CS577 Study Session

Dynamic Programming

April 25, 2023

§1 Count the Sequence

Given two positive integers N, develop an efficient algorithm to find how many different **strictly** increasing sequence of positive integers that the sum is N. The algorithm should have time complexity $\mathcal{O}(N^2)$ or better.

§1.1 Examples

Let's start with an example. Suppose we are given $N = 5$. There are 3 possible sequences, which are the following.

$$\begin{aligned} 5 \\ 1+4 \\ 2+3 \end{aligned}$$

Hence, the algorithm should return 3. Notice that $1+2+2=5$ but $1,2,2$ is not strictly increasing, hence do not count toward our answer. Also notice that $4+1=5$ but $4,1$ does not count toward our answer because it is not even an increasing sequence.

Let's do another example. Suppose we are given $N = 9$. There are 8 possible sequences, which are the following.

$$\begin{aligned} 9 \\ 1+8 \\ 2+7 \\ 3+6 \\ 1+2+6 \\ 4+5 \\ 1+3+5 \\ 2+3+4 \end{aligned}$$

Hence, the algorithm should return 8.

§1.2 Recursive Algorithm

Let's start by an inefficient recursive algorithm. The first question to ask is "how do we write a bruteforce algorithm for this problem?"

This might be a bit challenging if you haven't written enough bruteforce algorithms before, so let me walk you through it.

This is one of the technique that you might keep in mind: if you don't know where to start on writing bruteforce algorithm, try framing the question to look more recursive-able.

Let's frame the problem into the following question:

"What is the number of strictly increasing sequences such that the last element less than or equal to M and the sum is equal to N?"

The problem now looks very recursive-able. Can we write a recursive function out of this?

Let $OPT(N, M)$ be the number of strictly increasing sequences such that the last element less than or equal to M and the sum is equal to N.

We can divide the problem into the following (base) cases:

1. If $N < 0$ or ($M \leq 0$ and $N \neq 0$), then you could never construct such sequences.
2. If $N = 0$, then there is one possible answer, which is to not have any numbers at all in the sequence.
3. If $N > 0$, then there are 2 categories of the strictly increasing sequences
   a) If the sequence ends with M: we can take all the sequences which the last element is less than or equal to $M - 1$ with the sum being $N - M$, and then place the number M next to the last element, which result in sequences with last element being M and the sum equal to N. Therefore, the number of sequences in this case is $OPT(N - M, M - 1)$
   b) If the sequence does not end with M: then we will be consider all the sequences whose last element is $M - 1$ or less, and the sum is equal to N. Therefore, the number of sequence in this case is $OPT(N, M - 1)$.

Therefore, we can see that,

$$OPT(N, M) = \begin{cases} 0 & \text{if } N < 0 \\ 0 & \text{if } M \leq 0 \text{ and } N > 0 \\ 1 & \text{if } N = 0 \\ OPT(N - M, M - 1) + OPT(N, M - 1) & \text{Otherwise} \end{cases}$$

Thus, we could write the following algorithm 1.1

Algorithm 1.1 Finding the number of strictly increasing sequences with sum N and ends with less than or equal to M

```
1: procedure OPT(N,M)
2:     if N < 0 or (M ≤ 0 and N > 0) then
3:         return 0
4:     else if N = 0 then
5:         return 1
6:     else
7:         return OPT(N − M, M − 1) + OPT(N, M − 1)
8:     end if
9: end procedure
```

§1.3 Dynamic Programming

Now that we have the recursive algorithm for the problem. But as we know, the recursive algorithm runs in exponential time (why?), which would take too long once N becomes bigger than 1000. However, we already know that the recursive function runs in exponential time complexity because they keep recomputing the same problem instances (a.k.a. "subproblems") over and over.

We can simply just speed it up by memorize that we had already calculated, and thus achieving $\mathcal{O}(N^2)$ time complexity as the following algorithm 1.2.

Algorithm 1.2 Finding the number of strictly increasing sequences with sum N and ends with less than or equal to M

```
1: procedure Solver(N)
2:     memo ← {}                          ▷ Assign an empty dictionary
3:     procedure OPT(N,M)
4:         if N < 0 or (M ≤ 0 and N > 0) then
5:             return 0
6:         else if N = 0 then
7:             return 1
8:         else if (N, M) is in memo then
9:             return memo[(N, M)]
10:        else
11:            memo[(N, M)] ← OPT(N − M, M − 1) + OPT(N, M − 1)
12:            return memo[(N, M)]
13:        end if
14:    end procedure
15:    return OPT(N, N)
16: end procedure
```

However, let's also explore iterative version of this dynamic programming algorithm.

§1.3.1 Definitions required for the algorithm to work

None.

§1.3.2 Description of Matrix

A 2-dimensional array dp, with dimensions $(N+1) \times (N+1)$. Assume that this array is 0-index based. We will define that $dp[N][M]$ is the number of strictly increasing sequences whose last element is at most M and the sum is equal to N.

Base Cases: When the sum is $N = 0$, there is always 1 possible sequence. When $M = 0$ and $N > 0$, it is impossible to construct any such sequences. In other words, for all $1 \leq i \leq N$, $0 \leq j \leq N$,

$$dp[i][0] = 0$$
$$dp[0][j] = 1$$

§1.3.3 Bellman Equation

Normally, once you come up with the recursive algorithm, you will naturally find the Bellman equation. Why? Because that recursive definition itself is the recurrence relation (a.k.a. Bellman Equation) that we look for!

For all any (i, j) such that $i \geq 1$ and $j \geq 1$,

$$dp[i][j] = dp[i-1][j] + \begin{cases} 0 & \text{if } j > i \\ dp[i-1][j-i] & \text{Otherwise} \end{cases}$$

§1.3.4 Order to populate

We can set up all the base cases in any order. Once we set up all the base cases, we can populate from row 1 to row N, and for each row, we populate from column 1 to column N.

§1.3.5 The location of the solution

After populating all entries in dp, the answer will be on $dp[N][N]$.

§1.4 Implementation

The following is the python code for this algorithm. (Note: the dimensions i, j is swapped in this implementation.)

```
def play(n):
    memo = [[0 for j in range(n+1)] for i in range(n+1)]
    memo[0][0] = 1
    for i in range(1, n+1):
        memo[i][0] = 1
        for j in range(1, n+1):
            memo[i][j] = memo[i-1][j]
            if j >= i:
                memo[i][j] += memo[i-1][j-i]
            # print(memo[i][j],end=' ')
        # print()
    return memo[n][n]
```

```
13
14 def solver(n):
15     return play(n)
16
17 n = 9 # for example
18 print(solver(n)) # 8
```

§2 Count the Sequence 2

Given two positive integers N and M, develop an efficient algorithm to find how many different strictly increasing sequence of positive integers of length M that the sum is N. The algorithm should have time complexity $\mathcal{O}(N^2M)$ or better.