Problem 31
Investigating combinations of English currency denominations

In England the currency is made up of pound, £, and pence, p, and there are eight coins in general circulation:

1p, 2p, 5p, 10p, 20p, 50p, £1 (100p) and £2 (200p).

It is possible to make £2 in the following way:

1×£1 + 1×50p + 2×20p + 1×5p + 1×2p + 3×1p

How many different ways can £2 be made using any number of coins?

Let us start with the largest coins (£2). We can make the target amount (£2) either using just one £2 coin, or using some smaller coins instead. Therefore, the ways to make £2 with any coins are: 1 + ways to make £2 with smaller coins.

Now, let us calculate the ways to make £2 with smaller coins. We can classify the different ways in three groups:

- 0×£1 + even smaller coins (up to £2)
- 1×£1 + even smaller coins (up to £1)
- 2×£1 + nothing

The number of ways contained in the first group can be expressed like the original problem, but without £2 and £1 coins.

For the second group we have the original problem with a target amount of just £1 and without £2 and £1 coins.

The third group counts as just one way, since the target amount (£2) is a multiple of the largest coin (£1).

In general:

$$w(t,\ c) = \begin{cases} 1 & \text{if } c = 1 \text{ or } t = 0 \\ \sum_{i=0}^{\lfloor \frac{t}{c} \rfloor} w(t - ic,\ s(c)) & \text{if } c > 1 \text{ and } t > 0 \end{cases}$$

where:

- t is the target amount,
- c is the value of the largest available coin,
- $s(c)$ is the value of the next coin smaller than c, and
- $w(t,\ c)$ is the number of ways to make the target amount t with coins of value c and/or smaller coins.

The next program uses the aforementioned recursive function:

```
coins[1..8] = { 1, 2, 5, 10, 20, 50, 100, 200 }
amount = 200
function ways (target, avc):
    if avc <= 1 then return 1
    res = 0
    while target >= 0 do:
        res = res + ways (target, avc-1)
        target = target - coins[avc]
    return res
print ways (amount, 8)
```

This program calculates the correct result for the problem in the blink of an eye (4ms[1]). Nevertheless, it is far from being optimal. If we set the total amount to 1000 (£10), then the program takes more than 3s to compute the result. During this time, the function is called 325,195,472 times. The value of $w(10,\ 2)$ —which is 6, by the way— is calculated 94,300 times. For the lowest values of t and c, the function is called millions of times.

Obviously, we can use *memoization* to avoid those superfluous repeated calculations. The next program stores the results of $w(t,\ c)$ in an array, and reuses them if the function is called again with the same t and c:

```
coins[1..8] = { 1, 2, 5, 10, 20, 50, 100, 200 }
amount = 200
memo[0..amount][2..8] = { all 0 }
function ways (target, avc):
    if avc <= 1 then return 1
    t = target
    if memo[t][avc] > 0 then return memo[t][avc]
    res = 0
    while target >= 0 do:
        res = res + ways (target, avc-1)
        target = target - coins[avc]
    memo[t][avc] = res
    return res
print ways (amount, 8)
```

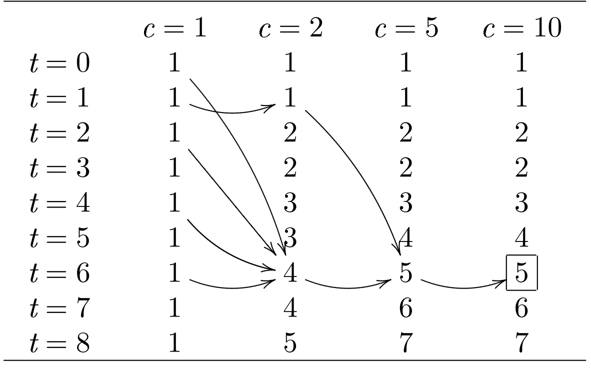
This version computes the ways to make £10 in 4ms, at the cost of using an array of 27KB approximately[2]. With this particular set of coins, the array size might be reduced, due to the fact that many elements are never used. Though, we will focus on a more drastic optimization.

[1]All times were measured with implementations in C, on an Intel® Core^{TM}2 T7200 @ 2.00GHz

[2]With 32-bit integers: $(1000 + 1) \times 7 \times 4 = 28028$B

Let us analyse the dependencies between the different values of $w(t,\ c)$.

Dependencies between values of $w(t,\ c)$



	$c = 1$	$c = 2$	$c = 5$	$c = 10$
$t = 0$	1	1	1	1
$t = 1$	1	1	1	1
$t = 2$	1	2	2	2
$t = 3$	1	2	2	2
$t = 4$	1	3	3	3
$t = 5$	1	3	4	4
$t = 6$	1	4	5	$\boxed{5}$
$t = 7$	1	4	6	6
$t = 8$	1	5	7	7

The function $w(t,\ c)$ can be reformulated as follows:

$$w'(t,\ c) = \begin{cases} 1 & \text{if } c = 1 \text{ or } t = 0 \\ w'(t,\ s(c)) & \text{if } c > 1 \text{ and } t < c \\ w'(t,\ s(c)) + w'(t - c,\ c) & \text{if } c > 1 \text{ and } t \geq c \end{cases}$$

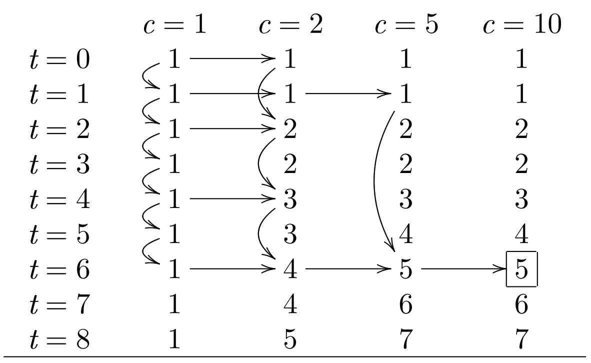
It is as straightforward as the previous version. The ways to make £2 with £1 coins and/or smaller coins, for instance, include:

- Ways to make £2 with coins smaller than £1 (this is the 0×£1 case)
- Ways to make £1 with £1 coins and/or smaller coins (cases 1×£1 and 2×£1)

If we implemented $w'(t,\ c)$ literally, we would get a very inefficient solution, since it uses recursion instead of the loop of the initial program. It would be slower and it would use too much stack space —the maximum number of nested calls would be proportional to the target amount—.

Having said this, the interesting aspect of $w'(t,\ c)$ is the graph of dependencies between different values of the function:

Dependencies[3] between values of $w'(t,\ c)$



	$c = 1$	$c = 2$	$c = 5$	$c = 10$
$t = 0$	1	1	1	1
$t = 1$	1	1	1	1
$t = 2$	1	2	2	2
$t = 3$	1	2	2	2
$t = 4$	1	3	3	3
$t = 5$	1	3	4	4
$t = 6$	1	4	5	$\boxed{5}$
$t = 7$	1	4	6	6
$t = 8$	1	5	7	7

[3]Note that some of the arrows depicted in the table —first column and first row— do not represent *recursive calls* of the function $w'(t,\ c)$, since it has been directly defined as 1 in these cases

The 1's from the first row and the first column follow longer paths of additions until they reach their final destination. In exchange, every element is calculated by adding at most two other elements. Thus, we can precalculate the whole array in less time.

Furthermore, every $w'(t,\ c)$ element depends, at most, on:

1. a previous element of its column c, and
2. the element at the same row t in the previous column $c-1$

Due to this interesting fact, we can use a simple 1D array, calculating a full column at a time, while we overwrite the previous column.

The next program is an adaptation of the solution provided by wizeman —post number 16 in the thread for problem 31 in the forum—. Using 64-bit numbers it can calculate, for instance, the ways to make £100 (1,133,873,304,647,601) in 3ms —setting `amount=10000`—.

```
coins[1..8] = { 1, 2, 5, 10, 20, 50, 100, 200 }
amount = 200
ways[0..amount] = { uninitialized array }
ways[0] = 1;
for i = 1 to 8 do:
    for j = coins[i] to amount do:
        ways[j] = ways[j] + ways[j-coins[i]]
print ways[amount]
```

The second program solves subproblems on demand, attacking the largest problem first, and moving to smaller and smaller subproblems as required. It represents the top-down approach of dynamic programming.

The third program, instead, starts solving the smallest subproblem, and then it solves larger and larger subproblems. It represents the bottom-up approach of dynamic programming, and it is significantly faster for several reasons:

- As mentioned above, every element is calculated by adding at most two other elements.
- It requires less memory and accesses it in a cache-friendly way.
- It uses loops instead of expensive function calls.

The first reason is quite important from the point of view of computational complexity. A memoized recursive implementation of $w'(t,\ c)$, though maybe stack-abusive[4], would also be faster than the memoized recursive implementation of $w(t,\ c)$.

[4]Interestingly, when memoization is used, the maximum number of nested recursive calls depends strongly on the order in which the two recursive calls of $w'(t,\ c)$ are issued.