Minimum Coins

Given a list of **N** coins, their values (**V$_1$**, **V$_2$**, …, **V$_N$**), and the total sum **S**. Find the minimum number of coins the sum of which is **S** (we can use as many coins of one type as we want), or report that it's not possible to select coins in such a way that they sum up to **S**.

```
minCoinsDP(11, new int[] {9, 6, 5, 1}) >>> 2
minCoinsDP(20, new int[] {9, 6, 5, 1}) >>> 3
```

Algorithm help:

First, recall the basic recursive algorithm you memoized previously:

> Suppose we have already found out the best way to sum up to amount `a`, then for the last step, we can choose any coin type which gives us a remainder `r` where `r = a - coins[i]` for all `i`'s. For every remainder, go through exactly the same process as before until either the remainder is 0 or less than 0 (meaning not a valid solution). With this idea, the only remaining detail is to store the minimum number of coins needed to sum up to `r` so that we don't need to recompute it over and over again.

To make change for `n` cents in a DP fashion, we are going to figure out how to make change for every value `x` < `n` first. We then build up the solution out of the solution for smaller values, given that optimal solutions of sub-problems will yield optimum results of larger problems. Having found the minimum number of coins which sum up to `x`, we can easily find the next state – the solution for `x` + 1.

For now, we will only concentrate on computing the minimum number of coins.

- Let `C[p]` be the minimum number of coins needed to make change for `p` cents.
- Let `x` be the value of the first coin used in the optimal solution.

    Then `C[p] = 1 + C[p - x]`.

Problem: We don't know `x`. Answer: We will try all possible `x` and take the minimum.

$$C[p] = \begin{cases} \min_{i:d_i \le p}\{C[p-d_i]+1\} & \textbf{if } p > 0 \\ 0 & \textbf{if } p = 0 \end{cases}$$

Longer explanation courtesy topcoder.com

For each coin **j, $V_j \leq i$**, look at the minimum number of coins found for the **$i - V_j$** sum (we have already found it previously). Let this number be **m**. If **m + 1** is less than the minimum number of coins already found for current sum **i**, then we write the new result for it.

For a better understanding let's take this example:

Given coins with values 1, 3, and 5, and the sum **S** is set to be 11.

First of all we mark that for state 0 (sum 0) we have found a solution with a minimum number of 0 coins. We then go to sum 1. First, we mark that we haven't yet found a solution for this one (a value of Infinity would be fine). Then we see that only coin 1 is less than or equal to the current sum. Analyzing it, we see that for sum 1 - **V_1** = 0 we have a solution with 0 coins. Because we add one coin to this solution, we'll have a solution with 1 coin for sum 1. It's the only solution yet found for this sum. We write (save) it.

Then we proceed to the next state – **sum 2**. We again see that the only coin which is less or equal to this sum is the first coin, having a value of 1. The optimal solution found for sum (2 - 1) = 1 is coin 1. This coin 1 plus the first coin will sum up to 2, and thus make a sum of 2 with the help of only 2 coins. This is the best and only solution for sum 2.

Now we proceed to sum 3. We now have 2 coins which are to be analyzed – first and second one, having values of 1 and 3. Let's see the first one. There exists a solution for sum 2 (3 – 1) and therefore we can construct from it a solution for sum 3 by adding the first coin to it. Because the best solution for sum 2 that we found has 2 coins, the new solution for sum 3 will have 3 coins. Now let's take the second coin with value equal to 3. The sum for which this coin needs to be added to make 3 , is 0. We know that sum 0 is made up of 0 coins. Thus we can make a sum of 3 with only one coin – 3. We see that it's better than the previous found solution for sum 3 , which was composed of 3 coins. We update it and mark it as having only 1 coin. The same we do for sum 4, and get a solution of 2 coins – 1 + 3. And so on. **Pseudocode:**

```
Set Min[i] equal to Infinity for all of i

Min[0] = 0



For i = 1 to S

       For j = 0 to N - 1

                If  (V_j <= i AND Min[i - V_j] + 1 < Min[i])
                         Then Min[i] = Min[i - V_j] + 1

Output Min[S]
```

Here are the solutions found for all sums:

Sum	Min. of coins	Coin value added to a smaller sum to obtain this sum (it is displayed in brackets)
0	0	-
1	1	1 (0)
2	2	1 (1)
3	1	3 (0)
4	2	1 (3)
5	1	5 (0)
6	2	3 (3)
7	3	1 (6)
8	2	3 (5)
9	3	1 (8)
10	2	5 (5)
11	3	1 (10)

As a result we have found a solution of 3 coins which sum up to 11.

Additionally, by tracking data about how we got to a certain sum from a previous one, we can find what coins were used in building it. For example: to sum 11 we got by adding the coin with value 1 to a sum of 10. To sum 10 we got from 5. To 5 – from 0. This way we find the coins used: 1, 5 and 5.

Having understood the basic way **DP** is used, we may now see a slightly different approach to it. It involves the change (update) of best solution yet found for a sum i, whenever a better solution for this sum was found. In this case the states aren't calculated consecutively. Let's consider the problem above. Start with having a solution of 0 coins for sum 0. Now let's try to add first coin (with value 1) to all sums already found.

If the resulting sum **t** will be composed of fewer coins than the one previously found – we'll update the solution for it. Then we do the same thing for the second coin, third coin, and so on for the rest of them. For example, we first add coin 1 to sum 0; and get sum 1. Because we haven't yet found a possible way to make a sum of 1 – this is the best solution yet found, and we mark **S[1] = 1**. By adding the same coin to sum 1, we'll get sum 2, thus making **S[2] = 2**. And so on for the first coin.

After the first coin is processed, take coin 2 (having a value of 3) and consecutively try to add it to each of the sums already found. Adding it to 0, a sum 3 made up of 1 coin will result. Till now, **S[3]** has been equal to 3, thus the new solution is better than the previously found one. We update it and mark **S[3] = 1**. After adding the same coin to sum 1, we'll get a sum 4 composed of 2 coins. Previously we found a sum of 4 composed of 4 coins; having now found a better solution we update **S[4]** to 2.

The same thing is done for next sums – each time a better solution is found, the results are updated.

The first integer in the input file, *"coins.dat"*, will be the number of cases followed by the total sum **S** and then the list of **N** coins to be used. An output file has been provided to check your results.