PROBLEM A: Dynamic Programming for Minimum Coin Change

(Targeted time to finish : not longer than 1 hour)

INPUT:

Line 1 : the list of coin denominator

Line 2 : the amount of change

OUTPUT: The minimum number of coins required for the change

EXAMPLE

INPUT	OUTPUT
1 3 4 5 7	2
1 2 5 10 13 3377	260

The following code is a memoized minimum coin change function.

```
mm = [-1]*(V+1)

def mincoin(v):
    global coin, mm

    if mm[v] == -1:
        if v == 0:
            mm[v] = 0
        else:
            minc = 10000000000
            for c in coin:
                if c <= v:
                    minc = min(minc, 1 + mincoin(v-c))
            mm[v] = minc
    return mm[v]
```

1. Given that v1 ≥ v2,
   1.1 which recursive call, to mincoin(v1) or to mincoin(v2), is made first?

   1.2 which recursive function, mincoin(v1) or mincoin(v2), returns first?

   1.3 which mm's entry, mm[v1] or mm[v2], obtains its final value first?

2. Develop a *non-recursive* minimum coin change solution i.e. does not utilize recursive function, by iterating through mm's indices with an appropriate sequence, computing value of corresponding mm's entry along the way.

PROBLEM B: 0-1 KNAPSACK (Continue)

Given a maximum weight you can carry in a knapsack and items, each with a weight and a value, find a set of items you can carry in the knapsack so as to maximize the total value.

1. If you successfully developed the memoized solution (version 2) of this problem, continue with your code. Otherwise, download the supplemented v2_mm.py to be used in this class.

The following steps will assume identifiers in v2_mm.py. You may adapt the concept to your own code as appropriate.

2. By observing how recursive calls are made,

   - it is obvious that maxVal(i,C) requires the return values of maxVal(i+1,C) and maxVal(i+1,X) where X < C.
   - According to the order of returned values, it is possible to write *a non-recursive version* of the program that enforces an order of subproblems to be solved which does not contradict that of the recursive version.
   - In agreement with recursion in KnapsackMemoi.py;
     - ☞ i may loop such that the next value of i is smaller
     - ☞ C may loop such that the next value of C is larger.

   Apply nested loops to the recursive code. This way, all the needed values for each computation will already be pre-computed and recorded in the table, according to the loops' direction.

   This pattern of looped code that building the answers, in bottom-up manner, from terminal cases up to the target case is called "**DYNAMIC PROGRAMMING**".

3. Try solving this problem by using dynamic programming technique. https://judge.u-aizu.ac.jp/onlinejudge/description.jsp?id=DPL_1_B.

EXTRA

The only way to develop skill for solving problem with dynamic programming technique through experience. So try solving this problem. Longest Common Subsequence - DMOJ: Modern Online Judge

Research online resources as necessary. A Python 3 program with a correct dynamic programming approach would pass all test cases within time limit.