Dynamic Programming

Alex Li

February 25, 2019

1 Introduction

Today we will be talking about dynamic programming. Again. The reason is because dynamic programming is just that useful. Let's start with a quick review. DP involves computing lots of intermediate results and storing them so that we can obtain the result that we finally want by using a recurrence relation. In this way, it is a lot like recursion. Once you have made the recurrence, the DP solution is natural. We will explore this route today, and show how we can use 2D DPs to satisfy recurrences that we could not have otherwise done by using 1D DPs.

2 Problem 1: Concerts

https://codeforces.com/gym/101669 John would like to listen to certain bands in some order. 1 band will play each day over the next N days. After listening to a certain band, he can't listen to any band for a certain number of days. How many ways can do this (modulo a prime)?
01
ABA
$AABAA$
Letters represent distinct bands. First row: waiting time for bands A, B. Second row: John's list. Last row: days each concert plays. Ways = 2. How can we tell? With normal dynamic programming, we would just find the number of ways for each prior day and use it to compute the current day. This doesn't work very well, though, since the number of ways is 0 while we fill in the sequence. Well, it works if the length of the sequence is 1. Thinking more, this is because we know if answer if the length of the sequence is 0 - hence, we can just do a multidimensional DP on both!

	X	A	A	B	A	A
X	1	1	1	1	1	1
A	0	1	2	2	3	4
B	0	0	0	2	2	2
A	0	0	0	0	0	2

: Another example
$1, 0$
AB
$ABBBBABBBB$

	X	A	B	B	B	B	A	B	B	B	B
X	1	1	1	1	1	1	1	1	1	1	1
A	0	1	1	1	1	1	2	2	2	2	2
B	0	0	0	1	2	3	3	4	6	8	10

3 Problem 2: Generalized Roman Numerals

https://codeforces.com/gym/100641 Ancient Rome has a numbering system based on adding and subtracting. The rule to evaluating is to evaluate the first letter and the rest. If the first letter is less than the rest, negate it, then add. $SoCXIV = C(X(I(V))) = 100 + (10 + (-1 + 5))$. If we change the order of parenthesis, what different sums can we get? For CXIV, we can also get ((CX)I)V = ((100 + 10) + 1) + 5.
What can XIXIX get? (8,10,28,30,32)
How can we generate a recurrence relation for the set of sums we can get? (not dp solution yet)
Hint 0: the answer is a bit complex Hint 1: In normal roman numerals, we split a string into a start and end part, where the start part has length 1. So the recurrence is $strVal = \pm firstletter + restoftheletters$
Hint 2: Instead of evaluating based on the first letter like normal roman numerals, we are evaluating based on the first part and second part. So the recurrence will have several parts to it
Answer: ok it is the union of every split, so something like

$$ans(1, n) = \bigcup_{i=1}^{n} \{\pm x + y \mid x \in ans(1, i),\ y \in ans(i, n)\}$$

How can we turn this into a DP? We need to know answers for any substring of a given string (provided that the substring starts at the beginning xor ends at the end.) So far we have just been going from left to right, but that won't work here. Ideas? (Give hints until they find the answer) ans: DP on length first, then go left to right or something

```
\\ note that input is processed already, we are getting an int[] representing letters (I
Set solve(int[] expression){
    int n = expression.length;
    \\ validValues[i][j] is the set of all values that can
    \\ be made by expression.substring(i,j)
    validValues = new Set[n + 1][n + 1];
```

```
    for (int i = 0; i < n; i++) {
        validValues[i][i + 1].add(expression[i]);
    }
    for (int len = 2; len <= n; len++) {
        for (int st = 0; st <= n - len; st++) {
            int end = st + len;
            for (int mid = st + 1; mid < end; mid++) {
                for (int l : validValues[st][mid]) {
                    for (int r : validValues[mid][end]) {
                        if (l >= r) {
                            validValues[st][end].add(l + r);
                        } else {
                            validValues[st][end].add(r - l);
                            }
                        }
                    }
                }
            }
        }
    }
    return validValues[0][n];
}
```