Work through the steps in the paradigm one-by-one, making sure everyone is caught up before moving on to the next step. For example, it's pointless to try to define an array until you have a clear idea what sub-problems to consider and how they relate to one another -- so you really do need to complete step 0 before moving on to step 1 -- and so on.

0. Recursive structure of sub-problems.
   In every optimum solution, either
   - there is at least one coin worth d[m] (in which case the rest of the solution is optimum for amount A - d[m]), or
   - there is no coin worth d[m] (in which case the solution is optimum for denominations d[1],...,d[m-1]).

1. Definition of array that stores optimum values for sub-problems.
   Sub-problems based on restricting amount and denominations: define two-dimensional array N[a,j] = minimum number of coins required to make change for amount 'a' using denominations d[1],...,d[j], for 0 <= a <= A and 0 <= j <= m.

2. Recurrence relation for array values.

```
    - For a = 0,...,A:
          N[a,0] = oo (no change possible without coins)
    - For j = 1,2,...,m:
          N[0,j] = 0 (no coin required for amount 0)
    - For j = 1,...,m and a = 1,...,A:
          N[a,j] = N[a,j-1] if a < d[j]
               (impossible to use coins with value d[j] if d[j] > a)
          N[a,j] = min( N[a,j-1], 1 + N[a-d[j],j] ) if a >= d[j]
               (from reasoning in step 0)
```

3. Simple algorithm to compute array values bottom-up.

```
for a <- 0,...,A:
    N[a,0] <- oo
for j <- 1,...,m:
    N[0,j] <- 0
    for a <- 1,...,A:
        N[a,j] <- N[a,j-1]
        if a >= d[j] and N[a,j] > 1 + N[a-d[j],j]:
            N[a,j] <- 1 + N[a-d[j],j]
```

4. Reconstruction of optimum solution using computed array values.

   Idea: start with N[A,m] and work our way back, checking at every step whether N[a,j] == N[a,j-1] (in which case don't use any coins with value d[j]) or not (in which case use one coin with value d[j]).

```
C <- []
j <- m
a <- A
while j > 0 and a > 0:
    if N[a,j] == N[a,j-1]:
        j <- j - 1
    else:
        C <- C + [d[j]]
        a <- a - d[j]
```

```
        return C

Runtime:
    O(mA) for step 3; O(m + A) for step 4: O(mA) in total.

    This is _not_ polytime because the runtime depends on the _value_ of
    parameter A -- as a function of the size of A (the number of bits
    required to write down A's value), this is actually exponential. This
    kind of running time (technically exponential as a function of input
    size, but polynomial as a function of input values) is called
    "pseudo-polynomial time".
```