in xrange(C[i - 1]):
            dp[i][j] = dp[i - 1][j]
        for j in xrange(C[i - 1], k + 1):
            dp[i][j] = min(dp[i][j - C[i - 1]] + 1, dp[i - 1][j])
    return dp[n]
```

Both the time complexity and the space complexity of the above algorithm is $O(n \cdot k)$. In the above implementation, memory usage can be optimized. Notice that, during the calculation of $dp$, we only use the previous row, so we don't need to remember all of the rows.

**17.2: The dynamic algorithm for finding change with optimized memory.**

```
def dynamic_coin_changing(C, k):
    n = len(C)
    dp = [0] + [MAX_INT] * k
    for i in xrange(1, n + 1):
        for j in xrange(C[i - 1], k + 1):
            dp[j] = min(dp[j - C[i - 1]] + 1, dp[j])
    return dp
```

The time complexity is $O(n \cdot k)$ and the space complexity is $O(k)$.

## 17.2. Exercise

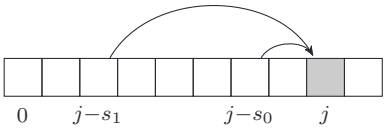
**Problem:** A small frog wants to get from position 0 to $k$ ($1 \leqslant k \leqslant 10\,000$). The frog can jump over any one of $n$ fixed distances $s_0, s_1, \ldots, s_{n-1}$ ($1 \leqslant s_i \leqslant k$). The goal is to count the number of different ways in which the frog can jump to position $k$. To avoid overflow, it is sufficient to return the result modulo $q$, where $q$ is a given number.

We assume that two patterns of jumps are different if, in one pattern, the frog visits a position which is not visited in the other pattern.

**Solution $O(n \cdot k)$:** The task can be solved by using dynamic programming. Let's create an array $dp$ consisting of $k$ elements, such that $dp[j]$ will be the number of ways in which the frog can jump to position $j$.

We update consecutive cells of array $dp$. There is exactly one way for the frog to jump to position 0, so $dp[0] = 1$. Next, consider some position $j > 0$.

The number of ways in which the frog can jump to position $j$ with a final jump of $s_i$ is $dp[j - s_i]$. Thus, the number of ways in which the frog can get to position $j$ is increased by the number of ways of getting to position $j - s_i$, for every jump $s_i$.




More precisely, $dp[j]$ is increased by the value of $dp[j - s_i]$ (for all $s_i \leqslant j$) modulo $q$.

**17.3: Solution in time complexity $O(n \cdot k)$ and space complexity $O(k)$.**

```
1  def frog(S, k, q):
2      n = len(S)
3      dp = [1] + [0] * k
4      for j in xrange(1, k + 1):
5          for i in xrange(n):
6              if S[i] <= j:
7                  dp[j] = (dp[j] + dp[j - S[i]]) % q;
8      return dp[k]
```

The time complexity is $O(n \cdot k)$ (all cells of array $dp$ are visited for every jump) and the space complexity is $O(k)$.

Every lesson will provide you with programming tasks at `http://codility.com/programmers`.