Competitive Programming 2

Week 4: Dynamic Programming 2

Last week

Basic DP versions

1. **Parameter: Index i in an array - ex: LIS**
2. **Parameter: Subarray (i, j) of an array - ex: Matrix Chain multiplication**
3. Parameter: Indices (i, j) in two arrays - ex: Edit Distance, LCS
4. Parameter: Knapsack-Style Parameter - ex: Knapsack, Coin change etc…

Today

Basic DP versions

1. Parameter: Index i in an array - ex: LIS
2. Parameter: Subarray (i, j) of an array - ex: Matrix Chain multiplication
3. **Parameter: Indices (i, j) in two arrays - ex: Edit Distance, LCS**
4. **Parameter: Knapsack-Style Parameter - ex: Knapsack, Coin change etc…**

What is Dynamic Programming?

- Exploit a recursion to solve a problem more efficiently than bruteforce.
- Two skills
    - Master the mechanics of implementing DP
        - Top-down (recursion with memoing) and Bottom-up
        - Reusing memory in bottom-up (less critical)
        - Recover additional information regarding the solution
    - Identify the recursive relation
        - The DP structure (defining states and identifying dependency)
            - This determines the number of states
        - The exact recursive relationship
            - This determines the complexity of computing each state

Basic DP Problems on LeetCode

- 70. Climbing Stairs (Easy)
- 746. Min Cost Climbing Stairs (Easy)
- 198. House Robber (Easy)   Rob it max in ICPC qual
- 213. House Robber II (Medium)
- 337. House Robber III (Medium)   Breakfast club in ICPC qual
  - A bit harder than basic; DP over binary tree
- 62. Unique Paths (Medium)
- 63. Unique Paths II (Medium)
- 64. Minimum Path Sum (Medium)
- 120. Triangle (Medium)   Understand triangular 2D array
- 174. Dungeon Game (Hard)
  - Easier to go backwards, think about what to compute recursively

LeetCode: 91 Decode Ways (1616 Thumbs up, 1866 down)

A message containing letters from A-Z is being encoded to numbers using the following mapping:

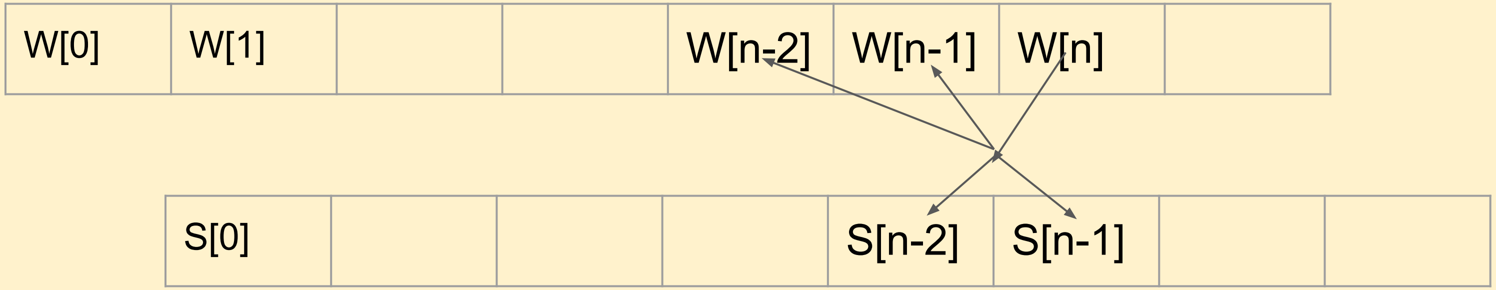
```
'A' -> 1  'B' -> 2      ...      'Z' -> 26
```

Given a non-empty string containing only digits, determine the total number of ways to decode it.

```
Input: "226"       Output: 3
Explanation: It could be decoded as "BZ" (2 26), "VF" (22 6), or
"BBF" (2 2 6).
```

Decode Ways Analysis

- Given a string S of length n, suppose we know W[0], W[1], …, W[n-1]: the number of ways to decode all prefixes up to length n-1. How to compute W[n]?
- Case analysis on the n'th letter S[n-1] and the (n-1)-th letter S[n-2]:
    - If S[n-1] can encode a single letter: then need to add W[n-1].
    - If the two digits S[n-2]S[n-1] can encode a letter, then add W[n-2].



Decode Ways: Code (Using Rolling Variables)

```
public int numDecodings(String s) {
  int a=1, b=0;             // a for W[n-2], b for W[n-1]
  if (canSingle(s.charAt(0)))  b = 1;
  for (int i=1; i<s.length(); i++) {
    int  c = 0;
    if (canSingle(s.charAt(i)))               c += b;
    if (canDouble(s.charAt(i-1),s.charAt(i)))  c += a;
    a = b;  b = c;
  }
  return b;
}
```

O(N) time complexity
O(1) extra space complexity

LeetCode 639. Decode Ways II (291 up; 392 down)

A message containing letters from A-Z is being encoded to numbers using the following mapping:

```
'A' -> 1  'B' -> 2        ...        'Z' -> 26
```

The encoded string can also contain the character '*', which can be treated as one of the numbers from 1 to 9. Given the encoded message containing digits and the character '*', return the total number of ways to decode it mod 1e9 + 7.

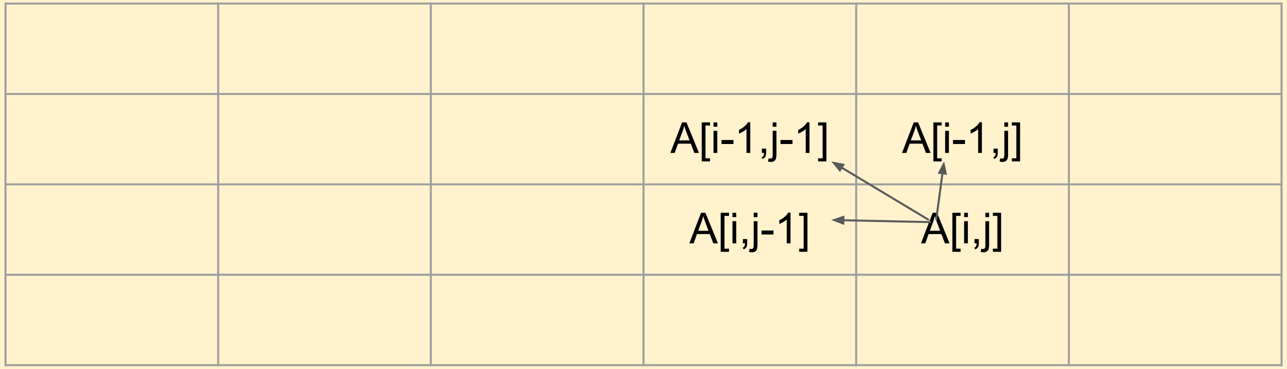
Same DP structure, but (somewhat) more complicated recursive function.

DP Problems involving Two Strings

- 1143. Longest Common Subsequence (Medium)
- 712. Minimum ASCII Delete Sum for Two Strings (Medium)
- 583. Delete Operation for Two Strings (Medium)
- 072. Edit Distance (Hard)
- 115. Distinct Subsequences (Hard)
- 097. Interleaving Strings (Hard)
- 044. Wildcard Matching (Hard)
- 10. Regular Expression Matching (Hard)
  - Different dependency from others
- 1092. Shortest Common Supersequence (Hard)

DP Problems involving Two Strings

- For many (but not all) such problems, DP structure is a 2-D table, where each cell depends on the three cells (the top, the left, and the top-left).
- Given two strings of lengths m and n, it often helps to consider an (m+1) by (n+1) DP table, where first row/column for empty string



1143. Longest Common Subsequence

Given two strings S and T, return the length of their longest common subsequence. A *subsequence* of a string is a new string generated from the original string with some characters(can be none) deleted without changing the relative order of the remaining characters.

```
Input: S = "badge", T = "abcdef"
Output: 3
Explanation:  LCS are "bde" and "ade"
```

LCS: Recursive Relation

How to compute the length of LCS of S[1..i] and T[1..j] recursively?

Case analysis on S[i] and T[j]

If (S[i] == T[j]) then LCS(i,j) = 1 + LCS(i-1,j-1)

- The LCS is the LCS of S[1..i-1] and T[1..j-1] plus the character S[i]

If (S[i] != T[j])   then LCS(i,j) = max(LCS(i,j-1), LCS(i-1,j);

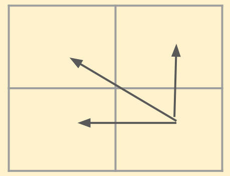
- The LCS of S[1..i] and T[1..j] is either the LCS of S[1..i-1] and T[1..j], or the LCS of S[1..i] and T[1..j-1]

LCS: the DP Table: Length of LCS

S

		a	b	c	d	e	f
	0	0	0	0	0	0	0
b	0	0	1	1	1	1	1
a	0	1	1	1	1	1	1
d	0	1	1	1	2	2	2
g	0	0	1	1	2	2	2
e	0	0	1	1	2	3	3

Recursion dependency:



Starting at the bottom-right cell, and going backwards, we can extract all LCS.

Template Code for Two-String DP with O(mn) Memory

```
int twoStringDP(string s, string t) {
  int m = s.length(), n = t.length();
  vector<int> dp(m+1,vector(n+1,0));   // (m+1)(n+1) array with all 0’s
  dp[0][0] = ??            // initialize base case if not 0
  for (int j=1; j<=n; j++)  dp[0][j] = ??;  // initialize row 0 if needed
  for (int i=1; i<=m; i++) {
    char& ch = s[i-1];
    dp[i][0] = ??          // initialize column 0 if needed
    for (int j=1; j<=n; j++) {
      // update dp[i][j] based on ch, t[j-1], dp[i-1][j-1], dp[i][j-1], …
    }
  }
  return dp[m][n];
}
```

44 Wildcard Matching

Given an input string (s) and a pattern (p), implement wildcard pattern matching with support for '?' and '*'. Where '?' Matches any single character, and '*' Matches any sequence of characters (including the empty sequence). The matching should cover the entire input string (not partial).

```
Input: s = "adceb"      p = "*a*?b"            Output: true

Input:  s = "acdcb"     p = "a*c?b"        Output: false
```

Wildcard Matching: the DP Table: Matches or not

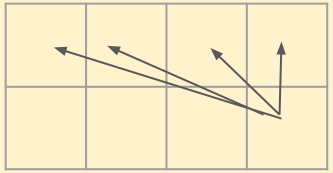
Empty pattern matches empty string

Empty pattern does not match non-empty string

P

		a	d	a	e	b
	T	F	F	F	F	F
*	T	T	T	T	T	T
a	F	T	F	T	F	F
*	F	T	T	T	T	T
?	F	F	T	T	T	T
b	F	F	F	F	F	T

Recursion dependency:



How to fill a cell when we see a letter in a pattern?
How to fill a cell when we see a ? in a pattern?
How to fill a cell when we see a * in a pattern?

Wildcard Matching

Given a pattern P and string S. Let M[i][j] denote whether the prefix of P with i symbols matches the prefix of S with j letters. Then M[i][j] can be computed as follows:

When P[i-1] is a letter	M[i][j] = (P[i-1] == S[j-1]) && M[i-1][j-1]
When P[i-1] is ‘?’	M[i][j] = M[i-1][j-1]
When P[i-1] is ‘*’	M[i][j] = M[i-1][0] || M[i-1][1] || …. || M[i-1][j]

Naive implementation has worst-case time complexity O(M^2N). Why? Keeping track of M[i-1][0] || M[i-1][1] || …. || M[i-1][j] results in time complexity O(MN)

LeetCode: Coin Change

You are given coins of different denominations and a total amount of money *amount*. Write a function to compute the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1. You may assume that you have an infinite number of each kind of coin.

```
Input: coins = [2, 3, 8, 11], amount = 18        Output: 3
Explanation: 18 = 8 + 8 + 2
```

Note that greedy approach (always choosing a coin with largest value) does not give correct answer here: 18 = 11 + 3 + 3 + 2

(Greedy would work on some denominations, such as US coins.)

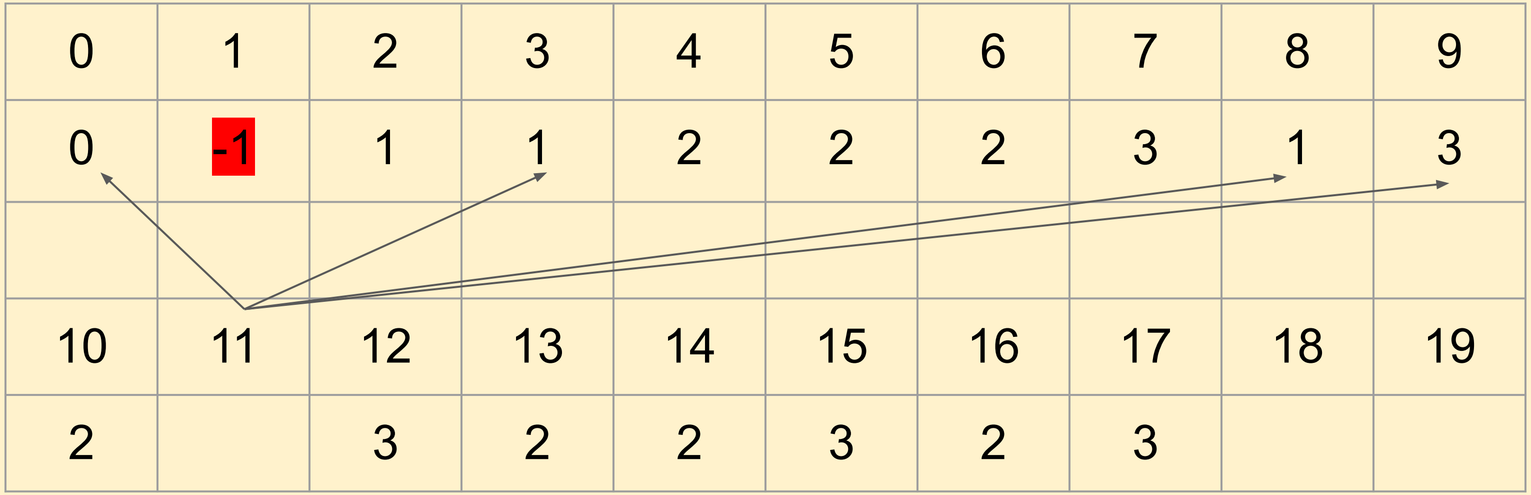
Coin Change Analysis

- In general, the problem is NP-Complete, so intractable (requires exponential time) in the worst case.
- However, it is solvable if the number of desired amount is not too large.
- Given m coins with values c_1, c_2, …, c_m, the way to make N with the least number of coins is 1 plus the least number of coins needed to make any of N-c_1, N-c_2, …, N-c_m
- The number of states equals N.
- If we have figured out the optimal way to reach any amount <N, it takes O(C) time to find the optimal way to reach N.
- Time complexity is O(N C), where C is number of coins.

Coin Change DP Table

```
Input: coins = [2, 3, 8, 11], amount = 18          Output: 3
Explanation: 18 = 8 + 8 + 2
```

What value should we put for 11?
Which cell do we look at for 18?



0	1	2	3	4	5	6	7	8	9
0	-1	1	1	2	2	2	3	1	3
10	11	12	13	14	15	16	17	18	19
2		3	2	2	3	2	3		

LeetCode: Partition Equal Subset Sum

Given a non-empty array containing only positive integers, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal. Note: 1. Each of the array element will not exceed 100. 2. The array size will not exceed 200.

```
Input: [1, 5, 3, 5, 8]        Output: true
```

```
Explanation: 1 + 5 + 5 = 3 + 8
```

Partition Equal Subset Sum: Analysis

- Let S be the sum. The question asks whether we can pick a subset of integers that sum to T=S/2, which is at most 10,000.
- Can we use recursion structure similar to Coins Change, e.g., determining whether T is reachable using whether 1,2,...,T-1 is reachable?
- No, we did not keep track of all state information. What is the key difference?
- In Coins Change, we have unlimited number of coins of each type, which coins we have used does not limit our choices going further. Here once we have used a number, it is no longer available.

Partition Equal Subset Sum: Analysis

- We need to expand the state information to include which numbers have been considered.
- Consider whether we can use the first i numbers in the array to make a number j. Call this dp[i][j]
- Then 0<= i <=200, 0<= j <= 10,000.
- And dp[i][j] is true if either dp[i-1][j] is true or dp[i-1][j-num_i]
    - Either j equals sum of some subset of the first i-1 numbers
    - Or j minus the i-th number equals sum of some subset of the first i-1 number
- With N numbers and target T, there are O(N T) states, and to compute each new state takes constant time

0/1 Knapsack Problem (Version 1)

Given n items, each of weight w_i and value v_i. Select some items to put in a knapsack of capacity W to get the maximum total value.

$1 \le n \le 100$ $1 \le w\_i \le 100$ $1 \le v\_i \le 100$ $1 \le W \le 10000$

How to define the DP state? What is the computational complexity?

Dp[i][j] is the maximum value that can be obtained under weight j; using the first j items. O(n W)

0/1 Knapsack Problem (Version 2)

Given n items, each of weight w_i and value v_i. Select some items to put in a knapsack of capacity W to get the maximum total value.

$1 \leq n \leq 100$ $1 \leq w\_i \leq 1e7$ $1 \leq v\_i \leq 100$ $1 \leq W \leq 1e9$

How to solve this? Can we use the same approach as the previous one?

Define DP over all possible value. Compute minimal weight to reach a given value.

0/1 Knapsack Problem (Version 3)

Given n items, each of weight w_i and value v_i. Select some items to put in a knapsack of capacity W to get the maximum total value.

$1 \le n \le 20$ $1 \le w\_i \le 1e8$ $1 \le v\_i \le 1e8$ $1 \le W \le 1e9$

How to solve this?

Searching over all 2^n subsets.

A Harder Variant of Knapsack

Given a multi-set of numbers such that the number a_i appears m_i times, where 1≤i ≤n. Determine whether one can choose some numbers among them such that they sum to K.

1≤ n ≤100   1≤ a_i,m_i ≤1e5   1≤K≤1e5

Which of the problem we have discussed is this most similar to?

(coins change, equal subset sum, 0/1 knapsack)

If we use the same approach, what is the time complexity?

A Harder Variant of Knapsack: Analysis

- Define dp[i][j] to be the number of a_i left when using only a_1, … a_i to make j, and -1 if making j using only a_1, … a_i is impossible
- To compute dp[i][j], there are three cases
  - dp[i-1][j] >= 0,   i.e., we can make j using only a_1, …, a_{i-1}, then all m_i of a_i are left   so dp[i][j] = m_i
  - dp[i-1][j] <0, and j<a_i or dp[i][j-a_i] <= 0,   then dp[i][j] = -1
  - Otherwise, dp[i][j] = dp[i][j-a_i] - 1

Interview Questions (Best Time to Buy and Sell Stock)

Given an array of the prices of a given stock on each day. How to gain maximum profit if

- You can buy one and then sell one share of the stock (LeetCode 121)
- Can buy/sell multiple times, can hold at most one share. (LeetCode 122)
- Can buy/sell multiple times, can hold at most one share, and has to wait for one day before buying again. (LeetCode 309)
- Can buy/sell multiple times, can hold at most one share, and has to pay a fee for each transaction. (LeetCode 714)
- Can buy/sell at most twice, and can hold at most one share. (LeetCode 123)
- Can buy/sell at most K times, and can hold at most once share. (188)

Announcements

- Next week: Dynamic Programming 3
    - Optimizations
    - ICPC questions
- Sign the attendance sheet before you leave!