CIS 575. Introduction to Algorithm Analysis
Material for March 18, 2024

Dynamic Programming for Giving Exact Change

1 Giving Optimal Exact Change

We shall consider the problem of giving **exact change**, and assume we live in a country where coins have denominations $d_1 \dots d_n$ (each a positive integer). For a given (non-negative integer) A, our task is to find a collection of coins such that

- their total value is exactly A units
- no fewer number of coins will suffice.

We shall assume that

1. we have an unlimited supply of each denomination
2. there is a coin worth 1 unit (a "penny") since otherwise we cannot solve the problem when $A = 1$.

You may ask why this is a challenging problem, since in the US it's quite easy how to do it: to give back say 47c, we

1. first give back as many quarters as possible, that is **one quarter** with 22c left
2. next give back as many dimes as possible, that is **two dimes** with 2c left
3. next give back as many nickels as possible, that is **zero nickels** with 2c left
4. finally give back the rest in pennies: **two pennies**.

This is what we shall call a *greedy* strategy (much more about that later in the course!) and which works due to the US coin system being well designed.

But assume we didn't have nickels, and needed to give back 30c: then the above strategy would give back one quarter, and then five pennies, whereas the optimal solution is to give back three dimes. In the general case, we often cannot predict whether we should use a given coin or not, and must explore various options (eventually picking the best one).

Phrasing the problem in terms of the general recipe for dynamic programming, the *subproblems* are to compute $c[i, a]$, the minimum number of coins needed to give back a using only the denominations $d_1 \dots d_i$, when $0 \leq a \leq A$ and $0 \leq i \leq n$; then the desired answer can be retrieved as $c[n, A]$. Let us first study how to solve "small" subproblems:

- if $a = 0$ then there is nothing to give back so we don't need any coins:

$$c[i, 0] = 0 \text{ when } 0 \leq i \leq n \tag{1}$$

- if $a > 0$ but $i = 0$ then we need to give back some money but don't have any coins to do it with, so our task is impossible:

$$c[0, a] = \infty \text{ when } 0 < a \leq A \tag{2}$$

Let us next study how to solve problems whose solution depends on the solutions to smaller problems, so assume that $i > 0$ and $a > 0$.

- if $d_i > a$ then denomination i doesn't help us and we would do as well without it. Thus

$$c[i, a] = c[i - 1, a] \text{ when } 0 < i \leq n \text{ and } d_i > a \tag{3}$$

- if $d_i \leq a$ then we can use denomination i towards paying a, though that may not be the best thing to do. Thus there are two options:

  1. not use any coins of denomination i, in which case we will need $c[i - 1, a]$ coins
  2. use at least one coin of denomination i, in which case we will still need to pay $a - d_i$ units, and thus will need $1 + c[i, a - d_i]$ coins.

  Of these two options, we should pick the best:

$$c[i, a] = \mathbf{min}(c[i - 1, a],\ 1 + c[i, a - d_i]) \text{ when } 0 < i \leq n \text{ and } d_i \leq a \tag{4}$$

We have developed 4 equations: (1–4). It is straightforward to translate those into a recursive algorithm, but that algorithm will (unless memoization is used) compute many entries over and over. Instead, we shall calculate the entries bottom-up, as done by the algorithm

```
for i ← 0 to n
    c[i, 0] ← 0
    for a ← 1 to A
        if i = 0
            c[i, a] ← ∞
        else if d_i > a
            c[i, a] ← c[i − 1, a]
        else
            c[i, a] ← min(c[i − 1, a], 1 + c[i, a − d_i])
```

It is easy to check (as we must always do when implementing dynamic programming!) that each entry is defined before it is referenced (which would not be the case if we had let i run from n down to 0).

In general, the algorithm generates a table that uses space in $\mathbf{\Theta(n \cdot A)}$. Since each entry can be computed in constant time, the **running time** of the algorithm is also in $\mathbf{\Theta(n \cdot A)}$.

Let us illustrate the algorithm with $n = 3$, $d_1 = 1$, $d_2 = 4$, $d_3 = 5$, and $A = 8$: we then get the below table by first constructing the first row and next calculating the second row (from

left to right), the third row (from left to right), and so on:

$i\backslash a$	0	1	2	3	4	5	6	7	8
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1	**0**	1	2	3	4	5	6	7	8
2	**0**	1	2	3	**1**	2	3	4	**2**
3	0	1	2	3	1	1	2	3	**2**

The bottom right entry is computed as follows:

$$c[3,8] = \mathbf{min}(c[2,8], 1 + c[3,3]) = \mathbf{min}(2, 1+3) = \mathbf{2}$$

and denotes the answer we are looking for: we can do our task using **2** coins. Still, we cannot immediately see *which* two coins to use. But the entries in boldface indicate how to retrieve that information: since $c[3,8] = c[2,8]$ we see that we do not need any coins of value $d_3 = 5$; since $c[2,8] < c[1,8]$ we see that we need at least one coin of value 4; since $c[2,4] < c[1,4]$ we see that we need at least one more coin of value 4; since $c[2,0] = c[1,0]$ we see that we do not need any more coins of value 4; since $c[1,0] = c[0,0]$ we see that we do not need any coins of value 1. To conclude, the best solution is (as expected) to give back two coins of value 4.