dp

2021 年 11 月 12 日

目录

Chapter 1

Dynamic Programming, 动态规划

1.1 总结

1.2 第二次仍不会的题号记这里

- 1872 (stone game 8), 647 329 494 131 518 1723(need summary)
- 根据 CLRS，动态规划分为两种：
- top-down with memoization (递归记忆化搜索)

等价于带缓存的，搜索树上的 DFS 比较贴近于新问题正常的思考习惯

- bottom-up (自底向上循环迭代)

以"reverse topological order" 处理每个子问题下面依赖的所有子问题都算完了才开始计算当前一般依赖于子问题之间天然的"size

1.2.1 647. Palindromic Substrings - Medium

Given a string s, return the number of palindromic substrings in it.
A string is a palindrome when it reads the same backward as forward.
A substring is a contiguous sequence of characters within the string.

```java
public int countSubstrings(String t) {
    int n = t.length(), ans = 0;
    char [] s = t.toCharArray();
    boolean [][] dp = new boolean [n][n];
    for (int i = n-1; i >= 0; i--)
        for (int j = i; j < n; j++) {
            dp[i][j] = s[i] == s[j] && (j-i <= 2 || dp[i+1][j-1]);
            if (dp[i][j]) ans++;
        }
    return ans;
}
```

1.2.2 516. Longest Palindromic Subsequence - Medium

Given a string s, find the longest palindromic subsequence's length in s.
A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.
只要把原字符串反过来，两个字符串找最长公共子序列，就是最长回文了

```java
public int longestPalindromeSubseq(String tt) {
    int n = tt.length();
    char [] s = tt.toCharArray(); // ori
    char [] t = (new StringBuilder(tt).reverse().toString()).toCharArray(); // reverse
    int [][] dp = new int [n+1][n+1];
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++) {
            if (s[i-1] == t[j-1]) dp[i][j] = dp[i-1][j-1] + 1;
            else dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
        }
    return dp[n][n];
}
```

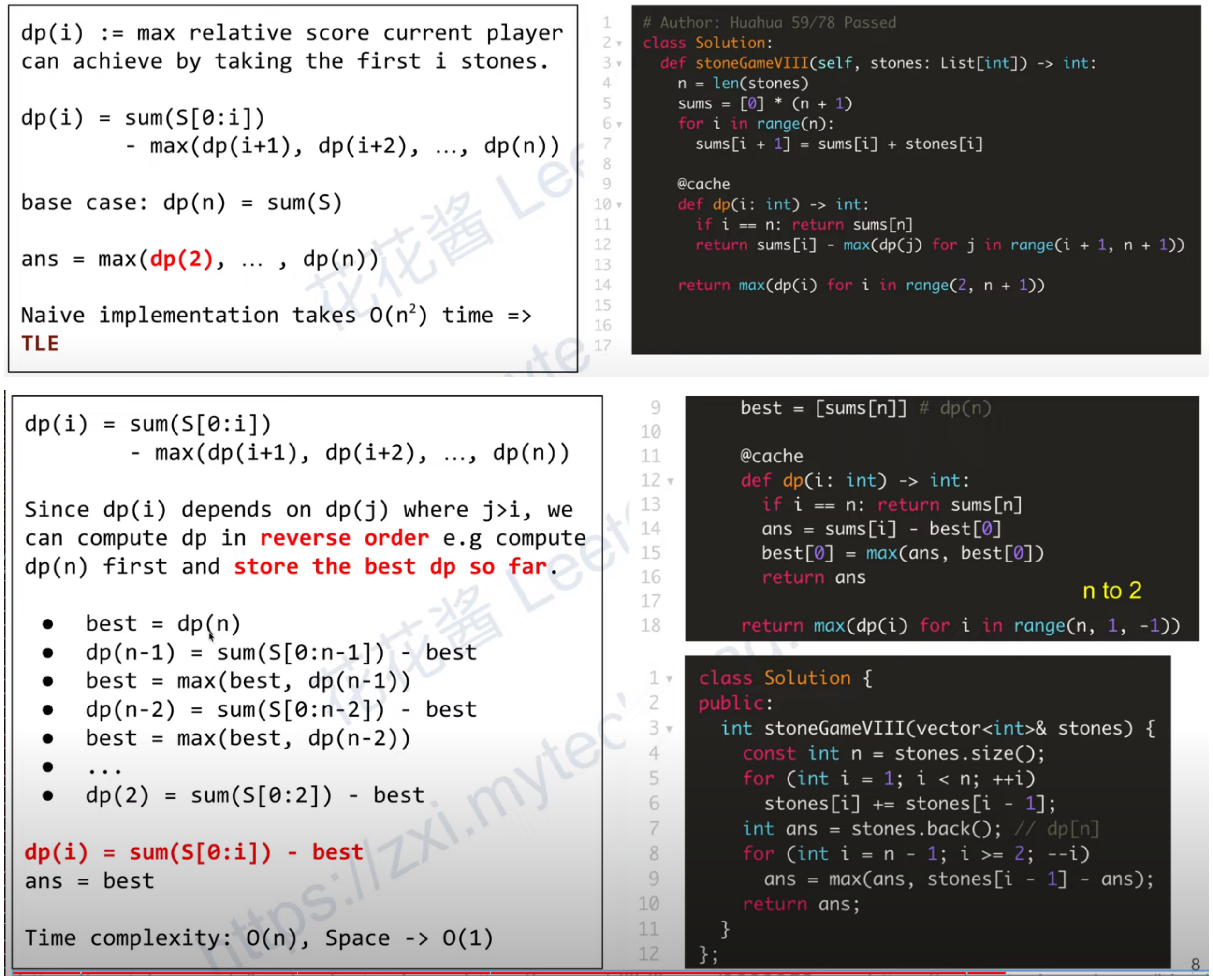
1.2.3 1872. Stone Game VIII - hard 需要好好理解消化

Alice and Bob take turns playing a game, with Alice starting first. There are n stones arranged in a row. On each player's turn, while the number of stones is more than one, they will do the following: Choose an integer x > 1, and remove the leftmost x stones from the row. Add the sum of the removed stones' values to the player's score. Place a new stone, whose value is equal to that sum, on the left side of the row. The game stops when only one stone is left in the row. The score difference between Alice and Bob is (Alice's score - Bob's score). Alice's goal is to maximize the score difference, and Bob's goal is the minimize the score difference. Given an integer array stones of length n where stones[i] represents the value of the ith stone from the left, return the score difference between Alice and Bob if they both play optimally.

1. 解题思路与分析

   这里我原始的做法 dfs+ 记忆数组会超时，是因为数组发生了改变，其盘状态发生了改变，所以记忆无效?!!! 才会超时（感觉还理解得不透，这里）

   所以采用反向遍历的方法，将 O(N^2) 变为 O(N)



```
dp(i) := max relative score current player
can achieve by taking the first i stones.

dp(i) = sum(S[0:i])
        - max(dp(i+1), dp(i+2), ..., dp(n))

base case: dp(n) = sum(S)

ans = max(dp(2), ... , dp(n))

Naive implementation takes O(n²) time =>
TLE
```

```python
# Author: Huahua 59/78 Passed
class Solution:
  def stoneGameVIII(self, stones: List[int]) -> int:
    n = len(stones)
    sums = [0] * (n + 1)
    for i in range(n):
      sums[i + 1] = sums[i] + stones[i]

    @cache
    def dp(i: int) -> int:
      if i == n: return sums[n]
      return sums[i] - max(dp(j) for j in range(i + 1, n + 1))

    return max(dp(i) for i in range(2, n + 1))
```

```
dp(i) = sum(S[0:i])
        - max(dp(i+1), dp(i+2), ..., dp(n))

Since dp(i) depends on dp(j) where j>i, we
can compute dp in reverse order e.g compute
dp(n) first and store the best dp so far.

  •  best = dp(n)
  •  dp(n-1) = sum(S[0:n-1]) - best
  •  best = max(best, dp(n-1))
  •  dp(n-2) = sum(S[0:n-2]) - best
  •  best = max(best, dp(n-2))
  •  ...
  •  dp(2) = sum(S[0:2]) - best

dp(i) = sum(S[0:i]) - best
ans = best

Time complexity: O(n), Space -> O(1)
```

```python
    best = [sums[n]] # dp(n)

    @cache
    def dp(i: int) -> int:
      if i == n: return sums[n]
      ans = sums[i] - best[0]
      best[0] = max(ans, best[0])
      return ans
                                      n to 2
    return max(dp(i) for i in range(n, 1, -1))
```

```cpp
class Solution {
public:
  int stoneGameVIII(vector<int>& stones) {
    const int n = stones.size();
    for (int i = 1; i < n; ++i)
      stones[i] += stones[i - 1];
    int ans = stones.back(); // dp[n]
    for (int i = n - 1; i >= 2; --i)
      ans = max(ans, stones[i - 1] - ans);
    return ans;
  }
};
```

```java
// 使用 dp(i) 表示还剩下 [i, n) 要选择的情况下，Alice 所能得到的最大分数差。
//     对于某个玩家来说，其对应决策可以分为两种：
//     选取当前数及之前的所有数（等价于 pres[pos]，其中 pos 为上个玩家选完后的下个位置），那么 dp[i] = pres[i] - dp[i+1]。
//     这是因为 bob 也会最大化发挥。
//     不选择当前数（可能选下一个，下下一个。。。 etc），那么 dp[i] = dp[i + 1]
public int stoneGameVIII(int[] stones) {
    int n = stones.length;
    int [] dp = new int [n];
    Arrays.fill(dp, Integer.MIN_VALUE);
    int [] pre = new int [n+1];
    for (int i = 1; i <= n; i++)
        pre[i] = pre[i-1] + stones[i-1];
    dp[n-1] = pre[n];
    for (int i = n-2; i >= 0; i--)
        dp[i] = Math.max(dp[i+1], pre[i+1]-dp[i+1]);
    return dp[1];
}
```

- 更精简的代码如下

```java
public int stoneGameVIII(int[] stones) {
    int n = stones.length;
    for (int i = 1; i < n; i++)
        stones[i] += stones[i-1]; // 原位求前缀和
    int ans = stones[n-1];
    for (int i = n-1; i >= 2; i--)
        ans = Math.max(ans, stones[i-1] - ans); // 一遍反向遍历求最优解
    return ans;
}
```

1.2.4 464. Can I Win 这个题：为什么顺序无关了？

In the "100 game" two players take turns adding, to a running total, any integer from 1 to 10. The player who first causes the running total to reach or exceed 100 wins. What if we change the game so that players cannot re-use integers? For example, two players might take turns drawing from a common pool of numbers from 1 to 15 without replacement until they reach a total >= 100. Given two integers maxChoosableInteger and desiredTotal, return true if the first player to move can force a win, otherwise, return false. Assume both players play optimally.

```java
// state 是前走的人走完之后的局面，sum 是当前数字总和，返回的是当前走的人是否能赢
private boolean dfs(int max, int target, int state, int val) {
    if (dp[state] != -1) return dp[state] > 0;
    if (val >= target) { // 如果对方取数的时候总和达到 target 了，则当前走的人输了，做记忆并返回 false
        dp[state] = 0;
        return false;
    }
    for (int i = 1; i <= max; i++) {  // 枚举当前人取哪个数
        if ((state >> i-1 & 1) == 0 && !dfs(max, target, state | (1 << i-1), val + i)) {
            dp[state] = 1;
            return true;
        }
    }
    dp[state] = 0;
    return false;
}
int [] dp;
public boolean canIWin(int maxChoosableInteger, int desiredTotal) {
    if (desiredTotal <= maxChoosableInteger) return true;
    if (desiredTotal > (maxChoosableInteger + 1)*maxChoosableInteger / 2) return false;
    dp = new int[1 << maxChoosableInteger]; // 时空复杂度 O ( 2 m ) O(2^m)O(2
    Arrays.fill(dp, -1);
    return dfs(maxChoosableInteger, desiredTotal, 0, 0);
}
```

- 另外这第二次又看见的解法

```java
public boolean canIWin(int maxChoosableInteger, int desiredTotal) { // 这个师与其它类假题相比，为什么顺序无关？
    if (desiredTotal == 0) return true; // 如果 1 到最大能选的值所有和都不能满足目标值，那么肯定失败
    if ((maxChoosableInteger+1) * maxChoosableInteger / 2 < desiredTotal) return false;
    char [] state = new char [maxChoosableInteger];
    for (int i = 0; i < maxChoosableInteger; i++) state[i] = '0';
    return dfs(desiredTotal, state, new HashMap<>());
}
private boolean dfs(int sum, char [] st, Map<String, Boolean> map) {
    String key = new String(st);
    if (map.containsKey(key)) return map.get(key);
    for (int i = 0; i < st.length; i++) {
        if (st[i] != '0') continue;
        st[i] = '1';
        if (sum <= i+1 || !dfs(sum - (i+1), st, map)) {
            map.put(key, true);
            st[i] = '0';
            return true;
        }
        st[i] = '0';
    }
    map.put(key, false);
    return false;
}
```

- // 下面这个效率更高

```
public boolean canIWin(int maxChoosableInteger, int desiredTotal) {
    if (desiredTotal <= 0) return true;
    int sum = (maxChoosableInteger + 1) * maxChoosableInteger / 2;
    if (sum < desiredTotal) return false;
    boolean[] vis = new boolean[maxChoosableInteger+1];
    return helper(desiredTotal, vis);
}
Map<Integer, Boolean> map = new HashMap<>();
public boolean helper(int desiredTotal, boolean[] vis) {
    if (desiredTotal <= 0) return false;
    int symbol = format(vis);
    if (map.containsKey(symbol)) return map.get(symbol);
    for (int i = 1 ; i < vis.length ; i++) {
        if (!vis[i]) {
            vis[i] = true;
            if (!helper(desiredTotal-i, vis)) {
                vis[i] = false; // 这里不回复状态会影响其它结果
                map.put(symbol, true);
                return true;
            }
            vis[i] = false;
        }
    }
    map.put(symbol, false);
    return false;
}
public int format(boolean[] vis) {
    int symbol = 0;
    for (boolean select : vis) {
        symbol <<= 1;
        if (select) symbol |= 1;
    }
    return symbol;
}
```

1.2.5 494. Target Sum - Medium

You are given an integer array nums and an integer target.

You want to build an expression out of nums by adding one of the symbols '+' and '-' before each integer in nums and then concatenate all the integers.

For example, if nums = [2, 1], you can add a '+' before 2 and a '-' before 1 and concatenate them to build the expression "+2-1". Return the number of different expressions that you can build, which evaluates to target.

- 该题是一道非常经典的题目，在面试中很可能会考到。该题有多种解法。
- 第一种解法：DFS，brute force。我们对 nums 数组中的每个数字，都尝试在其前面添加正号和负号，最后暴力求解，统计数组中各数字组合值为 target 的情况。(该理解是错误的，我们可以使用带备忘录机制的自顶向下的 DP 方法，代码见下)

1. 回溯 O（2^N

```
private int getAllSums(int [] a, int target, int idx, int sum, int cnt) { // (2^20) 可否一试呢？理论上是可以过的
    if (idx == a.length) {                                               // n < 17 比较好 这个 2^N 的复杂度，真要命呀。。。。。
        if (sum == target) cnt++;
        return cnt; // 有 return int 代码更简洁，但是全局变量 cnt 效率更高
    }
    // for (int i = idx; i < a.length; i++) { // 为什么要画蝇添足，加个多余的 for loop 呢？
        // getAllSums(a, target, idx+1, sum + a[idx]);
        // getAllSums(a, target, idx+1, sum - a[idx]);
    // }
    return getAllSums(a, target, idx+1, sum + a[idx], cnt)
        + getAllSums(a, target, idx+1, sum - a[idx], cnt);
}
public int findTargetSumWays(int[] a, int target) {
    int n = a.length;
    return getAllSums(a, target, 0, 0, 0);
}
```

2. 解题思路与分析: dfs 记忆化搜索

```java
private int dfs(int [] a, int target, int idx, int sum) {
    String key = idx + "_" + sum;
    if (dp.containsKey(key)) return dp.get(key);
    if (idx == n) {
        if (sum == target) return 1;
        else return 0;
    }
    int add = dfs(a, target, idx+1, sum + a[idx]);
    int sub = dfs(a, target, idx+1, sum - a[idx]);
    dp.put(key, add+sub);
    return add + sub;
}
Map<String, Integer> dp = new HashMap<>();
int n;
public int findTargetSumWays(int[] a, int target) {
    n = a.length;
    return dfs(a, target, 0, 0);
}
```

- 上面的方法比较慢，下面这个效率更好一点儿

```java
private int dfs(int [] a, int sum, int idx) {
    if (idx == a.length) {
        if (sum == 0) return 1;
        else return 0;
    }
    Map<Integer, Integer> tmp = dp.get(idx);
    if (tmp != null) {
        if (tmp.containsKey(sum))
            return tmp.get(sum);
    } else {
        tmp = new HashMap<>();
        dp.put(idx, tmp);
    }
    int cnt = dfs(a, sum - a[idx], idx+1) + dfs(a, sum + a[idx], idx+1);
    tmp.put(sum, cnt);
    return cnt;
}
Map<Integer, Map<Integer, Integer>> dp = new HashMap<>();
public int findTargetSumWays(int[] nums, int target) {
    return dfs(nums, target, 0);
}
```

3. DP

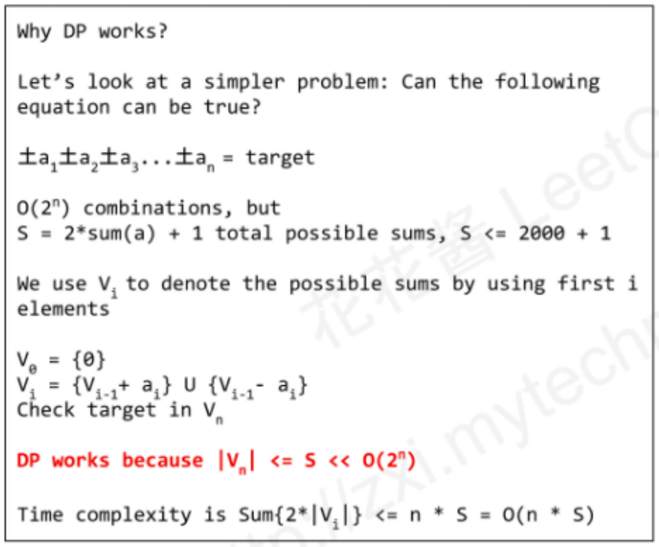
```java
// sum[p] + sum[n] = sum[nums];
// sum[p] - sum[n] = S;
// 2sum[p] = sum[nums] + S
// sum[p] = (sum[nums] +S) / 2
public int findTargetSumWays(int [] a, int S) {
    int sum = Arrays.stream(a).sum(), target = (sum + S) / 2; // 根据推导公式，计算出 target
    if (S > 0 && sum < S || S < 0 && -sum > S) return 0; // 如果和小于 S，说明无法得到解，返回 false。(注意 S 有可能为负)
    if ((sum + S) % 2 != 0) return 0; // 如果计算出的 target 不是整数，返回 false。
    int [] dp = new int [target + 1]; // dp[i] 表示在原数组中找出一些数字，并且他们的和为下标 i 的可能有多少种。
    dp[0] = 1; // 初始化 dp[0] 为 1
    for (Integer v : a)
        // for (int i = target-v; i >= 0; i--) { // 从 0 循环到 target - n，注意逆序
        //     if (dp[i] > 0)        // dp[i] 大于 0 说明，存在 dp[i] 种组合，其和为 i 的可能性
        //         dp[i+v] += dp[i]; // 既然存在和为 i 的可能，那么 i 加上当前数字的和也是存在的
        // }
        for (int i = target; i >= v; i--)  // 从 0 循环到 target - n，注意逆序
            dp[i] += dp[i-v];              // 两种写法都对
    return dp[target];
}
```

4. dp todo 我们使用 Vi 来表示数组中的前 i 个数所能求得的和的集合。初始化时

```
V0 = {0}     //表示前 0 个数的和为 0
Vi = {V(i-1) + ai} U {V(i-1) - ai}
```

Vn 就是 nums 数组所有数字的组合值之和的集合

根据上面的思路，我们知道数组中数字若全为正号其和为 sum，全为负号其和为-sum。若不选数组中任何一个数，则和为 0。因此，我们设立一个长度为 2*sum+1 的数组 ways，ways[i] 表示我们选择前 m 个数，其和可能为 i 的情况数，m = 0,1,. . . nums.length。可参考下图



Why DP works?

Let's look at a simpler problem: Can the following equation can be true?

$\pm a_1 \pm a_2 \pm a_3 \ldots \pm a_n$ = target

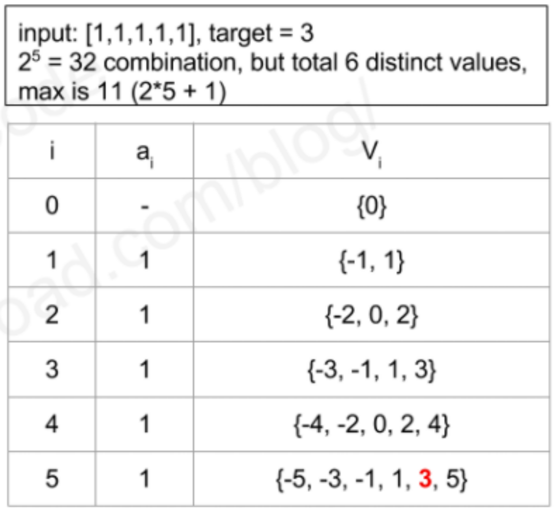
$O(2^n)$ combinations, but
S = 2*sum(a) + 1 total possible sums, S <= 2000 + 1

We use V_i to denote the possible sums by using first i elements

$V_0 = \{0\}$
$V_i = \{V_{i-1} + a_i\} \cup \{V_{i-1} - a_i\}$
Check target in V_n

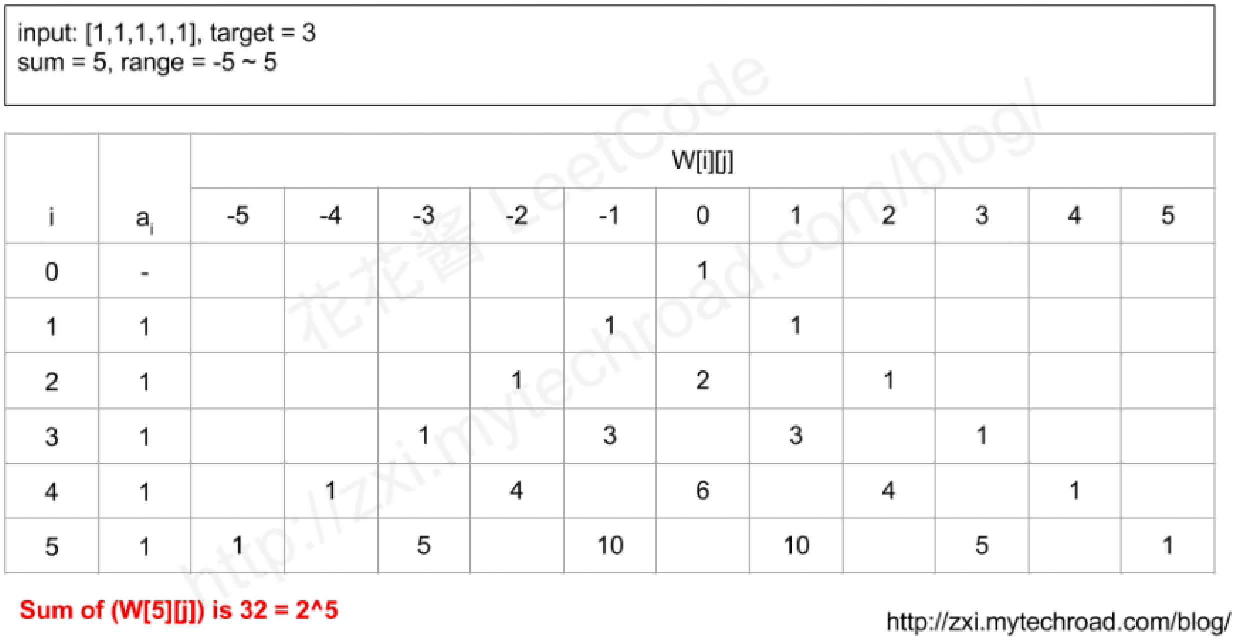
DP works because $|V_n| \le S \ll O(2^n)$

Time complexity is $\text{Sum}\{2*|V_i|\} \le n * S = O(n * S)$



input: [1,1,1,1,1], target = 3
2^5 = 32 combination, but total 6 distinct values, max is 11 (2*5 + 1)

i	a_i	V_i
0	-	{0}
1	1	{-1, 1}
2	1	{-2, 0, 2}
3	1	{-3, -1, 1, 3}
4	1	{-4, -2, 0, 2, 4}
5	1	{-5, -3, -1, 1, 3, 5}



input: [1,1,1,1,1], target = 3
sum = 5, range = -5 ~ 5

i	a_i	W[i][j]										
		-5	-4	-3	-2	-1	0	1	2	3	4	5
0	-						1					
1	1					1		1				
2	1				1		2		1			
3	1			1		3		3		1		
4	1		1		4		6		4		1	
5	1	1		5		10		10		5		1

Sum of (W[5][j]) is 32 = 2^5

http://zxi.mytechroad.com/blog/

https://www.cnblogs.com/cnoodle/p/14869498.html https://leetcode.com/problems/target-sum/discuss/97334/Java-(15-ms)-C++-(3-ms)-O(ns)-iterative-DP-solution-using-subset-sum-with-explanation/239290 http://www.noteanddata.com/leetcode-494-Target-Sum-java-solution-note.html https://www.i4k.xyz/article/gqk289/54709004 https://github.com/cherryljr/LeetCode/blob/master/Target%20Sum.java

1.2.6 518. Coin Change 2

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money. Return the number of combinations that make up that amount. If that amount of money cannot be made up by any combination of the coins, return 0. You may assume that you have an infinite number of each kind of coin. The answer is guaranteed to fit into a signed 32-bit integer.

```java
public int change(int target, int[] nums) {
    int[] dp = new int[target + 1];
    // 初始化 dp[0] 为 1
    dp[0] = 1;
    // 循环数组中所有数字
    for (int val : nums) {
        for (int i = 0; i <= target - val; i++) {
            // dp[i] 大于 0 说明，存在 dp[i] 种组合，其和为 i 的可能性
            if (dp[i] > 0) {
                // 既然存在和为 i 的可能，那么 i 加上当前数字的和也是存在的
                dp[i + val] += dp[i];
            }
        }
    }
    return dp[target];
}
```

1.2.7 1444. Number of Ways of Cutting a Pizza - Hard

Given a rectangular pizza represented as a rows x cols matrix containing the following characters: 'A' (an apple) and '.' (empty cell) and given the integer k. You have to cut the pizza into k pieces using k-1 cuts.

For each cut you choose the direction: vertical or horizontal, then you choose a cut position at the cell boundary and cut the pizza into two pieces. If you cut the pizza vertically, give the left part of the pizza to a person. If you cut the pizza horizontally, give the upper part of the pizza to a person. Give the last piece of pizza to the last person.

Return the number of ways of cutting the pizza such that each piece contains at least one apple. Since the answer can be a huge number, return this modulo $10^9 + 7$.

1. 解题思路与分析: 自底向上

   常规的矩阵 DP 做法，这里还需要通过前缀和的思想来快速获取指定范围矩阵的苹果数量。

   首先是建立状态表示数组，通过一个三维数组，分别代表矩阵左上角顶点 xy 坐标和需要分配的人数，数组值表示分该状态下的配方案数；

   然后是进行状态转移，从右下角开始枚举所有以该点为状态中左上角的状态，再从低到高枚举需要分配的人数，接着进行切的操作，可以横着切和竖着切，分别枚举所有可能的切除的长度，当前状态的方案数需要从切除后剩下的矩阵状态中进行转移累加。

   最后返回以原矩阵左上角为顶点的，分配人数为 k 的方案数即可。

   这里为什么需要将状态表示中的 xy 设定为矩阵的左上角，还有为什么苹果数的前缀和也是求的右下角的前缀和呢？

   因为题意中的切除操作后，要将上半部分或者左半部分给分掉，所以只有右下部分是剩余状态的，我们需要从切除之前的状态获取剩余状态。

```
public int ways(String[] pizza, int p) {
    int mod = (int)1e9 + 7;
    int m = pizza.length, n = pizza[0].length();
    int [][] cnt = new int [m+1][n+1]; // 苹果数的前缀和，用于快速获得在指定矩阵范围内的苹果数量，两个维度也分别是左上角的 x、y
    for (int i = m-1; i >= 0; i--)
        for (int j = n-1; j >= 0; j--)
            cnt[i][j] = cnt[i+1][j] + cnt[i][j+1] - cnt[i+1][j+1] + (pizza[i].charAt(j) == 'A' ? 1 : 0);
    int [][][] dp = new int [m+1][n+1][p+1]; // 状态数组，三个维度分别表示以 x、y 为左上角的矩阵中，分给 k 个人，元素值表示方案数
    for (int i = m-1; i >= 0; i--)      // 遍历矩阵，获取指定左上角矩阵中范围内的苹果数量
        for (int j = n-1; j >= 0; j--) { // 从右下角开始，向左上角开始枚举所有状态
            if (cnt[i][j] > 0) dp[i][j][1] = 1; // 如果这个范围矩阵内存在苹果，那么这个矩阵肯定可以分给 1 个人，且方案数为 1
            for (int k = 2; k <= p; k++) {      // 枚举所有人数状态下的方案，前面已经判断了人数为 1 的状态，所以这里只需要从 2 开始枚举
                for (int x = m-1-i; x >= 0; x--)     // 横着切，枚举所有切法
                    if (cnt[i][j] - cnt[i+x][j] > 0) // 如果当前切掉的矩阵内存在苹果，则可以进行状态转移
                        dp[i][j][k] = (dp[i][j][k] + dp[i+x][j][k-1]) % mod;
                for (int y = n-1-j; y >= 0; y--)     // 竖着切
                    if (cnt[i][j] - cnt[i][j+y] > 0)
                        dp[i][j][k] = (dp[i][j][k] + dp[i][j+y][k-1]) % mod;
            }
        }
    return (int)dp[0][0][p];
}
```

2. 解题思路与分析: 自顶向下

   先用 dp 方法求出以（i,j）位置为右下角，左上角为（0,0）的区域的苹果数量

   建立 3 维数组，dp[i][j][k] 表示切完 k 次后，剩余蛋糕左上角在 i, j 位置时的方案数

   初始化，dp[1], 1, 1 = 1

   样本维度为切的次数 k

   状态维度，这次切之前的状态（蛋糕左上角位置 i, j）

   状态转移，这次切完后蛋糕左上角位置（横向切，ni，j；竖向切，i, nj，切的次数 +1）

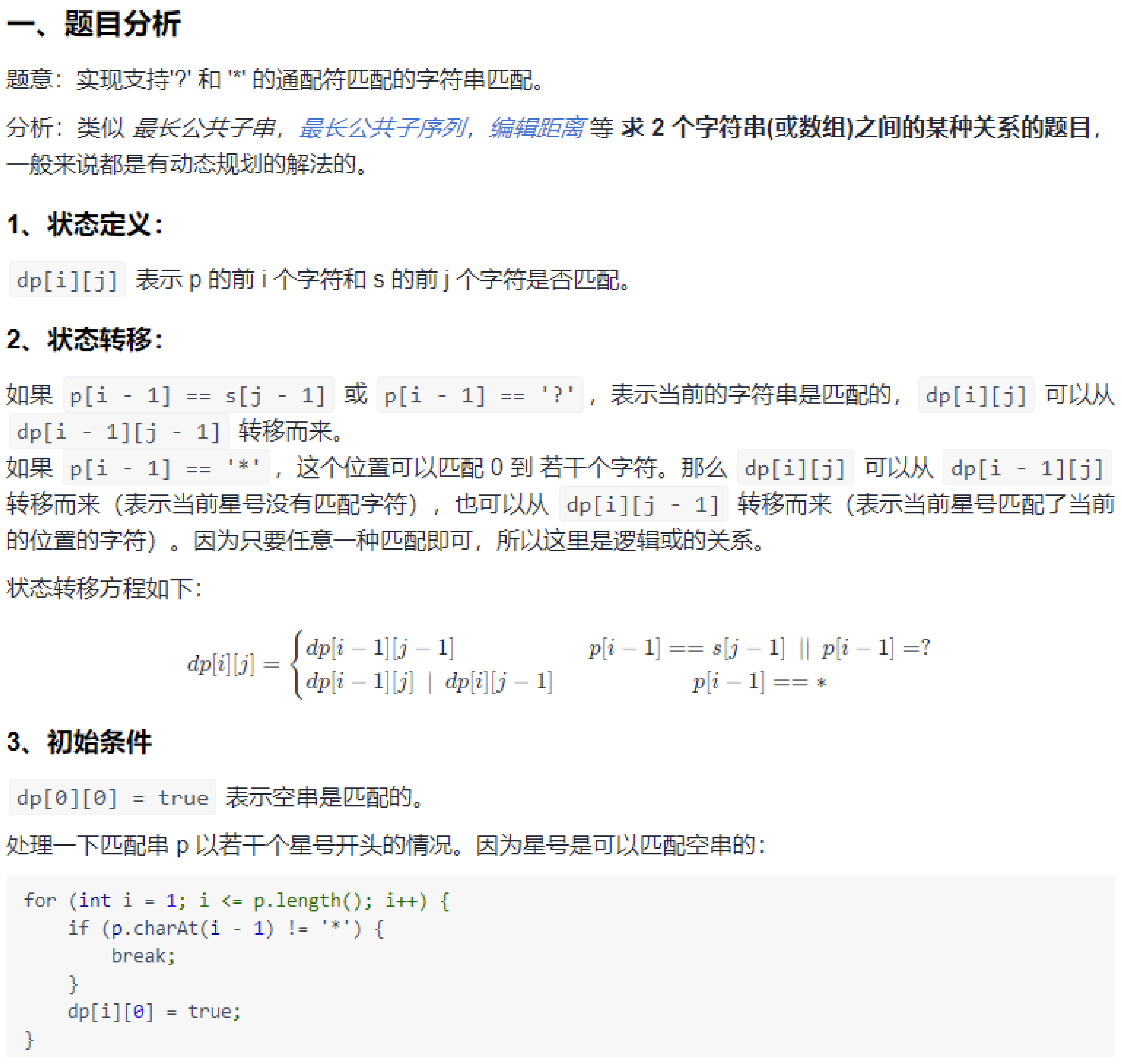
   转移条件：切出去的蛋糕当中有苹果（用上面求得的苹果数量，dp 公式求得）

   最后求结果总和：最后的一块蛋糕中有苹果，sum += dp[i][j][k-1]

[1]DEFINITION NOT FOUND.

```
public int ways(String[] pizza, int p) { // 自顶向下: 与自底向上相比
    int mod = (int)1e9 + 7;
    int m = pizza.length, n = pizza[0].length();
    int [][] cnt = new int [m+1][n+1];  // 苹果数的前缀和，用于快速获得在指定矩阵范围内的苹果数量，两个维度也分别是左上角的 x、y
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            cnt[i][j] = cnt[i-1][j] + cnt[i][j-1] - cnt[i-1][j-1] + (pizza[i-1].charAt(j-1) == 'A' ? 1 : 0);
    int [][][] dp = new int [m+1][n+1][p]; // dp[i][j][k] 表示切完 k 次后，剩余蛋糕左上角 在 i,j 位置时的方案数
    dp[1][1][0] = 1; // 初始值是为了程序的运行，
    for (int k = 1; k < p; k++)
        for (int i = 1; i <= m; i++)
            for (int j = 1; j <= n; j++) {
                System.out.println("(dp[i][j][k-1] == 0) : " + (dp[i][j][k-1] == 0) );
                if (dp[i][j][k-1] == 0) continue; // 上一次 cut 完后，剩余蛋糕左上角在 i,j
                for (int x = i+1; x <= m; x++)   // 横向切，切完后的剩余左上角为 x, j
                    if (cnt[x-1][n] - cnt[i-1][n] - cnt[x-1][j-1] + cnt[i-1][j-1] > 0)
                        dp[x][j][k] = (dp[x][j][k] + dp[i][j][k-1]) % mod;
                for (int y = j+1; y <= n; y++)  // 竖向切
                    if (cnt[m][y-1] - cnt[m][j-1] - cnt[i-1][y-1] + cnt[i-1][j-1] > 0)
                        dp[i][y][k] = (dp[i][y][k] + dp[i][j][k-1]) % mod;
            }
    long ans = 0;
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            if (cnt[m][n] - cnt[i-1][n] - cnt[m][j-1] + cnt[i-1][j-1] > 0) // 先前并没有确认切的结果有效，即最后剩下的那块是否有苹果
                ans = (ans + dp[i][j][p-1]) % mod;                         // 统计结果的时候，要先确保有效
    return (int)ans;
}
```

1.3 字符串、数组等双序列



一、题目分析

题意：实现支持'?' 和 '*' 的通配符匹配的字符串匹配。

分析：类似 *最长公共子串*，*最长公共子序列*，*编辑距离* 等 **求 2 个字符串(或数组)之间的某种关系的题目**，一般来说都是有动态规划的解法的。

1、状态定义：

`dp[i][j]` 表示 p 的前 i 个字符和 s 的前 j 个字符是否匹配。

2、状态转移：

如果 `p[i - 1] == s[j - 1]` 或 `p[i - 1] == '?'`，表示当前的字符串是匹配的，`dp[i][j]` 可以从 `dp[i - 1][j - 1]` 转移而来。
如果 `p[i - 1] == '*'`，这个位置可以匹配 0 到 若干个字符。那么 `dp[i][j]` 可以从 `dp[i - 1][j]` 转移而来（表示当前星号没有匹配字符），也可以从 `dp[i][j - 1]` 转移而来（表示当前星号匹配了当前的位置的字符）。因为只要任意一种匹配即可，所以这里是逻辑或的关系。

状态转移方程如下:

$$dp[i][j] = \begin{cases} dp[i-1][j-1] & p[i-1] == s[j-1] \;||\; p[i-1] = ? \\ dp[i-1][j] \mid dp[i][j-1] & p[i-1] == * \end{cases}$$

3、初始条件

`dp[0][0] = true` 表示空串是匹配的。

处理一下匹配串 p 以若干个星号开头的情况。因为星号是可以匹配空串的：

```
for (int i = 1; i <= p.length(); i++) {
    if (p.charAt(i - 1) != '*') {
        break;
    }
    dp[i][0] = true;
}
```

1.3.1 题目拓展

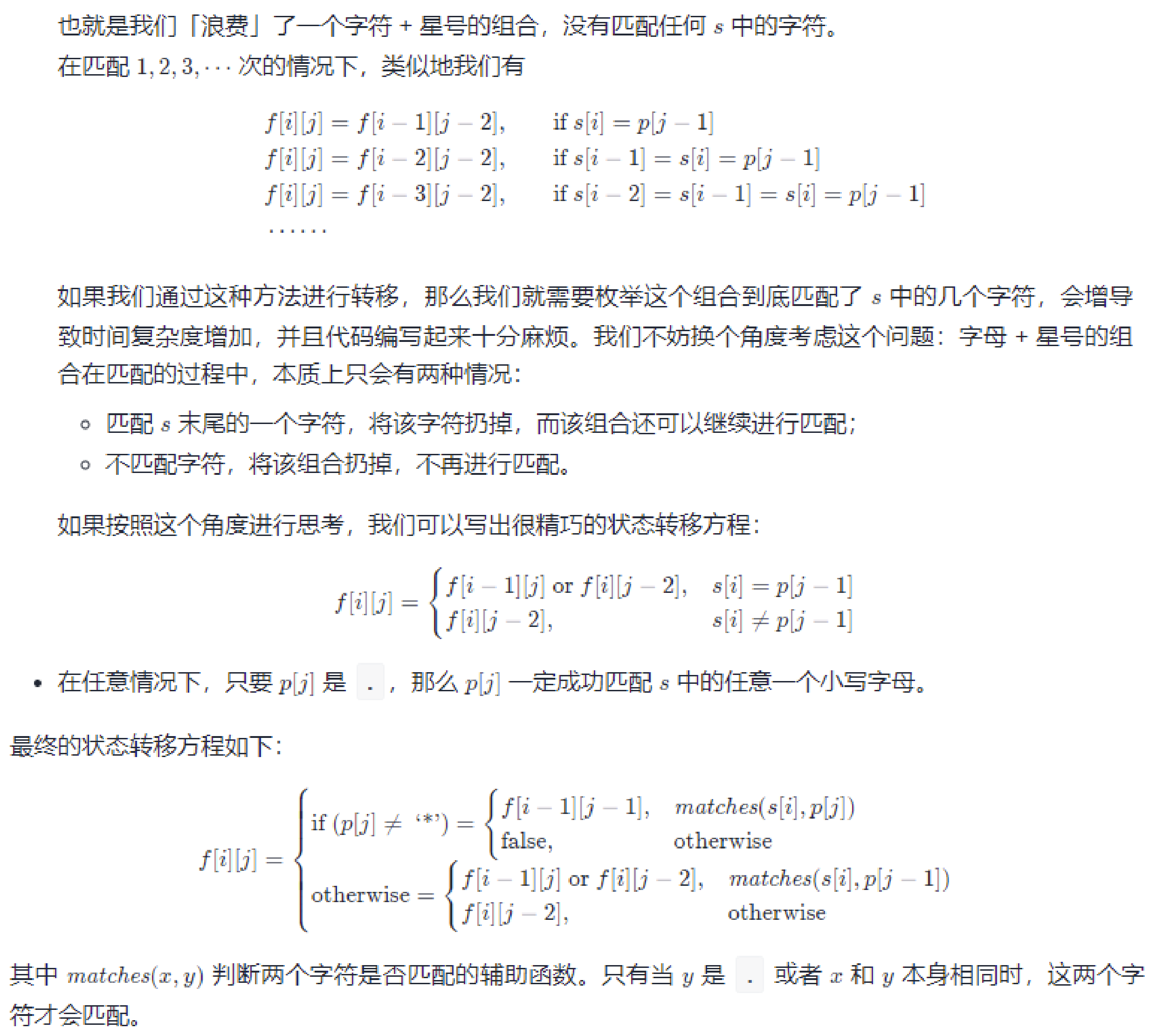
1. 718. 最长重复子数组（类似题目，只是由字符串变为数组）
2. 72. 编辑距离
3. 1143. 最长公共子序列
4. 10. 正则表达式匹配
5. 583. 两个字符串的删除操作
6. 727. 最小窗口子序列

你会发现这些都是求 2 个字符串 (或数组) 之间的某种关系的题目

1.3.2 10. Regular Expression Matching - Hard

Given an input string s and a pattern p, implement regular expression matching with support for ’.’ and ’*’ where: ’.’ Matches any single character. ’*’ Matches zero or more of the preceding element. The matching should cover the entire input string (not partial).

1. 解题思路与分析



也就是我们「浪费」了一个字符 + 星号的组合，没有匹配任何 s 中的字符。

在匹配 $1, 2, 3, \cdots$ 次的情况下，类似地我们有

$$\begin{aligned}
&f[i][j] = f[i-1][j-2], && \text{if } s[i] = p[j-1] \\
&f[i][j] = f[i-2][j-2], && \text{if } s[i-1] = s[i] = p[j-1] \\
&f[i][j] = f[i-3][j-2], && \text{if } s[i-2] = s[i-1] = s[i] = p[j-1] \\
&\cdots\cdots
\end{aligned}$$

如果我们通过这种方法进行转移，那么我们就需要枚举这个组合到底匹配了 s 中的几个字符，会增导致时间复杂度增加，并且代码编写起来十分麻烦。我们不妨换个角度考虑这个问题：字母 + 星号的组合在匹配的过程中，本质上只会有两种情况：

- 匹配 s 末尾的一个字符，将该字符扔掉，而该组合还可以继续进行匹配；
- 不匹配字符，将该组合扔掉，不再进行匹配。

如果按照这个角度进行思考，我们可以写出很精巧的状态转移方程：

$$f[i][j] = \begin{cases} f[i-1][j] \text{ or } f[i][j-2], & s[i] = p[j-1] \\ f[i][j-2], & s[i] \neq p[j-1] \end{cases}$$

- 在任意情况下，只要 $p[j]$ 是 `.`，那么 $p[j]$ 一定成功匹配 s 中的任意一个小写字母。

最终的状态转移方程如下：

$$f[i][j] = \begin{cases} \text{if } (p[j] \neq \text{‘*’}) = \begin{cases} f[i-1][j-1], & matches(s[i], p[j]) \\ \text{false}, & \text{otherwise} \end{cases} \\ \text{otherwise} = \begin{cases} f[i-1][j] \text{ or } f[i][j-2], & matches(s[i], p[j-1]) \\ f[i][j-2], & \text{otherwise} \end{cases} \end{cases}$$

其中 $matches(x, y)$ 判断两个字符是否匹配的辅助函数。只有当 y 是 `.` 或者 x 和 y 本身相同时，这两个字符才会匹配。

```java
public boolean isMatch(String s, String p) {
    int m = s.length();
    int n = p.length();
    boolean[][] f = new boolean[m + 1][n + 1];
    f[0][0] = true;
    for (int i = 0; i <= m; ++i)
        for (int j = 1; j <= n; ++j)
            if (p.charAt(j - 1) == '*') {
                f[i][j] = f[i][j - 2];
                if (matches(s, p, i, j - 1))
                    f[i][j] = f[i][j] || f[i - 1][j];
            } else {
                if (matches(s, p, i, j))
                    f[i][j] = f[i - 1][j - 1];
            }
    return f[m][n];
}
public boolean matches(String s, String p, int i, int j) {
    if (i == 0) return false;
    if (p.charAt(j - 1) == '.') return true;
    return s.charAt(i - 1) == p.charAt(j - 1);
}
```

1.3.3 将一个数组分为两个部分，分别求和 S1 与 S2，使得 |S1-S2| 最小

一、题目描述：

将一个数组分为两个部分，分别求和$S1$与$S2$，使得$|S1-S2|$最小。

解法：动态规划

解法描述：转化为求数组的一个子集，使得这个子集中的元素的和尽可能接近sum/2，其中sum为数组中所有元素的和。这样转换之后这个问题就很类似0-1背包问题了：在n件物品中找到m件物品，他们的可以装入背包中，且总价值最大不过这里不考虑价值，就考虑使得这些元素的和尽量接近sum/2。

状态转移方程：
$dp[i][j]$表示前i件物品中，总和最接近j的所有物品的总和，其中包括两种情况：

- 第i件物品没有包括在其中
- 第i件物品包括在其中
  如果第i件物品没有包括在其中，则$dp[i][j] = dp[i-1][j]$
  如果第i件物品包括在其中，则$dp[i][j] = dp[i-1][j-vec[i]]$

当然，这里要确保j-vec[i] >= 0。

所以状态转移方程为：

$$dp[i][j] = max(dp[i-1][j], dp[i-1][j-vec[i]] + vec[i]);$$

```
public static int getMaxDiff(int[] array) {
    int sum = Arrays.stream(array).sum();
    int length = array.length;
    int [][] f = new int[length+1][sum/2+1];
    for (int i = 0; i < length; i++)
        for (int j = 1; j <  = sum/2; j++) {
            f[i+1][j]  =  f[i][j];
            if (array[i] <= j && f[i][j-array[i]] + array[i] > f[i][j])
                f[i+1][j] = f[i][j-array[i]] + array[i];
        }
    return sum-2*f[length][sum/2];
}
```

1.3.4 给定一个序列，不保证有序，求这个序列的最长等差序列的长度。

二、题目描述

给定一个序列，不保证有序，求这个序列的最长等差序列的长度。

解法：动态规划

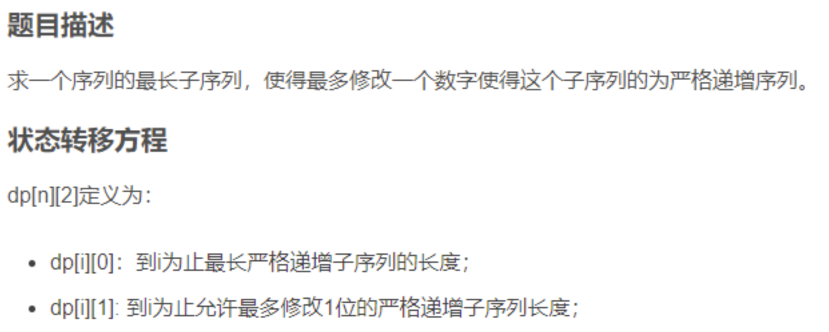
解法描述：令$f[i][j]$ 表示以i为结尾的某子序列（该子序列的等差为j)的最大长度；

状态转移方程：

$$f[i][j] = f[i-1][Num[i] - Num[j]] + 1, i > 0$$

```
private static int lengthOfLongest(int[] set){
    Arrays.sort(set);
    int n = set.length;
    if (n <= 2) return n;
    int llap = 2;
    int[][] dp = new int[n][n];
    for (int i=0; i<n; i++) dp[i][n-1] = 2;
    for (int j=n-2; j>=1; j--) {
        int i=j-1, k=j+1;
        while (i>=0 && k<=n-1) {
            if (set[i] + set[k] < 2 * set[j])
                k++;
            else if (set[i] + set[k] > 2 * set[j]) {
                dp[i][j] = 2;
                i--;
            } else {
                dp[i][j] = dp[j][k] + 1;
                llap = Math.max(llap, dp[i][j]);
                i--;
                k++;
            }
        }
        while (i >= 0) {
            dp[i][j] = 2;
            i--;
        }
    }
    return llap;
}
```

1.3.5 求一个序列的最长子序列，使得最多修改一个数字使得这个子序列的为严格递增序列



题目描述

求一个序列的最长子序列，使得最多修改一个数字使得这个子序列的为严格递增序列。

状态转移方程

dp[n][2]定义为：

- dp[i][0]：到i为止最长严格递增子序列的长度；
- dp[i][1]: 到为止允许最多修改1位的严格递增子序列长度；

```
private static int getMaxLength(int[] arr){
    if (arr.length <= 2) return arr.length;
    int[][] dp = new int[arr.length][2];
    dp[0][0] = 1;
    dp[0][1] = 1;
    for (int i = 1; i < arr.length; i++) {
        dp[i][0] = dp[i-1][0]+1;
        if (arr[i] <= arr[i-1])
            dp[i][0]--;
        if (dp[i-1][0] == dp[i-1][1] && arr[i] <= arr[i-1]) {// 说明前面还没有改的
            dp[i][1] = dp[i][0] + 1;
            arr[i] = arr[i-1]+1;
        } else {//说明前面已经改动或者 arr[i] <= arr[i-1]
            if (arr[i] > arr[i-1]) {
                //判断前面是否已经改动
                dp[i][1] = dp[i-1][1]+1;
                if (dp[i-1][0] != dp[i-1][1])
                    dp[i][1]--;
            } else
                dp[i][1] = dp[i-1][1];
        }
    }
    return dp[arr.length-1][1];
}
```

1.3.6 115. Distinct Subsequences - Hard

Given two strings s and t, return the number of distinct subsequences of s which equals t.

A string's subsequence is a new string formed from the original string by deleting some (can be none) of the characters without disturbing the remaining characters' relative positions. (i.e., "ACE" is a subsequence of "ABCDE" while "AEC" is not).

It is guaranteed the answer fits on a 32-bit signed integer.

1. 解题思路与分析 这道题不是求两个字符串是匹配，而是判断 S 有多少种方式可以得到 T。但其实还是动态规划，我们一个定义二维数组 dp，dp[i][j] 为字符串 s(0,i) 变换到 t(0,j) 的变换方法的个数。

   如果 S[i]==T[j]，那么 dp[i][j] = dp[i-1][j-1] + dp[i-1][j]

   意思是：如果当前 S[i]==T[j]，那么当前这个字符即可以保留也可以抛弃，所以变换方法等于保留这个字符的变换方法加上不用这个字符的变换方法，

   dp[i-1][j-1] 为保留这个字符时的变换方法个数，dp[i-1][j] 表示抛弃这个字符时的变换方法个数。

   如果 S[i]!=T[i]，那么 dp[i][j] = dp[i-1][j]，意思是如果当前字符不等，那么就只能抛弃当前这个字符。

```java
public int numDistinct(String ss, String tt) {
    int m = ss.length(), n = tt.length();
    char [] s = ("#"+ss).toCharArray();
    char [] t = ("#"+tt).toCharArray();
    int [][] dp = new int [m+1][n+1];
    dp[0][0] = 1;
    for (int j = 1; j <= n; j++) // 注意这两行初始状态的设置
        dp[0][j] = 0;
    for (int i = 1; i <= m; i++)
        dp[i][0] = 1;
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            if (s[i] == t[j])
                dp[i][j] = dp[i-1][j-1] + dp[i-1][j];
            else dp[i][j] = dp[i-1][j];
    return dp[m][n];
}
```

1.4 区间型 DP

1.4.1 2019. The Score of Students Solving Math Expression - Hard 有人说这是区间 dp，无感

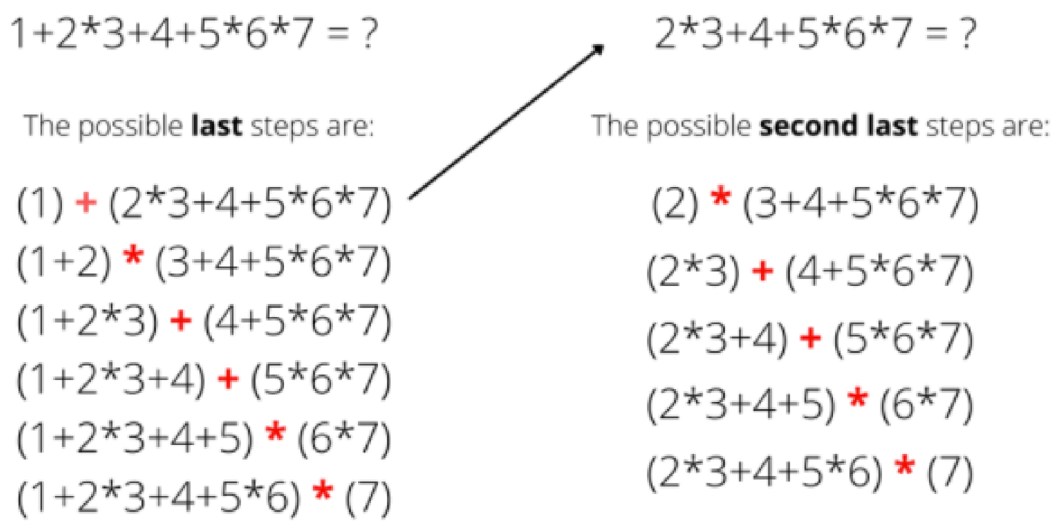
You are given a string s that contains digits 0-9, addition symbols '+', and multiplication symbols '*' only, representing a valid math expression of single digit numbers (e.g., 3+5*2). This expression was given to n elementary school students. The students were instructed to get the answer of the expression by following this order of operations:

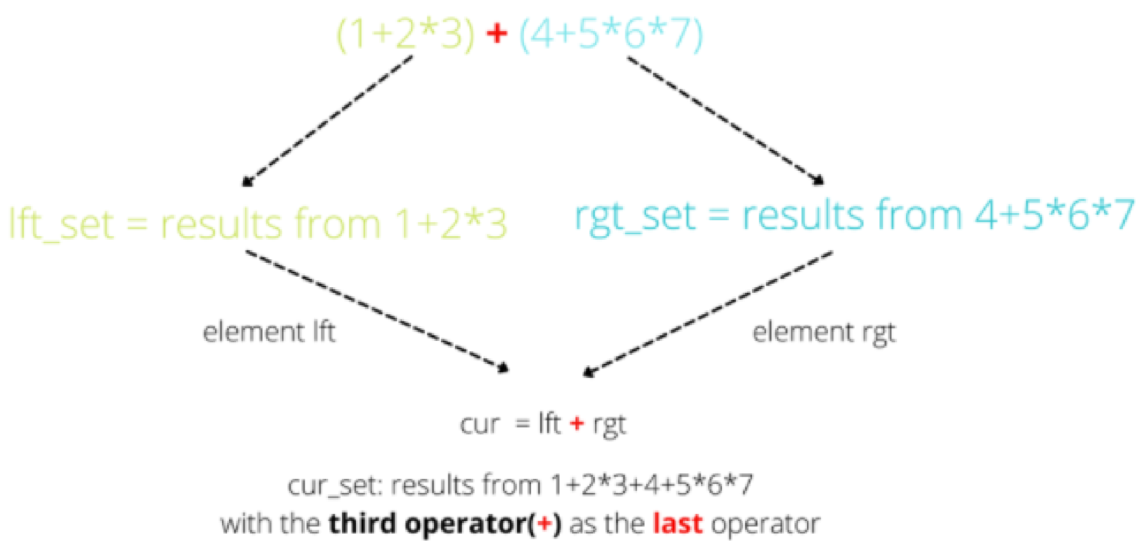
Compute multiplication, reading from left to right; Then, Compute addition, reading from left to right. You are given an integer array answers of length n, which are the submitted answers of the students in no particular order. You are asked to grade the answers, by following these rules:

If an answer equals the correct answer of the expression, this student will be rewarded 5 points; Otherwise, if the answer could be interpreted as if the student applied the operators in the wrong order but had correct arithmetic, this student will be rewarded 2 points; Otherwise, this student will be rewarded 0 points. Return the sum of the points of the students.

1. 解题思路与分析

- 思路是记忆化搜索。先求一下正确答案，然后开始算所有可能得到的错误答案。枚举运算符，然后递归求解两边可能的答案，汇总成当前表达式可能得到的答案。用记忆化的方式避免重复计算。
- 时间复杂度 O(l_s^3+l_A))，空间 O(l_s^2)。注意有 1000 这个限制，上面所说的复杂度的常数是 1000^2




(1+2*3) + (4+5*6*7)
lft_set = results from 1+2*3
rgt_set = results from 4+5*6*7
element lft
element rgt
cur = lft + rgt
cur_set: results from 1+2*3+4+5*6*7
with the third operator(+) as the last operator

```
private int compute(String t) {
    ArrayDeque<Integer> st = new ArrayDeque<>();
    char [] s = t.toCharArray();
    for (int i = 0; i < s.length; i++) {
        char c = s[i];
        if (Character.isDigit(c))
            if (i > 0 && s[i-1] == '*')
                st.push(st.pop() * (c-'0'));
            else st.push(c-'0');
    }
    int ans = 0;
    while (!st.isEmpty())
        ans += st.pop();
    return ans;
}
Set<Integer> dfs(String t, int l, int r, Set<Integer> [][] f) {
    if (f[l][r] != null) return f[l][r]; // 有记忆则调取记忆
    char [] s = t.toCharArray();
    int n = t.length(), v = 0;
    f[l][r] = new HashSet<>();
    if (l == r) {
        f[l][r].add(s[l] - '0');
        return f[l][r];
    }
    for (int i = l+1; i < r; i++)
        if (!Character.isDigit(s[i])) { // 递归求解左右两边可能算出的答案
            Set<Integer> left = dfs(t, l, i-1, f);
            Set<Integer> right = dfs(t, i+1, r, f);
            for (Integer va : left)
                for (Integer vb : right) {
                    if (s[i] == '*') v = va * vb;
                    else v = va + vb;
                    if (v >= 0 && v <= 1000) f[l][r].add(v);
                }
        }
    return f[l][r];
}
public int scoreOfStudents(String s, int [] num) {
    int m = num.length, res = compute(s), n = s.length(), ans = 0;
    Set<Integer> [][] f = new HashSet[n][n]; // 第一次见, 学习一下
    dfs(s, 0, n-1, f);
    Set<Integer> can = f[0][n-1];         // candidates: of wrong answers
    for (Integer v : num)
        if (v == res) ans += 5;
        else if (can.contains(v)) ans += 2;
    return ans;
}
```

1.4.2 312. Burst Balloons 区间型动态规划的典型代表仍是典型，最好不扔掉

You are given n balloons, indexed from 0 to n - 1. Each balloon is painted with a number on it represented by an array nums. You are asked to burst all the balloons. If you burst the ith balloon, you will get nums[i - 1] * nums[i] * nums[i + 1] coins. If i - 1 or i + 1 goes out of bounds of the array, then treat it as if there is a balloon with a 1 painted on it. Return the maximum coins you can collect by bursting the balloons wisely.

```
public int maxCoins(int[] nums) {
    int n = nums.length;
    int [][]  dp = new int [n+2][n+2];
    int [] arr = new int [n+2];
    System.arraycopy(nums, 0, arr, 1, n);
    arr[0] = arr[n+1] = 1;  // [0, n+1] ==> [1, n]
    int j = 0;
    for (int len = 1; len <= n; len++) { // [1, n]
        for (int i = 1; i+len-1 <= n; i++) { // [1, n]
            j = i + len - 1;
            for (int k = i; k <= j; k++)
                dp[i][j] = Math.max(dp[i][j], dp[i][k-1] + dp[k+1][j] + arr[i-1]*arr[k]*arr[j+1]);
        }
    }
    return dp[1][n];
}
// 0    0    0    0    0    0
// 0    3    30   159  167  0
// 0    0    15   135  159  0
// 0    0    0    40   48   0
// 0    0    0    0    40   0
// 0    0    0    0    0    0
private int memorizedSearch(int [] arr, int x, int y) {
    if (dp[x][y] > 0) return dp[x][y];
    // if (x == y) return dp[x][y] = arr[x]; // 没有这些个边际条件
    // if (x == y-1)
    //     return dp[x][y] = arr[x] * arr[y] + Math.max(arr[x], arr[y]);
    int max = 0;
    for (int i = x; i <= y; i++) {
        max = Math.max(max, memorizedSearch(arr, x, i-1) + memorizedSearch(arr, i+1, y) + arr[x-1]*arr[i]*arr[y+1]);
    }
    return dp[x][y] = max;
}
int [][] dp;
int n;
public int maxCoins(int[] nums) {
    int n = nums.length + 2;
    dp = new int [n][n];
    int [] arr = new int [n];
    System.arraycopy(nums, 0, arr, 1, n-2);
    arr[0] = arr[n-1] = 1;
    return memorizedSearch(arr, 1, n-2);
}
```

1.5 多维数个数、数种类数的

1.5.1 1866. Number of Ways to Rearrange Sticks With K Sticks Visible - Hard

There are n uniquely-sized sticks whose lengths are integers from 1 to n. You want to arrange the sticks such that exactly k sticks are visible from the left. A stick is visible from the left if there are no longer sticks to the left of it.

For example, if the sticks are arranged [1,3,2,5,4], then the sticks with lengths 1, 3, and 5 are visible from the left. Given n and k, return the number of such arrangements. Since the answer may be large, return it modulo 109 + 7.

```
// dp[i][j] 表示前面 i 根木棍可以看到 j 根
// 设 dp[i][j] 表示从高度为 1, 2, ..., i 的木棍中，高度逐渐递减地插入新的木棍，从左侧看恰好看到 k 根木棍的方案数。
// 后面说看到 ith 根，不是指从小到大的第 ith 根棍子，而是指 ith 这个位置上的棍子
// 如果可以看到 ith 根的话，那么数量为 dp[i-1][j-1]
// 如果看不到 ith 的话，那么取前面 (i-1) 里面任意一个出来放在 ith 的最后，接下来就是从前面 i-1 个棍子里面看到 j 根，所以结果是 (i-1)* dp[i-1][j]
public int rearrangeSticks(int n, int k) {
    int mod = (int)1e9 + 7;
    long [][] dp = new long [n+1][k+1];
    dp[0][0] = 1;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= Math.min(n, k); j++)
            dp[i][j] = (dp[i-1][j-1] + (dp[i-1][j] * (i-1)) % mod) % mod;
    return (int)dp[n][k];
}
```

- dfs + memo

```
long mod = 1000_000_000 + 7;
long[][] dp;
public int rearrangeSticks(int n, int k) {
    dp = new long[n + 1][k + 1];
    long ans = dfs(n, k);
    return (int) (ans % mod);
}
long dfs(int n, int k) {
    if(n < k || k == 0) return 0;
    if(n == k) return 1;
    if(dp[n][k] != 0) return dp[n][k];
    long ans = 0;
    // instead of iterating for every stick
    // we are just multiplying number of ways with (n - 1)
    ans += (((n - 1) * dfs(n - 1, k)) % mod);
    ans %= mod;
    ans += dfs(n - 1, k - 1);
    ans %= mod;
    return dp[n][k] = ans;
}
```

1.5.2 1916. Count Ways to Build Rooms in an Ant Colony - Hard

You are an ant tasked with adding n new rooms numbered 0 to n-1 to your colony. You are given the expansion plan as a 0-indexed integer array of length n, prevRoom, where prevRoom[i] indicates that you must build room prevRoom[i] before building room i, and these two rooms must be connected directly. Room 0 is already built, so prevRoom[1] = -1. The expansion plan is given such that once all the rooms are built, every room will be reachable from room 0.

You can only build one room at a time, and you can travel freely between rooms you have already built only if they are connected. You can choose to build any room as long as its previous room is already built.

Return the number of different orders you can build all the rooms in. Since the answer may be large, return it modulo 109 + 7.

对每个节点，可根据所有以其子节点为根的树的节点及排列数量，计算出以当前节点为根的树的节点及排列数量。

本题求解过程涉及较多前置知识点，包括排列组合、乘法逆元、快速乘方等

读者需要掌握如下进阶知识点才能解决本题：

- 排列数计算。假设有 a_0 个物品 0，a_1 个物品 1，…，a_{n-1} 个物品 $n-1$，我们需要将它们排成一排，那么方案数为

$$\frac{(a_0 + a_1 + \cdots + a_{n-1})!}{a_0! \cdot a_1! \cdot \cdots \cdot a_{n-1}!}$$

- 乘法逆元。如果需要计算 $\frac{b}{a}$ 对 m 取模的值，b 和 a 均为表达式（例如上面排列数的分子与分母）并且它们实际的值非常大，无法直接进行计算，那么我们可以求出 a 在模 m 意义下的乘法逆元 a^{-1}，此时有

$$\frac{b}{a} \equiv b \cdot a^{-1} \pmod m$$

  这样将除法运算转化为乘法运算，就可以在计算的过程中进行取模了。
  乘法逆元的具体计算方法可以参考题解的最后一节「进阶知识点：乘法逆元」。

方法一：动态规划

提示 1

由于除了 0 号房间以外的每一个房间 i 都有唯一的 $prevRoom[i]$，而 0 号房间没有 $prevRoom[i]$，并且我们可以从 0 号房间到达每个房间，即任意两个房间都是可以相互到达的，因此：

> 如果我们把所有的 n 间房间看成 n 个节点，任意的第 $i(i > 0)$ 号房间与 $prevRoom[i]$ 号房间之间连接一条有向边，从 $prevRoom[i]$ 指向 i，那么它们就组成了一棵以 0 号房间为根节点的树，并且树上的每条有向边总是从父节点指向子节点。

提示 2

根据题目要求，对于任意的 $i > 0$，我们必须先构筑 $prevRoom[i]$，再构筑 i，而我们根据提示 1 构造的树中恰好有一条从 $prevRoom[i]$ 指向 i 的边，因此：

> 题目中的要求与拓扑排序的要求是等价的。

也就是说，构筑所有房间的不同顺序的数目，等于**我们构造的树的不同拓扑排序的方案数。**

思路与算法

我们可以使用动态规划的方法求出方案数。

设 $f[i]$ 表示以节点 i 为根的子树的拓扑排序方案数，那么：

- 任意一种拓扑排序中的第一个节点一定是节点 i；
- 假设节点 i 有子节点 $ch_{i,0}, ch_{i,1}, \cdots, ch_{i,k}$，那么以 $ch_{i,0}, ch_{i,1}, \cdots, ch_{i,k}$ 为根的这 $k+1$ 棵子树，它们均有若干种拓扑排序的方案。如果我们希望把这些子树的拓扑排序方案「简单地」合并到一起，可以使用乘法原理，即方案数为：

$$f[ch_{i,0}] \times f[ch_{i,1}] \times \cdots \times f[ch_{i,k}]$$

乘法原理会忽略**子树之间**拓扑排序的顺序，所以我们还需要考虑这一层关系。记 $cnt[i]$ 表示以节点 i 为根的子树中节点的个数，那么除了拓扑排序中第一个节点为 i 以外，还剩余 $cnt[i]-1$ 个位置。我们需要在 $cnt[i]-1$ 个位置中，分配 $cnt[ch_{i,0}]$ 个给以 $ch_{i,0}$ 为根的子树，放置该子树的一种拓扑排序；分配 $cnt[ch_{i,1}]$ 个给以 $ch_{i,1}$ 为根的子树，放置该子树的一种拓扑排序。以此类推，分配的方案数乘以上述乘法原理得到的方案数，即为 $f[i]$。根据「前言」部分，我们知道分配的方案数为：

$$\frac{(cnt[i]-1)!}{cnt[ch_{i,0}]! \times cnt[ch_{i,1}]! \times \cdots \times cnt[ch_{i,k}]!}$$

因此，我们就可以得到状态转移方程：

$$f[i] = \left(\prod_{ch} f[ch]\right) \times \frac{(cnt[i]-1)!}{\prod_{ch} cnt[ch]!}$$

细节

当节点 i 为叶节点时，它没有子节点，因此对应着 1 种拓扑排序的方案，即 $f[i]=1$。

状态转移方程中存在除法，因此我们需要使用「前言」部分的乘法逆元将其转换为除法。

观察状态转移方程，它包含一些阶乘以及阶乘的乘法逆元，因此我们可以将它们全部预处理出来，这样就可以均摊 $O(1)$ 的时间在一对「父-子」节点之间进行状态转移。

```java
// 快速计算 x^y 的乘方
public int quickMul(int x , int y) {
    long res = 1, cur = x;
    while (y > 0) {
        if ((y & 1) == 1)
            res = res * cur % mod;
        cur = cur * cur % mod;
        y >>= 1;
    }
    return (int)res;
}
// 深度优先搜索，返回以当前节点为根的子树节点个数 及 内部排列数
public int [] dfs (int idx) {
    if (!map.containsKey(idx)) return new int [] {1, 1}; // 子节点，节点个数及内部排列数均为 1
    int cnt = 1, res = 1;       //  子树的节点个数、内部排列数
    for (Integer node : map.get(idx)) {
      int [] cur = dfs(node); // 递归得到子节点对应树的节点个数和排列数
        cnt += cur[0];
        res = (int)((long)res * cur[1] % mod * inv[cur[0]] % mod);
    }
    res = (int)((long)res * fac[cnt-1] % mod);
    return new int [] {cnt, res};
}
int mod = (int)1e9 + 7;
Map<Integer, List<Integer>> map = new HashMap<>();
int [] fac, inv;
public int waysToBuildRooms(int[] prevRoom) {
    int n = prevRoom.length;
    // 求阶乘数列及对应逆元
    this.fac = new int [n]; // fac[i]=i!
    this.inv = new int [n]; // inv[i]=i!^(-1)
    fac[0] = inv[0] = 1;
    for (int i = 1; i < n; i++) {
        fac[i] = (int)((long)fac[i-1] * i % mod);
        inv[i] = quickMul(fac[i], mod - 2); // 费马小定理: (fac[i]^(-1))%mod = (fac[i]^(mod-2))%mod
    }
    // 记录各个节点与子节点之间的边
    for (int i = 1; i < n; i++)
        map.computeIfAbsent(prevRoom[i], k->new ArrayList<>()).add(i);
    // 动态规划得到总体顺序数量 x
    return dfs(0)[1];
}
```

1.5.3 1987. Number of Unique Good Subsequences - Hard

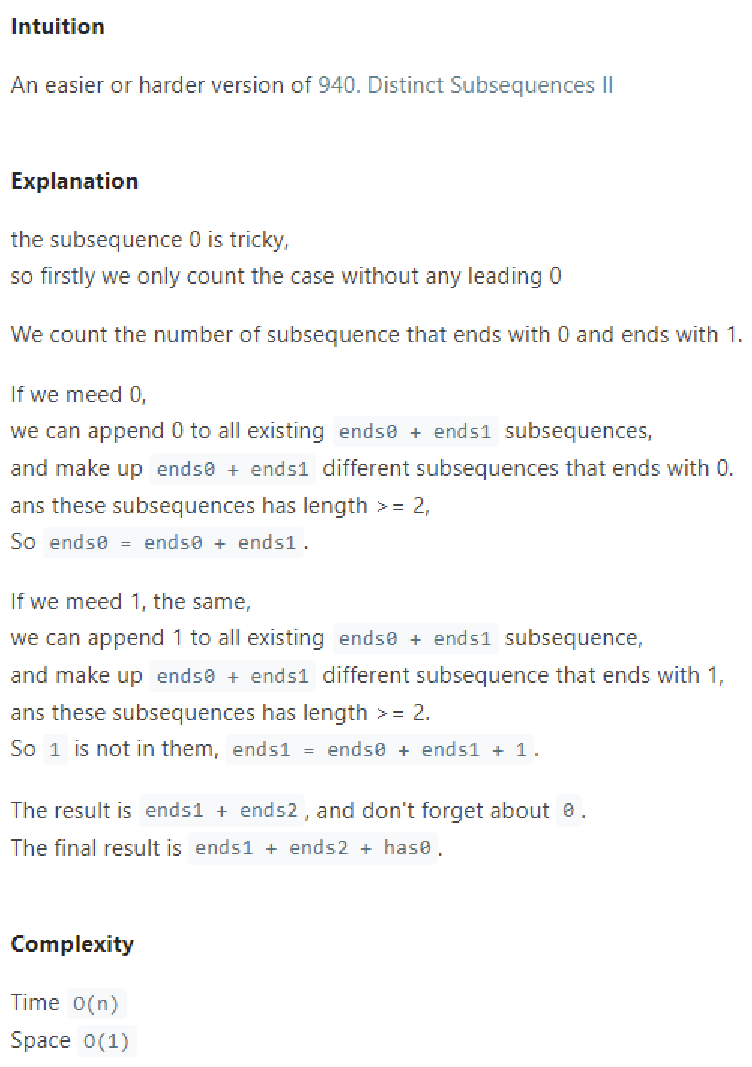
You are given a binary string binary. A subsequence of binary is considered good if it is not empty and has no leading zeros (with the exception of "0").

Find the number of unique good subsequences of binary.

For example, if binary = "001", then all the good subsequences are ["0", "0", "1"], so the unique good subsequences are "0" and "1". Note that subsequences "00", "01", and "001" are not good because they have leading zeros. Return the number of unique good subsequences of binary. Since the answer may be very large, return it modulo 109 + 7.

A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

```
public int numberOfUniqueGoodSubsequences(String binary) {
    int mod = (int)1e9 + 7;
    int n = binary.length(), preZoo = 0, preOne = 0, m = 1;
    long [] dp = new long [n+1];
    String s = "#" + binary;
    while (m <= n && s.charAt(m) == '0') m++;
    if (m == n+1) return 1;
    dp[m] = 1;
    preOne = m;
    preZoo = m-1;
    for (int i = m+1; i <= n; i++) {
        char c = s.charAt(i);
        int j = (c == '0' ? preZoo : preOne);
        dp[i] = (2 * dp[i-1] % mod - (j >= 1 ? dp[j-1] : 0) + mod) % mod;
        if (c == '0') preZoo = i;
        else preOne = i;
    }
    return (int)dp[n] + (s.indexOf("0") != -1 ?  1 : 0);
}
```



Intuition

An easier or harder version of 940. Distinct Subsequences II

Explanation

the subsequence 0 is tricky,
so firstly we only count the case without any leading 0

We count the number of subsequence that ends with 0 and ends with 1.

If we meed 0,
we can append 0 to all existing `ends0 + ends1` subsequences,
and make up `ends0 + ends1` different subsequences that ends with 0.
ans these subsequences has length >= 2,
So `ends0 = ends0 + ends1`.

If we meed 1, the same,
we can append 1 to all existing `ends0 + ends1` subsequence,
and make up `ends0 + ends1` different subsequence that ends with 1,
ans these subsequences has length >= 2.
So `1` is not in them, `ends1 = ends0 + ends1 + 1`.

The result is `ends1 + ends2`, and don't forget about `0`.
The final result is `ends1 + ends2 + has0`.

Complexity

Time `O(n)`
Space `O(1)`

```java
public int numberOfUniqueGoodSubsequences(String binary) {
    int mod = (int)1e9 + 7;
    int endZoo = 0, endOne = 0, hasZoo = 0;
    for (int i = 0; i < binary.length(); i++)
        if (binary.charAt(i) == '1')
            endOne = (endOne + endZoo + 1) % mod;
        else {
            endZoo = (endZoo + endOne) % mod;
            hasZoo = 1;
        }
    return (endOne + endZoo + hasZoo) % mod;
}
```

- 还有一个没有看懂的
  - https://leetcode-cn.com/problems/number-of-unique-good-subsequences/solution/ju-yi-fan-san-by-aver
  - https://leetcode-cn.com/problems/distinct-subsequences-ii/solution/dong-tai-gui-hua-cong-fen-xi-da

```python
def numberOfUniqueGoodSubsequences(self, binary: str) -> int:
        M = 10**9+7
        dp = [0]*10
        b = str(int(binary))
        l = len(binary) - len(b)
        if l > 0:
            dp[0] = 1
        for c in b:
            if dp[0] >= 1:
                dp[int(c)] = (sum(dp)) % M
            else:
                dp[int(c)] = ( 1+ sum(dp)) % M
        return sum(dp)%M
```

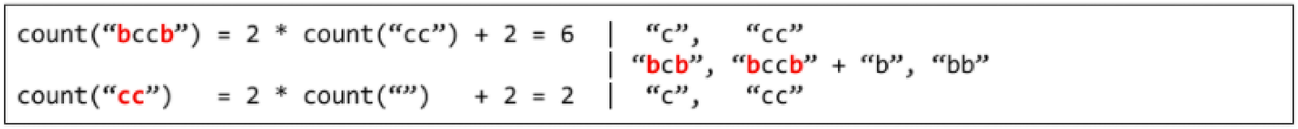
1.5.4 730. Count Different Palindromic Subsequences - Hard

Given a string s, return the number of different non-empty palindromic subsequences in s. Since the answer may be very large, return it modulo 109 + 7.

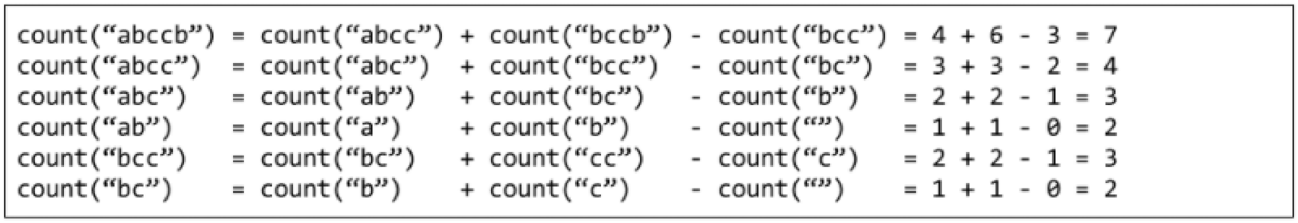
A subsequence of a string is obtained by deleting zero or more characters from the string.

A sequence is palindromic if it is equal to the sequence reversed.

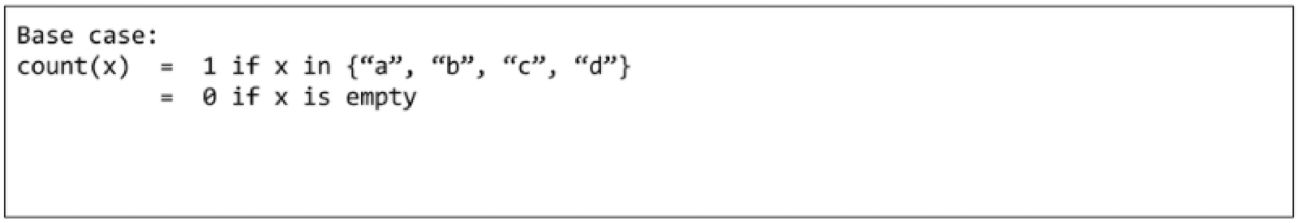
Two sequences a1, a2, . . . and b1, b2, . . . are different if there is some i for which ai != bi.



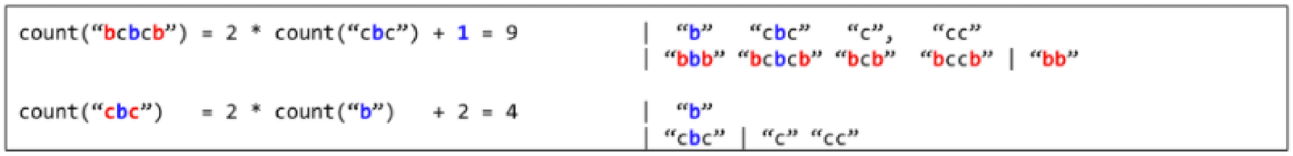
```
count("bccb") = 2 * count("cc") + 2 = 6  |  "c",   "cc"
                                         | "bcb", "bccb" + "b", "bb"
count("cc")   = 2 * count("")   + 2 = 2  |  "c",   "cc"
```



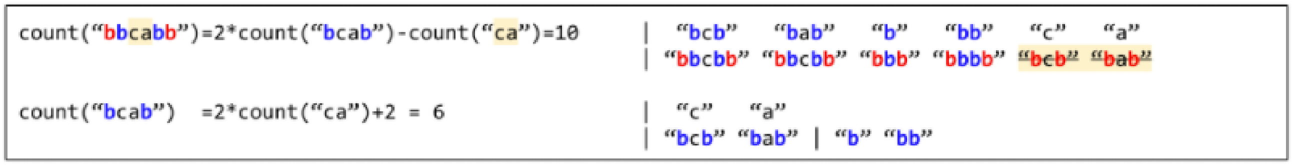
```
count("abccb") = count("abcc") + count("bccb") - count("bcc") = 4 + 6 - 3 = 7
count("abcc")  = count("abc")  + count("bcc")  - count("bc")  = 3 + 3 - 2 = 4
count("abc")   = count("ab")   + count("bc")   - count("b")   = 2 + 2 - 1 = 3
count("ab")    = count("a")    + count("b")    - count("")    = 1 + 1 - 0 = 2
count("bcc")   = count("bc")   + count("cc")   - count("c")   = 2 + 2 - 1 = 3
count("bc")    = count("b")    + count("c")    - count("")    = 1 + 1 - 0 = 2
```



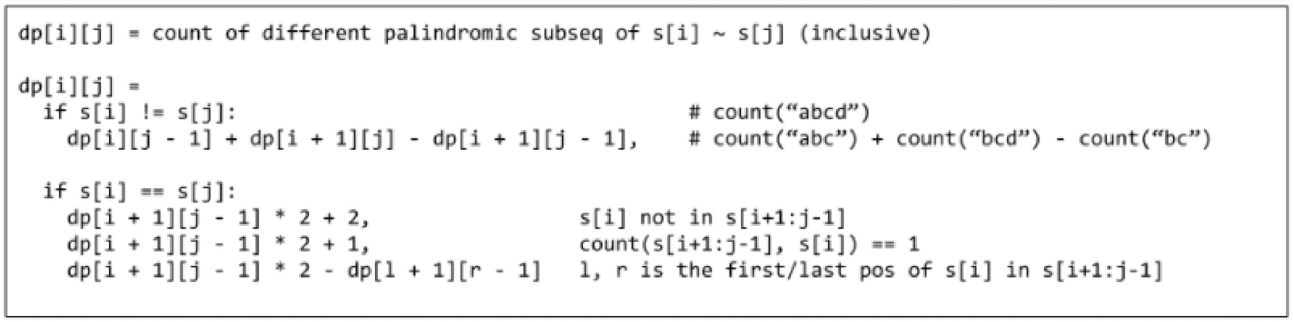
```
Base case:
count(x)  =  1 if x in {"a", "b", "c", "d"}
          =  0 if x is empty
```



```
count("bcbcb") = 2 * count("cbc") + 1 = 9        |  "b"   "cbc"   "c",   "cc"
                                                 | "bbb" "bcbcb" "bcb"  "bccb" | "bb"

count("cbc")   = 2 * count("b")   + 2 = 4        |  "b"
                                                 | "cbc" | "c" "cc"
```



```
count("bbcabb")=2*count("bcab")-count("ca")=10   |  "bcb"   "bab"   "b"   "bb"   "c"   "a"
                                                 | "bbcbb" "bbcbb" "bbb" "bbbb" "bcb" "bab"

count("bcab")  =2*count("ca")+2 = 6              |  "c"   "a"
                                                 | "bcb" "bab" | "b" "bb"
```



```
dp[i][j] = count of different palindromic subseq of s[i] ~ s[j] (inclusive)

dp[i][j] =
  if s[i] != s[j]:                                # count("abcd")
    dp[i][j - 1] + dp[i + 1][j] - dp[i + 1][j - 1],  # count("abc") + count("bcd") - count("bc")

  if s[i] == s[j]:
    dp[i + 1][j - 1] * 2 + 2,                 s[i] not in s[i+1:j-1]
    dp[i + 1][j - 1] * 2 + 1,                 count(s[i+1:j-1], s[i]) == 1
    dp[i + 1][j - 1] * 2 - dp[l + 1][r - 1]   l, r is the first/last pos of s[i] in s[i+1:j-1]
```

```java
private int dfs(char[] s, int i, int j) {
    if (i > j) return 0;
    if (i == j) return 1;
    if (dp[i][j] > 0) return dp[i][j];
    long ans = 0;
    if (s[i] == s[j]) {
        ans += dfs(s, i + 1, j - 1) * 2;
        int l = i + 1;
        int r = j - 1;
        while (l <= r && s[l] != s[i]) ++l;
        while (l <= r && s[r] != s[i]) --r;
        if (l > r) ans += 2;
        else if (l == r) ans += 1;
        else ans -= dfs(s, l + 1, r - 1);
    } else
        ans = dfs(s, i, j - 1) + dfs(s, i + 1, j) - dfs(s, i + 1, j - 1);
    return dp[i][j] = (int)((ans + mod) % mod);
}
private static final int mod = (int)1e9 + 7;
private int [][] dp;
public int countPalindromicSubsequences(String S) {
    int n = S.length();
    dp = new int[n][n];
    return dfs(S.toCharArray(), 0, n - 1);
}
```

- dp

```java
public int countPalindromicSubsequences(String s) {
    int n = s.length();
    int mod = (int)1e9 + 7;
    char [] arr = s.toCharArray();
    long [][] dp = new long [n][n];
    for (int i = 0; i < n; i++)
        dp[i][i] = 1;
    for (int len = 1; len <= n; len++) {
        for (int i = 0; i+len < n; i++) {
            int j = i + len;
            if (arr[i] == arr[j]) {
                dp[i][j] = dp[i+1][j-1] * 2;
                int l = i+1;
                int r = j-1;
                while (l <= r && arr[l] != arr[i]) ++l;
                while (l <= r && arr[r] != arr[i]) --r;
                if (l == r) dp[i][j] += 1;
                else if (l > r) dp[i][j] += 2;
                else dp[i][j] -= dp[l+1][r-1];
            } else dp[i][j] = dp[i][j-1] + dp[i+1][j] - dp[i+1][j-1];
            dp[i][j] = (dp[i][j] + mod) % mod;
        }
    }
    return (int)dp[0][n-1];
}
```

1.5.5 1125. Smallest Sufficient Team - Hard 这个题要多写几遍

In a project, you have a list of required skills req_{skills}, and a list of people. The ith person people[i] contains a list of skills that the person has.

Consider a sufficient team: a set of people such that for every required skill in req_{skills}, there is at least one person in the team who has that skill. We can represent these teams by the index of each person.

For example, team = [0, 1, 3] represents the people with skills people[1], people[2], and people[3]. Return any sufficient team of the smallest possible size, represented by the index of each person. You may return the answer in any order.

It is guaranteed an answer exists.

```java
// 强行剪枝: 收集到的 size >= 目前的结果, 直接 return;
// 这题的思路就是先把 skill 和 set of people 建立好,
// 然后去用 skill set 做 backtracking 收集, 如果 temp team 的 size 大于结果, 直接 return, 否则 update 结果,
// 这里有个小 tricky 的地方, 就是如果 people 是新人, 加入之后 dfs, backtracking 的时候, 要判断如果是新人, 则 remove, 否则不 remove;
private void dfs(String[] req_skills, HashSet<Integer> team, int idx) {
    if (team.size() >= minTeamSize) return; // 强行剪枝: 收集到的 size >= 目前的结果, 直接 return;
    if (idx == req_skills.length) {
        minTeamSize = team.size();
        resTeam = new HashSet<Integer>(team);
        return;
    }
    boolean isNewPerson = false;
    for (int people : map.get(req_skills[idx])) {
        isNewPerson = team.add(people);
        dfs(req_skills, team, idx + 1);
        if (isNewPerson)
            team.remove(people);
    }
}
HashMap<String, Set<Integer>> map;
Set<Integer> resTeam;
int minTeamSize;
public int[] smallestSufficientTeam(String[] req_skills, List<List<String>> people) {
    minTeamSize = people.size();
    this.map = new HashMap<>();
    for (int i = 0; i < minTeamSize; i++)
        for (String skill: people.get(i))
            map.computeIfAbsent(skill, k -> new HashSet<Integer>()).add(i);
```

[2] DEFINITION NOT FOUND.

[3] DEFINITION NOT FOUND.

```
    this.resTeam = new HashSet<Integer>();
    dfs(req_skills, new HashSet<Integer>(), 0);
    int [] res = new int[resTeam.size()];
    int idx = 0;
    for (int person : resTeam)
        res[idx++] = person;
    return res;
}
```

- Java soution using Bit DP 10ms

```
public int[] smallestSufficientTeam(String[] req_skills, List<List<String>> people) {
    int n = req_skills.length, range = 1 << n, cur, idx;
    Map<String, Integer> idxMap = new HashMap<>();
    for (int i = 0; i < n; i++)
        idxMap.put(req_skills[i], i);
    long [] dp = new long [range]; // 每个 bit 位实际存了构成答案最小组的各成员的下标, 60 个人, long
    int [] cnt = new int [range];
    Arrays.fill(cnt, Integer.MAX_VALUE);
    cnt[0] = 0;
    for (int i = 0; i < people.size(); i++) {
        List<String> l = people.get(i);
        cur = 0;
        for (String skill : l)
            if (idxMap.containsKey(skill))
                cur |= 1 << idxMap.get(skill);
        for (int j = range-1; j > 0; j--) {
            idx = (j & cur) ^ j; // 由其它人所构成的拥有 j 的这些种技能的子集/ j 的这些种技能可以由 j 一个人来替换 (其它可能需要很多人才能最终拥有这些技能)
            if (cnt[idx] != Integer.MAX_VALUE && cnt[j] > cnt[idx] + 1) {
                cnt[j] = cnt[idx] + 1;
                dp[j] = dp[idx] | (1L << i); // at most 60 people
            }
        }
    }
    int [] res = new int[cnt[range-1]];
    long preRes = dp[range-1]; // 5 people: 11111, 1111, 111, 11, 1
    int valIdx = 0;
    long val = 0;
    idx = 0;
    while (preRes != 0) {
        val = preRes & 1;
        if (val == 1) res[idx++] = valIdx;
        preRes >>= 1;
        valIdx++;
    }
    return res;
}
```

- DFS + Memorizaion (A real O(2^{skill} * people) Solution) Java 8ms
  - https://leetcode.com/problems/smallest-sufficient-team/discuss/1011135/DFS-%2B-Memorizaion-(A-real-O(2skill-*-people)-Solution)-Java-8ms

```
List<Integer> minComb;
int[] peopleSkillMasks;
Integer[] memo;  // 这个方法确实快一点儿
int[] nextPerson;
int n;
public int[] smallestSufficientTeam(String[] req_skills, List<List<String>> people) {
    // 1. some preprocess to get bitmask for people skills
    this.n = req_skills.length;
    Map<String, Integer> skillToIdx = new HashMap<>();
    for (int i = 0; i < n; i++)
        skillToIdx.put(req_skills[i], i);
    this.peopleSkillMasks = new int[people.size()];
    for (int i = 0; i < peopleSkillMasks.length; i++) {
        int skillMask = 0;
        for (String skill : people.get(i))
            skillMask |= (1 << skillToIdx.get(skill));
        peopleSkillMasks[i] = skillMask;
    }
    // 2. dfs
    memo = new Integer[1 << n];
    nextPerson = new int[1 << n];
    dfs(0, 0);
    // 3. reconstruct the path
    int curSkillSet = 0;
    List<Integer> res = new ArrayList<>();
    while(curSkillSet != (1 << n) - 1) {
        res.add(nextPerson[curSkillSet]);
        curSkillSet |= peopleSkillMasks[nextPerson[curSkillSet]];
    }
    return res.stream().mapToInt(i->i).toArray();
}
// a very simple dfs with memo to compute all combinations of people.
// Use memorization to optimize the time complexity to O(2^skill * people) 2^skill for 2^skill node in the tree, people because each node has people comp
private int dfs(int curSkillSet, int startIdx) {
    if (curSkillSet == (1 << n) - 1) return 0;
    if (memo[curSkillSet] == null) {
        int res = Integer.MAX_VALUE / 2;
        int nextPersonIdx = -1;
        for (int i = startIdx; i < peopleSkillMasks.length; i++) {
            int withNewSkill = peopleSkillMasks[i] | curSkillSet;
            if (withNewSkill != curSkillSet) {
                int numPeople = dfs(withNewSkill, i+1) + 1;
                if (res > numPeople) {
                    res = numPeople;
                    nextPersonIdx = i;
                }
            }
        }
        memo[curSkillSet] = res;
        nextPerson[curSkillSet] = nextPersonIdx;
    }
    return memo[curSkillSet];
}
```

- Recursion + Memoization + bit mask , with Simple JAVA solution
  - https://leetcode.com/problems/smallest-sufficient-team/discuss/1487180/Recursion-%2B-Memoization-%2B-bit-mask-with-Simple-JAVA-solution

上面的这些方法相对较偏，就暂时顾不上了

1.5.6 1575. Count All Possible Routes - Hard

You are given an array of distinct positive integers locations where locations[i] represents the position of city i. You are also given integers start, finish and fuel representing the starting city, ending city, and the initial amount of fuel you have, respectively.

At each step, if you are at city i, you can pick any city j such that j != i and 0 <= j < locations.length and move to city j. Moving from city i to city j reduces the amount of fuel you have by |locations[i] - locations[j]|. Please notice that |x| denotes the absolute value of x.

Notice that fuel cannot become negative at any point in time, and that you are allowed to visit any city more than once (including start and finish).

Return the count of all possible routes from start to finish.

Since the answer may be too large, return it modulo $10^9 + 7$.

```
// 自顶向下 （记忆化搜索）
// 每个 dfs 搜索当前状态为城市 i，油量 f 到达终点的方案数。这样决策的时候就很直观：当前这个状态的方案数，由可去的城市的，且油量为剩余油量的到达终点方案数加起来。
// 初始化：每个状态都初始化为-1。
// 当走到终点时，这个状态的可走到终点的方案数 +1。
private int dfs(int [] arr, int end, int idx, int fu) {
    if (dp[idx][fu] != -1) return dp[idx][fu];
    dp[idx][fu] = 0;
    if (idx == end) {
        dp[idx][fu] += 1;
        dp[idx][fu] %= mod;
    }
    for (int i = 0; i < n; i++) {
        if (i == idx || Math.abs(arr[i] - arr[idx]) > fu) continue;
        dp[idx][fu] = (dp[idx][fu] + dfs(arr, end, i, fu-Math.abs(arr[i]-arr[idx]))) % mod;
    }
    return dp[idx][fu];
}
int mod = (int)1e9 + 7;
int [][] dp;
int n;
public int countRoutes(int[] locations, int start, int finish, int fuel) {
    n = locations.length;
    if (fuel < Math.abs(locations[start] - locations[finish])) return 0;
    dp = new int[n][fuel+1];
    for (int i = 0; i < n; i++)
        Arrays.fill(dp[i], -1);
    dfs(locations, finish, start, fuel);
    return dp[start][fuel];
}
// 自底向上
// 为什么想到动态规划：最优子结构：到达终点的方案数肯定由到达其他点的，不同油量的方案数求和。
//     如何定义状态：城市肯定在状态里，因为其他城市有不同的剩余油量的状态，且油量为 0 无法到达，也成为限制之一。所以油量也必须在状态里：
//     d p ( i , f ) dp(i, f)dp(i,f) 表示到达第 i ii 个城市，剩余油量为 f ff 的方案数。
//     状态转移：第 i ii 个城市，可以由除本身外的城市转移过来，只要剩余的油量不小于所用的油量就够了，最后答案是求总共的个数，所以只要方案数相加就行：
//     dp(i,fdist)=dp(i,fdist)+dp(k,f)(fdist>=0)
//     枚举顺序：每个城市肯定都要枚举一遍，因为还需要从另一个城市转移过来，所以除本身外的城市肯定还要再枚举一遍。
//     关键是油量的枚举，因为油量肯定是慢慢减少的，可以想到是逆序枚举，而且油量要放在最外层枚举。因为如果先枚举城市 i ii，再枚举城市 j jj，再枚举油量的话，只是不断更
// dp：最优子结构 到达终点的方案数肯定由到达其他点的，不同油量的方案数求和
// 搜索：反过来 在第 i 个城市到达 fin 的方案数，也可以由其他的点到达 fin 的方案数转移过来，但是油量有限制，所以油量肯定在状态里
// 所以城市 和 剩余油量肯定在状态里
// dp(i, j) 表示到达第 i 个城市，剩余油量为 j 的方案数
// dp(i, j) = dp(i, j) + dp(k, j - dist)
public int countRoutes(int[] locations, int start, int finish, int fuel) {
    int n = locations.length;
    if (fuel < Math.abs(locations[start] - locations[finish])) return 0;
    int [][] dp = new int[n][fuel+1];
    dp[start][fuel] = 1; // 初始点且燃料满的点方案数为 1
    int leftFu = 0, mod = (int)1e9 + 7;
    for (int j = fuel; j >= 0; j--) { // fuel leftover
        for (int i = 0; i < n; i++) { // cur city
            for (int k = 0; k < n; k++) { // next city
                if (i == k) continue;
                leftFu = j - Math.abs(locations[i] - locations[k]);
                if (leftFu < 0) continue;
                dp[i][leftFu] = (dp[i][leftFu] + dp[k][j]) % mod; // 这里好别扭呀：想呀想呀
            }
        }
    }
    int ans = 0;
    for (int i = 0; i <= fuel; i++)
        ans = (ans + dp[finish][i]) % mod;
    return ans;
}
```

1.5.7 1012. Numbers With Repeated Digits - Hard

Given an integer n, return the number of positive integers in the range [1, n] that have at least one repeated digit.
题意: 统计 1-N 中，满足每个位置都不同的数有几个。
思路：数位 DP。通过一个 1«10 的 mask 表示当前这个数，1-9 哪些数被用了。
比赛的时候，一直想通过一个 dfs 直接找到不重复的数，一直不对。
赛后发现，别人都是通过一个 dfs 找重复的数，然后总个数减去。

```
private int dfs(int len, int limit, int mask) { // 不重复数的个数
    if (len == 0) return 1;
    if (limit == 0 && dp[len][mask][limit] > 0) return dp[len][mask][limit]; // 记忆化部分
    int maxn = limit > 0 ? bit[len] : 9; // 求出最高可以枚举到哪个数字
    int ans = 0;
    for (int i = 0; i <= maxn; i++)  // 当前位
        if ((mask&(1 << i)) == 0)
            if (mask == 0 && i == 0)
                ans += dfs(len - 1, (limit > 0 && i == maxn ? 1 : 0), mask); // 有前导 0，所以 0 不能统计，不更新 mask
            else ans += dfs(len - 1, (limit > 0 && i == maxn ? 1 : 0), mask | (1 << i)); // 更新 mask
    if (limit == 0) dp[len][mask][limit] = ans; // 如果没有限制，代表搜满了，可以记忆化，否则就不能
    return ans;
}
int [][][] dp;
int [] bit;
public int numDupDigitsAtMostN(int N) {
    int sum = N + 1;
    bit = new int [19];
    dp = new int [19][1 << 10][2];
    int idx = 0;
    while (N > 0) {
        bit[++idx] = N % 10;
        N /= 10;
    }
    return sum - dfs(idx, 1, 0);
}
```

- 这道题给了一个正整数 N，让返回所有不大于 N 且至少有一个重复数字的正整数的个数，题目中给的例子也可以很好的帮助我们理解。要求的是正整数的位数上至少要有一个重复数字，当然最简单暴力的方法就是从 1 遍历到 N，然后对于每个数字判断是否有重复数字，看了一眼题目难度 Hard，想都不用想，肯定是超时的。这道题需要更高效的解法，首先来想，若是直接求至少有一个重复数字的正整数，由于并不知道有多少个重复数字，可能 1 个，2 个，甚至全是重复数字，这样很难找到规律。有时候直接求一个问题不好求，可以考虑求其相反的情况，至少有一个重复数字反过来就是一

个重复数字都没有，所以这里可以求不大于 N 且一个重复数字都没有的正整数的个数，然后用 N 减去这个数字即为所求。好，接下来看怎么求，对于任意一个 N，比如 7918，是个四位数，而所有的三位数，两位数，一位数，都一定比其小，所以可以直接求出没有重复数字的三位数，两位数，和一位数。比如三位数，由于百位上不能有 0，则只有 9 种情况，十位上可以有 0，则有 9 种情况，个位上则有 8 种情况，所以就是 9*9*8。可以归纳出没有重复数字的 n 位数的个数，最高位去除 0 还有 9 种，剩余的 n-1 位则依次是 9，8，7... 则后面的 n-1 位其实是个全排列，从 9 个数中取出 n-1 个数字的全排列，初中就学过的。这里写一个全排列的子函数，求从 m 个数字中取 n 个数字的全排列，方便后面计算。算完这些后，还要来算符合题意的四位数，由于第一位是 7，若千位上是小于 7 的数字（共有 6 种，千位上不能是 0），则后面的百位，十位，个位又都可以全排列了，从 9 个数字中取 3 个数字的全排列，再乘以千位上小于 7 的 6 种情况。若当千位固定为 7，则百位上可以放小于 9 的数字（共有 8 种，百位不能放 7，但可以放 0），则后面的十位和个位都可以全排列了，从 8 个数字种取出 2 个数字的全排列，再乘以百位上小于 9 的 8 种情况。需要注意的是，遍历给定数字的各个位时，有可能出现重复数字，一旦出现了之后，则该 prefix 就不能再用了，因为已经不合题意了。所以要用一个 HashSet 来记录访问过的数字，一旦遇到重复数字后就直接 break 掉。最后还有一个小 trick 需要注意，由于 N 本身也需要计算进去，所以再计算的时候，使用 N+1 进行计算的话，就可以把 N 这种情况算进去了

```java
private int A(int m, int n) {
    return n == 0 ? 1 : A(m, n-1) * (m-n+1);
}
public int numDupDigitsAtMostN(int n) {
    List<Integer> digits = new ArrayList<>();
    Set<Integer> vis = new HashSet<>();
    for (int i = n+1; i > 0; i /= 10)
        digits.add(0, i % 10);
    int res = 0, m = digits.size();
    for (int i = 1; i < m; i++) res += 9 * A(9,  i-1);
    for (int i = 0; i < m; i++) {
        for (int j = i > 0 ? 0 : 1; j < digits.get(i); ++j) {
            if (vis.contains(j)) continue;
            res += A(9-i, m-i-1);
        }
        if (vis.contains(digits.get(i))) break;
        vis.add(digits.get(i));
    }
    return n - res;
}
```

1.5.8 514. Freedom Trail - Hard

In the video game Fallout 4, the quest "Road to Freedom" requires players to reach a metal dial called the "Freedom Trail Ring" and use the dial to spell a specific keyword to open the door.

Given a string ring that represents the code engraved on the outer ring and another string key that represents the keyword that needs to be spelled, return the minimum number of steps to spell all the characters in the keyword.

Initially, the first character of the ring is aligned at the "12:00" direction. You should spell all the characters in key one by one by rotating ring clockwise or anticlockwise to make each character of the string key aligned at the "12:00" direction and then by pressing the center button.

At the stage of rotating the ring to spell the key character key[i]:

You can rotate the ring clockwise or anticlockwise by one place, which counts as one step. The final purpose of the rotation is to align one of ring's characters at the "12:00" direction, where this character must equal key[i]. If the character key[i] has been aligned at the "12:00" direction, press the center button to spell, which also counts as one step. After the pressing, you could begin to spell the next character in the key (next stage). Otherwise, you have finished all the spelling.

1. 解题思路分析: 这个图把钥匙中每个字母的出现位置记住了，以后拿去用不搜 dfs + 记忆数组

   - 记录下所有字母对应的位置，这样在找字母相对位置的时候就不需要循环搜索了
   - 采用递归的方法，找出当前字母对应的位置最小的步数：只需要把当前字母对应的所有位置找出来，然后计算最小值即可
   - 下一个位置再次迭代计算即可

```java
public int minLen(int len, int i, int j) {
    int min = Math.min(i, j);
    int max = Math.max(i, j);
    return Math.min(Math.abs(i - j), Math.abs(len + min - max));
}
public int helper(String ring, int i, String key, int j) {
    if (j >= n) return 0;
    if (dp[i][j] > 0) return dp[i][j];
    List<Integer> nextPos = map.get(key.charAt(j));
    int min = Integer.MAX_VALUE;
    for (int k = 0; k < nextPos.size(); k++)
        min = Math.min(min, helper(ring, nextPos.get(k), key, j+1) + minLen(m, nextPos.get(k), i) + 1);
    dp[i][j] = min;
    return dp[i][j];
}
Map<Character, List<Integer>> map = new HashMap<>(); // 这个图把钥匙中每个字母的出现位置记住了，以后拿去用不搜
int[][] dp;
int m, n;
public int findRotateSteps(String ring, String key) {
    m = ring.length();
    n = key.length();
    dp = new int[m][n];
    for (int i = 0; i < m; i++) {
        if (key.indexOf(ring.charAt(i)) == -1) continue;
        char c = ring.charAt(i);
        List<Integer> li = map.get(c);
        if (li == null) {
            li = new ArrayList<>();
            map.put(c, li);
        }
        li.add(i);
    }
    return helper(ring, 0, key, 0);
}
```

2. 解题思路分析动态规划

- 博主最先尝试的用贪婪算法来做，就是每一步都选最短的转法，但是 OJ 中总有些 test case 会引诱贪婪算法得出错误的结果，因为全局最优解不一定都是局部最优解，而贪婪算法一直都是在累加局部最优解，这也是为啥 DP 解法这么叼的原因。贪婪算法好想好实现，但是不一定能得到正确的结果。DP 解法难想不好写，但往往才是正确的解法，这也算一个 trade off 吧。
- 此题需要使用一个二维数组 dp，其中 dp[i][j] 表示转动从 i 位置开始的 key 串所需要的最少步数 (这里不包括 spell 的步数，因为 spell 可以在最后统一加上)，此时表盘的 12 点位置是 ring 中的第 j 个字符。不得不佩服这样的设计的确很巧妙，我们可以从 key 的末尾往前推，这样 $dp^{1,1}$就是我们所需要的结果，因为此时是从 key 的开头开始转动，而且表盘此时的 12 点位置也是 ring 的第一个字符。现在我们来看如何找出递推公式，对于 dp[i][j]，我们知道此时要将 key[i] 转动到 12 点的位置，而此时表盘的 12 点位置是 ring[j]，我们有两种旋转的方式，顺时针和逆时针，我们的目标肯定是要求最小的转动步数，而顺时针和逆时针的转动次数之和刚好为 ring 的长度 n，这样我们求出来一个方向的次数，就可以迅速得到反方向的转动次数。为了将此时表盘上 12 点位置上的 ring[j] 转动到 key[i]，我们要将表盘转动一整圈，当转到 key[i] 的位置时，我们计算出转动步数 diff，然后计算出反向转动步数，并取二者较小值为整个转动步数 step，此时我们更新 dp[i][j]，更新对比值为 step + dp[i+1][k]，这个也不难理解，因为 key 的前一个字符 key[i+1] 的转动情况 suppose 已经计算好了，那么 dp[i+1][k] 就是当时表盘 12 点位置上 ring[k] 的情况的最短步数，step 就是从 ring[k] 转到 ring[j] 的步数，也就是 key[i] 转到 ring[j] 的步数，用语言来描述就是，从 key 的 i 位置开始转动并且此时表盘 12 点位置为 ring[j] 的最小步数 (dp[i][j]) 就等价于将 ring[k] 转动到 12 点位置的步数 (step) 加上从 key 的 i+1 位置开始转动并且 ring[k] 已经在表盘 12 点位置上的最小步数 (dp[i+1][k]) 之和。
- 突然发现这不就是之前那道 Reverse Pairs 中解法一中归纳的顺序重现关系的思路吗，都做了总结，可换个马甲就又不认识了，泪目中。。。

```
public int findRotateSteps(String ring, String key) {
    int m = key.length();
    int n = ring.length();
    int [][] dp = new int[m+1][n];
    int diff = 0, step = 0;
    for (int i = m-1; i >= 0; i--) {
        for (int j = 0; j < n; j++) {
            dp[i][j] = Integer.MAX_VALUE;
            for (int k = 0; k < n; k++) {
                if (ring.charAt(k) == key.charAt(i)) {
                    diff = Math.abs(j - k);
                    step = Math.min(diff, n-diff);
                    dp[i][j] = Math.min(dp[i][j], step + dp[i+1][k]);
                }
            }
        }
    }
    return dp[0][0] + m;
}
```

3. 解题思路分析: dfs + 记忆数组

- 过程就是需要一步一步求 key 里面的每个字符。如果当前位置已经是对应到这个字符，那么直接按按钮就可以
- 如果当前位置不是，那么有两种旋转方式，顺时针或者逆时针, 然后找到第一个字符就是在同一个方向上的最短距离，
- 因为在同一个方向上，即使后面有重复的字符，无论后面的字符在那里，遇到第一个符合条件的字符就按按钮一定是最优解。
- 但是在不同方向上就不一定了，有可能一个方向上当前字符距离更短，但是有可能后面的字符距离会更远，
  - 比如 ring=ABCDEFGBF , key=BG, 如果看第一个字符，那应该是顺时针，只需要转一格就到，逆时针需要转两格，
  - 但是顺时针第一步快了以后，后面到 G 会需要更长的步骤。而逆时针会比较快。
- 所以，基本的逻辑是每一步不能决定当前哪个方向是否是最优解，只有不断递归，把每步的两个方向全部尝试完到 key 结束才可以
- 当然，如果不做任何处理，这样做是要超时的（我开始就写了这样一个版本），一个直观的做法，就是在递归的基础上
  - 加一个记忆表，针对 ring 的位置 index 和 key 的 kindex 做记录，如果已经存在一个解了就可以直接返回结果
- 这个递归 +memorization 的解法，那一定存在一个 bottom up 的动态规划解法，这个后面再学习

```java
private int helper(String s, String t, int i, int j) { // s: ring, t: key, i: idxRing, j: idxKey
    Map<Integer, Integer> locMap = mem.get(i);
    if (locMap != null)
        if (locMap.get(j) != null) return locMap.get(j);
    if (j == n) return 0;
    int step = 0, k = i;
    boolean foundK = false;
    for (; step <= m/2; ++step) {
        k = (i + step + m) % m;
        if (s.charAt(k) == t.charAt(j)) {
            foundK = true;
            break;
        }
    }
    int rstep = 0, x = i;
    boolean foundX = false;
    while (rstep <= m/2) {
        x = (i - rstep + m) % m;
        if (s.charAt(x) == t.charAt(j)) {
            foundX = true;
            break;
        }
        rstep++;
    }
    int min = Integer.MAX_VALUE;
    if (foundK) min = helper(s, t, k, j+1) + step + 1;
    if (foundX) min = Math.min(min, helper(s, t, x, j+1) + rstep + 1);
    if (locMap == null) {
        locMap = new HashMap<>();
        mem.put(i, locMap);
    }
    locMap.put(j, min);
    return min;
}
Map<Integer, Map<Integer, Integer>> mem = new HashMap<>();
int m, n;
public int findRotateSteps(String ring, String key) {
    m = ring.length();
    n = key.length();
    return helper(ring, key, 0, 0);
}
```

1.5.9 847. Shortest Path Visiting All Nodes

You have an undirected, connected graph of n nodes labeled from 0 to n - 1. You are given an array graph where graph[i] is a list of all the nodes connected with node i by an edge. Return the length of the shortest path that visits every node. You may start and stop at any node, you may revisit nodes multiple times, and you may reuse edges.

```java
public int shortestPathLength(int[][] graph) {
    int n = graph.length;
    int tar = 0, res = 0;
    HashSet<String> s = new HashSet<>();
    Queue<Pair<Integer, Integer>> q = new LinkedList<>();
    for (int i = 0; i < n; i++) {
        int mask = (1 << i);
        tar |= mask;
        s.add(Integer.toString(mask) + "-" + Integer.toString(i));
        q.add(new Pair<>(mask, i));
    }
    while (!q.isEmpty()) {
        for (int i = q.size(); i > 0; i--) {
            Pair cur = q.remove();
            if ((int)cur.getKey() == tar) return res;
            for (int next : graph[(int)cur.getValue()]) {
                int path = (int)cur.getKey() | (1 << next);
                String str = Integer.toString(path) + "-" + Integer.toString(next);
                if (s.contains(str)) continue;
                s.add(str);
                q.add(new Pair<>(path, next));
            }
        }
        ++res;
    }
    return -1;
}
```

1.5.10 1931. Painting a Grid With Three Different Colors

You are given two integers m and n. Consider an m x n grid where each cell is initially white. You can paint each cell red, green, or blue. All cells must be painted. Return the number of ways to color the grid with no two adjacent cells having the same color. Since the answer can be very large, return it modulo 109 + 7.

- lightweighted 轻巧点儿的解题方案: bitmask

```
// time O( (2^5) *2 * N)
// SPACE O(N)
//     For m = 5, there are at most 48 valid states for a single column so we can handle it column by column.
//     We encode the color arrangement by bit mask (3 bit for a position) and use dfs to generate the all valid states.
//         Then for each column, we iterator all the states and check if its still valid with the previous column.
public void helper(int m, int pos, HashMap<Integer, Long> dic, int pre, int cur) {
    if (pos == m) {
        dic.put(cur, 1L);
        return;
    }
    //不需要 {1, 2, 4} {0, 1, 2} is ok 每个格（实际占用 3 个 bit）
    for (int i = 0; i < 3; i++) {
        if (i == pre) continue;
        helper(m, pos + 1, dic, i, (cur << 3) | (1 << i)); // 每处理一格，将当前状态左移 3 位？（实际每个格占用 3 个 bit 位）| 现在这个格的值？这个，我好昏呀
    }
}
static int mod = (int) 1e9 + 7;
public int colorTheGrid(int m, int n) {
    HashMap<Integer,Long> dic = new HashMap<>();
    helper(m, 0, dic, -1, 0);     // 这应该就是我想找的精巧不占多少空间的 mask 了，可是有点儿看不懂
    HashSet<Integer> set = new HashSet<>(dic.keySet());
    for (int i = 1; i < n; i++) { // 动态规划：用两个图像滚动数组一样轮流记载得出答案
        HashMap<Integer, Long> tmp = new HashMap<>();
        for (int x: set)
            for (int y : set)
                if ((x & y) == 0) // 相邻涂色方案为有效方案
                    tmp.put(y, (tmp.getOrDefault(y, 0L) + dic.get(x)) % mod);
        dic = tmp;
    }
    long res = 0L;
    for (Long x : dic.values()) {
        res += x;
        res %= mod;
    }
    return (int) res;
}
```

- 比较传统一点儿的解法，思路清晰

```
// 参考的答案里，这个最逻辑简单、通俗大众易懂，但稍显笨重，两个图，用一个链表来记忆一行的涂色方案，如果有更精巧一点儿的 bitmask，是我想找的答案
// https://leetcode.com/problems/painting-a-grid-with-three-different-colors/discuss/1334366/Easy-Java-comments-28ms-O(n*P*P)-complexity-memory-O(P)-wher
// 先预处理得到单行的所有有效涂色方案，
// 再进一步计算得到每种单行方案对应的有效邻行方案
// 在此基础上，结合动态规划方法，逐行求解各种涂色状态对应的方案总数，最后统计得到总方案数。
```

```
public int colorTheGrid(int m, int n) {
// 获得单行所有涂色方案
    Map<Integer, List<Integer>> line = new HashMap<>(); //  3^m ways of paying one row
    int range = (int)Math.pow(3, m); // 用 0、1、2 表示各个网格的颜色, key 为方案对应的数值, value 为方案对应的数组
    for (int i = 0; i < range; i++) {
        List<Integer> list = new ArrayList<>(); //  val val values (0, 1, 2) of every m cols into list
        int val = i;
        for (int j = 0; j < m; j++) {
            list.add(val % 3);
            val /= 3;
        }
        boolean valid = true; // 确认该数组中是否存在相邻位置颜色相同
        for (int j = 1; j < m; j++)
            if (list.get(j-1) == list.get(j)) {
                valid = false;
                break;
            }
        if (valid) line.put(i, list); // 相邻网格颜色均不同, 为有效方案, 加入哈希表
    }
// 预处理得到每种单行方案对应的有效邻行方案
    Map<Integer, List<Integer>> adj = new HashMap<>();
    Iterator it = line.entrySet().iterator();
    while (it.hasNext()) {     //  3^m ways of paying one row
        Map.Entry entry = (Map.Entry)it.next();
        int va = (int)entry.getKey();
        List<Integer> lva = (List<Integer>)entry.getValue();
        adj.put(va, new ArrayList<Integer>());
        Iterator itb = line.entrySet().iterator();
        while (itb.hasNext()) { //  3^m ways of paying one row
            Map.Entry enb = (Map.Entry)itb.next();
            int vb = (int)enb.getKey();
            List<Integer> lvb = (List<Integer>)enb.getValue();
            boolean valid = true;
            for (int i = 0; i < m; i++)
                if (lva.get(i) == lvb.get(i)) {
                    valid = false;
                    break;
                } // among 3^m ways of painting one row, how many is valid, and valid mask into adj.get(va);
            if (valid) adj.get(va).add(vb);
        }
    }
// 动态规划, 逐行求解方案数
    int mod = (int)(1e9+7);
    long [] dp = new long [range];  // 上一行各种涂色方案对应的总方法数
    for (int i = 0; i < range; i++) // 初始化
        dp[i] = line.containsKey(i) ? 1 : 0;
    for (int i = 1; i < n; i++) {   // 从第二行开始动态规划
        long [] cur = new long [range];  // 新一行各种涂色方案对应的总方法数
        for (int j = 0; j < range; j++)
            if (adj.containsKey(j)) {    // 该方案有效
                for (int v : adj.get(j)) // 遍历有效的相邻方案
                    cur[j] = (cur[j] + dp[v]) % mod; // 总方法数累加
            }
        System.arraycopy(cur, 0, dp, 0, range);
    }
    long ans = 0;
    for (int i = 0; i < range; i++)
        ans = (ans + dp[i]) % mod;
    return (int)ans;
}
```

1.5.11 313. Super Ugly Number

A super ugly number is a positive integer whose prime factors are in the array primes. Given an integer n and an array of integers primes, return the nth super ugly number. The nth super ugly number is guaranteed to fit in a 32-bit signed integer.

```
static class Node implements Comparable<Node> {
    private int index;
    private int val;
    private int prime;
    public Node(int index, int val, int prime) {
        this.index = index;
        this.val = val;
        this.prime = prime;
    }
    public int compareTo(Node other) {
        return this.val - other.val;
    }
}
public int nthSuperUglyNumber(int n, int[] primes) {
    final int [] arr = new int[n];
    arr[0] = 1;              // 1 is the first ugly number
    final Queue<Node> q = new PriorityQueue<>();
    for (int i = 0; i < primes.length; ++i)
        q.add(new Node(0, primes[i], primes[i]));
    for (int i = 1; i < n; ++i) {
        Node node = q.peek(); // get the min element and add to arr
        arr[i] = node.val;
        do {             // update top elements
            node = q.poll();
            node.val = arr[++node.index] * node.prime;
            q.add(node); // push it back
        } while (!q.isEmpty() && q.peek().val == arr[i]); // prevent duplicate
    }
    return arr[n - 1];
}
```

- 下面这种解法也很巧妙

```
public int nthSuperUglyNumber(int n, int[] primes) {
    int m = primes.length;
    int [] ans = new int[n]; // 存放 1-n 个 SuperUglyNumber
    ans[0] = 1;              // 第一个 SuperUglyNumber 是 1
    int [] next = new int[m];
    for (int i=0; i < m; i++)
        next[i] = 0;         // 初始化
    int cnt = 1, min = Integer.MAX_VALUE, tmp = 0;
    while (cnt < n) {
        min = Integer.MAX_VALUE;
        for (int i = 0; i < m; i++){
             tmp = ans[next[i]] * primes[i];
             min = Math.min(min, tmp);
        }
        for (int i = 0; i < m; i++)
            if (min == ans[next[i]] * primes[i])
                next[i]++;
        ans[cnt++] = min;^^I^^I^^I
    }
    return ans[n-1];^^I^^I
}
```

1.5.12 1786. Number of Restricted Paths From First to Last Node - Dijkstra 算法

There is an undirected weighted connected graph. You are given a positive integer n which denotes that the graph has n nodes labeled from 1 to n, and an array edges where each edges[i] = [ui, vi, weighti] denotes that there is an edge between nodes ui and vi with weight equal to weighti. A path from node start to node end is a sequence of nodes [z0, z1, z2, . . . , zk] such that z0 = start and zk = end and there is an edge between zi and zi+1 where 0 <= i <= k-1. The distance of a path is the sum of the weights on the edges of the path. Let distanceToLastNode(x) denote the shortest distance of a path between node n and node x. A restricted path is a path that also satisfies that distanceToLastNode(zi) > distanceToLastNode(zi+1) where 0 <= i <= k-1. Return the number of restricted paths from node 1 to node n. Since that number may be too large, return it modulo 109 + 7.

```
public void dijkstra(int n) {
    Queue<int []> q = new PriorityQueue<>((a, b) -> (a[1] - b[1]));
    q.add(new int [] {n, 0});
    Arrays.fill(dist, Integer.MAX_VALUE);
    dist[n] = 0;
    int [] cur = null;
    int u = 0, d = 0;
    while (!q.isEmpty()) {
        cur = q.poll();
        u = cur[0];
        d = cur[1];
        if (dist[u] < d) continue;
        if (m.get(u) != null)
            for (int v : m.get(u).keySet())
                if (dist[v] > dist[u] + m.get(u).get(v)) {
                    dist[v] = dist[u] + m.get(u).get(v);
                    q.offer(new int [] {v, dist[v]});
                }
    }
}
private int dfs(int n, int i) {
    if (i == n) return 1;
    if (dp[i] != -1) return dp[i];
    long res = 0;
    if (m.get(i) != null) {
        for (int v : m.get(i).keySet()) {
            if (dist[i] > dist[v])
                res = (res + dfs(n, v)) % mod;
        }
    }
    return dp[i] = (int)res;
}
HashMap<Integer, Map<Integer, Integer>> m = new HashMap<>();
int mod = (int)(1e9+7);
int [] dist;
int [] dp;
public int countRestrictedPaths(int n, int[][] edges) {
    for (int [] v : edges) {
        m.computeIfAbsent(v[0], k->new HashMap<>()).put(v[1], v[2]);
        m.computeIfAbsent(v[1], k->new HashMap<>()).put(v[0], v[2]);
    }
    dist = new int[n+1];
    dijkstra(n);
    dp = new int [n+1];
    Arrays.fill(dp, -1);
    return dfs(n, 1);
}
```

1.5.13 913. Cat and Mouse

A game on an undirected graph is played by two players, Mouse and Cat, who alternate turns. The graph is given as follows: graph[a] is a list of all nodes b such that ab is an edge of the graph. The mouse starts at node 1 and goes first, the cat starts at node 2 and goes second, and there is a hole at node 0. During each player's turn, they must travel along one edge of the graph that meets where they are. For example, if the Mouse is at node 1, it must travel to any node in graph^2. Additionally, it is not allowed for the Cat to travel to the Hole (node 0.) Then, the game can end in three ways: If ever the Cat occupies the same node as the Mouse, the Cat wins. If ever the Mouse reaches the Hole, the Mouse wins. If ever a position is repeated (i.e., the players are in the same position as a previous turn, and it is the same player's turn to move), the game is a draw. Given a graph, and assuming both players play optimally, return 1 if the mouse wins the game, 2 if the cat wins the game, or 0 if the game is a draw.

```
private int dfs(int [][] arr, int t, int i, int j) { // t: steps, i: mouse, j: cat, mouse goes first
    if (t == 2 * n) return 0;
    if (i == j) return dp[t][i][j] = 2;
    if (i == 0) return dp[t][i][j] = 1;
    if (dp[t][i][j] != -1) return dp[t][i][j];
    int tmp = 0;
    if (t % 2 == 0) { // mouse's turn
        boolean catWin = true;
        for (int k = 0; k < arr[i].length; k++) {
            tmp = dfs(arr, t+1, arr[i][k], j);
            if (tmp == 1) return dp[t][i][j] = 1;
            else if (tmp != 2) catWin = false;
        }
        if (catWin) return dp[t][i][j] = 2;
        else return dp[t][i][j] = 0;
    } else { // cat's turn, can NOT step on node # 0
        boolean mouseWin = true;
        for (int k = 0; k < arr[j].length; k++) {
            if (arr[j][k] == 0) continue;
            tmp = dfs(arr, t+1, i, arr[j][k]);
            if (tmp == 2) return dp[t][i][j] = 2;
            else if (tmp != 1) mouseWin = false;
        }
        if (mouseWin) return dp[t][i][j] = 1;
        else return  dp[t][i][j] = 0;
    }
}
int [][][] dp;
int n;
public int catMouseGame(int[][] graph) {
    n = graph.length;
    dp = new int [2*n][n][n];
    for (int i = 0; i < 2*n; i++)
        for (int j = 0; j < n; j++)
            Arrays.fill(dp[i][j], -1);
    dfs(graph, 0, 1, 2);
    return dp[0][1][2];
}
```

1.5.14 1728. Cat and Mouse II

A game is played by a cat and a mouse named Cat and Mouse. The environment is represented by a grid of size rows x cols, where each element is a wall, floor, player (Cat, Mouse), or food. Players are represented by the characters 'C'(Cat),'M'(Mouse). Floors are represented by the character '.' and can be walked on. Walls are represented by the character '#' and cannot be walked on. Food is represented by the character 'F' and can be walked on. There is only one of each character 'C', 'M', and 'F' in grid. Mouse and Cat play according to the following rules: Mouse moves first, then they take turns to move. During each turn, Cat and Mouse can jump in one of the four directions (left, right, up, down). They cannot jump over the wall nor outside of the grid. catJump, mouseJump are the maximum lengths Cat and Mouse can jump at a time, respectively. Cat and Mouse can jump less than the maximum length. Staying in the same position is allowed. Mouse can jump over Cat. The game can end in 4 ways: If Cat occupies the same position as Mouse, Cat wins. If Cat reaches the food first, Cat wins. If Mouse reaches the food first, Mouse wins. If Mouse cannot get to the food within 1000 turns, Cat wins. Given a rows x cols matrix grid and two integers catJump and mouseJump, return true if Mouse can win the game if both Cat and Mouse play optimally, otherwise return false.

```java
private boolean dfs(String [] arr, int t, int i, int j) {
    if (dp[t][i][j] != null) return dp[t][i][j];
    if (t == m*n*2) return false;
    if (arr[i/n].charAt(i%n) == 'F') return true;
    if (arr[j/n].charAt(j%n) == 'F') return false;
    if (i == j) return false;
    int r = 0, c = 0;
    if (t % 2 == 0) { // mouse's turn 老鼠的：只要它能赢一个状态就是赢了
        for (int [] d : dirs)
            for (int k = 0; k <= mj; k++) {
                r = i / n + d[0] * k;
                c = i % n + d[1] * k;
                if (r >= 0 && r < m && c >= 0 && c < n && arr[r].charAt(c) != '#') {
                    if (dfs(arr, t+1, r*n+c, j))
                        return dp[t][i][j] = true; // Mouse could win
                } else break;
            }
        return dp[t][i][j] = false;
    } else { // cat's turn: 但是当是猎的：需要猫不能赢，老鼠才能赢；但是当猫哪怕是赢了只一局，老鼠也就输了
        for (int [] d : dirs)
            for (int k = 0; k <= cj; k++) {
                r = j / n + d[0] * k;
                c = j % n + d[1] * k;
                if (r >= 0 && r < m && c >= 0 && c < n && arr[r].charAt(c) != '#') {
                    if (!dfs(arr, t+1, i, r*n+c))  // Can cat find a path that mouse looses in it?
                        return dp[t][i][j] = false; // Cat wins = mouse loose
                } else break; // 上面这一点儿很重要
            }
        return dp[t][i][j] = true;
    }
}
int [][] dirs = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
Boolean [][][] dp;
int m, n, cj, mj;
public boolean canMouseWin(String[] grid, int catJump, int mouseJump) {
    m = grid.length;
    n = grid[0].length();
    cj = catJump;
    mj = mouseJump;
    dp = new Boolean [1001][m*n][m*n];
    int x = 0, y = 0;
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            if (grid[i].charAt(j) == 'M')
                x = i * n + j;
            else if (grid[i].charAt(j) == 'C')
                y = i * n + j;
    return dfs(grid, 0, x, y);
}
```

1.5.15 810. Chalkboard XOR Game - Hard

You are given an array of integers nums represents the numbers written on a chalkboard.

Alice and Bob take turns erasing exactly one number from the chalkboard, with Alice starting first. If erasing a number causes the bitwise XOR of all the elements of the chalkboard to become 0, then that player loses. The bitwise XOR of one element is that element itself, and the bitwise XOR of no elements is 0.

Also, if any player starts their turn with the bitwise XOR of all the elements of the chalkboard equal to 0, then that player wins.

Return true if and only if Alice wins the game, assuming both players play optimally. There are three cases to consider:

```
Case 1- At the beginning of the game, XOR of all the elements are 0, then Alice wins before the game starts.

Case 2 - XOR!=0 and nums.length is even:
Let’s try to use proof by contradiction. S=(x1^x2⋯^xn)
Assume s!=0, let’s try to find contradiction
XOR s to both sides
s^s=s^(x1^x2⋯^xn)
s^s=0 => 0= s^(x1^x2⋯^xn)
0=(s^x1)^(s^x2)⋯^(s^xn)
Now let’s factor s from each bracket
0=(s^s⋯^s)^(x1^x2⋯^xn)
Since the number of x1..xn is even, the number of s in the left bracket is even, each number ^ itself even times results to 0.
0=0^(x1^x2⋯^xn)
0^ any number is itself so
0=(x1^x2⋯^xn)=s => 0=s
You see that there is a contradiction (compare with initial assumption s!=0), at the beginning we assumed s!=0
Then our assumption is wrong. So, s==0 then Alice wins

Case 3- XOR!=0 and nums.length is odd:
Let’s try to use proof by contradiction here like the other case
Assume s!=0, let’s try to find contradiction
XOR s to both sides
s^s=s^(x1^x2⋯^xn)
s^s=0 => 0= s^(x1^x2⋯^xn)
0=(s^x1)^(s^x2)⋯^(s^xn)
Now let’s factor s from each bracket
0=(s^s⋯^s)^(x1^x2⋯^xn)
Since the number of x1..xn is odd, the number of s in the left bracket is odd, each number ^ itself odd times results to itself.
0=s^(x1^x2⋯^xn) => 0=s^s
Any number XOR itself becomes zero
0=s^s=0
You see here we couldn’t find the contradiction
```

```java
public boolean xorGame(int[] nums) {
    int xor = 0 ;
    for (int i : nums)
        xor = xor ^ i ;
    if (xor == 0 || (nums.length & 1) == 0)
        return true ;
    return false ;
}
```

- 硬掰出来的：注意同猫老鼠游戏 2 一样，要回的是某一方赢与否，与 1 有点儿区别.

```java
private boolean helper(int [] arr, int i, int xor) { // xor: the current leftover array xor result
    if (i == n) return (i % 2 == 0);
    if (dp[i] != null) return dp[i];
    if (xor == 0) return (i % 2 == 0); // to be noted
    int tmp = 0;
    if (i % 2 == 0) { // alice's turn
        for (int j = 0; j < n; j++) {
            if (arr[j] == -1) continue;
            if ((arr[j] ^ xor) == 0) continue;
            tmp = arr[j];
            arr[j] = -1;
            if (helper(arr, i+1, xor^tmp)) return dp[i] = true;
            arr[j] = tmp;
        }
        return dp[i] = false;
    } else { // bob's turn
        for (int j = 0; j < n; j++) {
            if (arr[j] == -1) continue;
            if ((arr[j] ^ xor) == 0) continue;
            tmp = arr[j];
            arr[j] = -1;
            if (!helper(arr, i+1, xor^tmp)) return dp[i] = false;
            arr[j]= tmp;
        }
        return dp[i] = true;
    }
}
Boolean [] dp; // alice win states
int n;
public boolean xorGame(int[] arr) {
    n = arr.length;
    dp = new Boolean [n];
    int [] xor = new int [n];
    for (int i = 0; i < n; i++)
        xor[i] = (i == 0 ? 0 : xor[i-1]) ^ arr[i];
    return helper(arr, 0, xor[n-1]); // i: turn
}
```

1.5.16 backpack III

```
public int backPackIII(int[] A, int[] V, int m) {
    int n = A.length;
    int [] dp = new int[m+1];
    for (int i = 1; i <= m; i++) {
        for (int j = 0; j < n; j++) {
            if (i - A[j] >= 0)
                dp[i] = Math.max(dp[i], dp[i-A[j]] + V[j]);
        }
    }
    return dp[m];
}
```

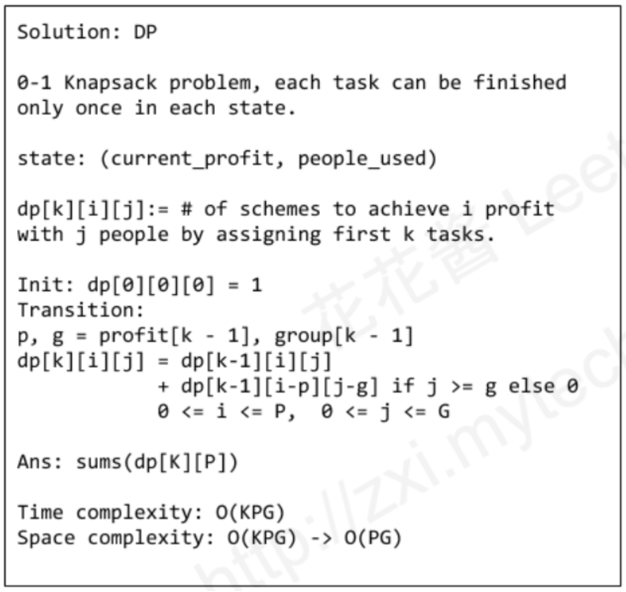
1.5.17 879. Profitable Schemes - Hard 0-1 背包问题

There is a group of n members, and a list of various crimes they could commit. The ith crime generates a profit[i] and requires group[i] members to participate in it. If a member participates in one crime, that member can't participate in another crime.

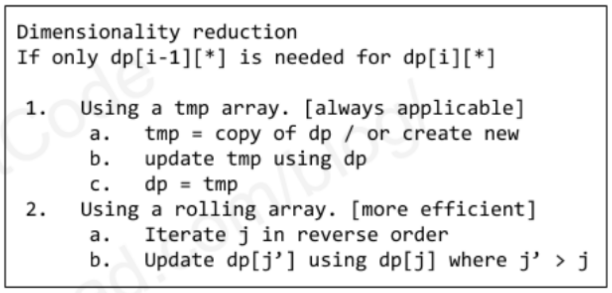
Let's call a profitable scheme any subset of these crimes that generates at least minProfit profit, and the total number of members participating in that subset of crimes is at most n.

Return the number of schemes that can be chosen. Since the answer may be very large, return it modulo 109 + 7.

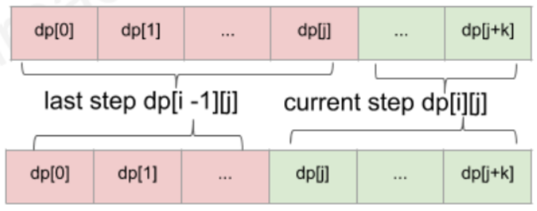
1. 解题思路与分析



```
Solution: DP

0-1 Knapsack problem, each task can be finished
only once in each state.

state: (current_profit, people_used)

dp[k][i][j]:= # of schemes to achieve i profit
with j people by assigning first k tasks.

Init: dp[0][0][0] = 1
Transition:
p, g = profit[k - 1], group[k - 1]
dp[k][i][j] = dp[k-1][i][j]
            + dp[k-1][i-p][j-g] if j >= g else 0
            0 <= i <= P,  0 <= j <= G

Ans: sums(dp[K][P])

Time complexity: O(KPG)
Space complexity: O(KPG) -> O(PG)
```



```
Dimensionality reduction
If only dp[i-1][*] is needed for dp[i][*]

1.  Using a tmp array. [always applicable]
    a.  tmp = copy of dp / or create new
    b.  update tmp using dp
    c.  dp = tmp
2.  Using a rolling array. [more efficient]
    a.  Iterate j in reverse order
    b.  Update dp[j'] using dp[j] where j' > j
```



- 题目中说了结果可能非常大,要对一个超大数取余,看到这里,我们也就该明白为了不爆栈,只能用动态规划 Dynamic Programming 来做，LeetCode 里有好多题都是要对这个 1e9+7 取余，不知道为啥都是对这个数取余。Anyway，who cares，还是来想想 dp 数组如何定义以及怎么推导状态转移方程吧。

首先来看分配黑帮资源时候都需要考虑哪些因素，总共有三点，要干几票买卖，要用多少人，能挣多少钱。所以我们需要一个三维的 dp 数组，其中 dp[k][i][j] 表示最多干 k 票买卖，总共用了 i 个人，获得利润为 j 的情况下分配方案的总数，初始化 $dp^{1,1,1}$ 为 1。

现在来推导状态转移方程，整个规划的核心是买卖，总共买卖的个数是固定的，每多干一票买卖，可能的分配方法就可能增加，但不可能减少的，因为假如当前已经算出来做 k-1 次买卖的分配方法总数，再做一次买卖，之前的分配方法不会减少，顶多是人数不够，做不成当前这票买卖而已，所以我们的 dp[k][i][j] 可以先更新为 dp[k-1][i][j]，然后再来看这第 k 个买卖还能不能做，我们知道假设这第 k 个买卖需要 g 个人，能获得利润 p，只有当我们现在的人数 i 大于等于 g

的时候，才有可能做这个任务，我们要用 g 个人来做任务 k 的话，那么其余的 k-1 个任务只能由 i-g 个人来做了，而且由于整个需要产生利润 j，第 k 个任务能产生利润 p，所以其余的 k-1 个任务需要产生利润 j-p，由于利润不能是负值，所以我们还需要跟 0 比较，取二者的最大值

综上所述，若我们选择做任务 k，则能新产生的分配方案的个数为 dp[k-1][i-g][max(0,j-p)]，记得每次累加完要对超大数取余。最终我们需要将 dp[n][i][P] (0 <= i <= G) 累加起来，因为我们不一定要全部使用 G 个人，只要能产生 P 的利润，用几个人都没关系，而 k 是表示最多干的买卖数，可能上并没有干到这么多，所以只需要累加人数这个维度即可，

```java
public int profitableSchemes(int n, int minProfit, int[] group, int[] profit) { // 0-1 背包问题：每场罪恶在每个状态里最多只能存在一次
    int mod = (int)1e9 + 7, ans = 0;
    int m = group.length;
    int [][][] dp = new int [m+1][n+1][minProfit + 1];
    dp[0][0][0] = 1;
    for (int k = 1; k <= m; k++) {
        int p = profit[k-1], g = group[k-1];
        for (int i = 0; i <= n; i++)
            for (int j = 0; j <= minProfit; j++) {
                dp[k][i][j] = dp[k-1][i][j];
                if (i >= g)
                    dp[k][i][j] = (dp[k][i][j] + dp[k-1][i-g][Math.max(0, j-p)]) % mod;
            }
    }
    for (int i = 0; i <= n; i++)
        ans = (ans + dp[m][i][minProfit]) % mod;
    return ans;
}
```

2. 优化一下空间复杂度: 二维 dp Dimension reduction by using rolling array.

   因为当前做的第 k 个任务，只跟前 k-1 个任务的分配方案有关，所以并不需要保存所有的任务个数的分配方式。这样我们就节省了一个维度，但是需要注意的是，更新的时候 i 和 j 只能从大到小更新，这个其实也不难理解，因为此时 dp[i][j] 存的是前 k-1 个任务的分配方式，所以更新第 k 个任务的时候，一定要从后面开始覆盖，因为用到了前面的值，若从前面的值开始更新的话，就不能保证用到的都是前 k-1 个任务的分配方式，有可能用到的是已经更新过的值，就会出错.

```java
public int profitableSchemes(int n, int minProfit, int[] group, int[] profit) { // 0-1 背包问题：每场罪恶在每个状态里最多只能存在一次
    int mod = (int)1e9 + 7, ans = 0;
    int m = group.length;
    int [][] dp = new int [n+1][minProfit + 1];
    dp[0][0] = 1;
    for (int k = 1; k <= m; k++) {
        int p = profit[k-1], g = group[k-1];
        for (int i = n; i >= g; i--)  // i >=
            for (int j = minProfit; j >= 0; j--)
                dp[i][j] = (dp[i][j] + dp[i-g][Math.max(0, j-p)]) % mod; // 保证了这一行覆盖原数组的正确性
    }
    for (int i = 0; i <= n; i++)
        ans = (ans + dp[i][minProfit]) % mod;
    return ans;
}
```

3. 递归 + 记忆数组来做 todo: 改天补上

   基本思想跟解法一没有太大的区别，递归的记忆数组其实跟迭代形式的 dp 数组没有太大的区别，作用都是保存中间状态从而减少大量的重复计算。这里稍稍需要注意下的就是递归函数中的 corner case，当 k=0 时，则根据 j 的值来返回 0 或 1，当 j 小于等于 0，返回 1，否则返回 0，相当于修改了初始化值（之前都初始化为了整型最小值），然后当 j 小于 0 时，则 j 赋值为 0，因为利润不能为负值。然后就看若当前的 memo[k][i][j] 已经计算过了，则直接返回即可，

   - https://www.cnblogs.com/grandyang/p/11108205.html

1.5.18 377. Combination Sum IV 没能认出这个题目是考 DP

Given an array of distinct integers nums and a target integer target, return the number of possible combinations that add up to target. The answer is guaranteed to fit in a 32-bit integer.

```java
public int combinationSum4(int[] nums, int target) {
    int n = nums.length;
    int [] dp = new int [target +1 ];
    dp [0] = 1;
    for (int i = 1; i <= target; i++) {
        for (int j = 0; j < n; j++) {
            if (i - nums[j] >= 0)
                dp[i] += dp[i-nums[j]];
        }
    }
    return dp[target];
}
```

1.5.19 1049. Last Stone Weight II

You are given an array of integers stones where stones[i] is the weight of the ith stone. We are playing a game with the stones. On each turn, we choose any two stones and smash them together. Suppose the stones have weights x and y with x <= y. The result of this smash is: If x = `y, both stones are destroyed, and If x !` y, the stone of weight x is destroyed, and the stone of weight y has new weight y - x. At the end of the game, there is at most one stone left. Return the smallest possible weight of the left stone. If there are no stones left, return 0.

```
public int lastStoneWeightII(int[] stones) {
    int n = stones.length;
    int sum = Arrays.stream(stones).sum();
    boolean[] dp = new boolean[sum+1];
    dp[0] = true;
    sum = 0;
    for (int v : stones) {
        sum += v;
        for (int i = sum; i >= v; i--)
            if (dp[i-v]) dp[i] = true;
    }
    for (int i = sum/2; i >= 0; i--)
        if (dp[i]) return sum - i * 2;
    return 0;
}
```

1.5.20 1449. Form Largest Integer With Digits That Add up to Target

Given an array of integers cost and an integer target. Return the maximum integer you can paint under the following rules: The cost of painting a digit (i+1) is given by cost[i] (0 indexed). The total cost used must be equal to target. Integer does not have digits 0. Since the answer may be too large, return it as string. If there is no way to paint any integer given the condition, return "0".

```
public String largestNumber(int[] cost, int target) {
    int n = cost.length;
    int [] dp = new int [target+1];
    Arrays.fill(dp, -1);
    dp[0] = 0;
    for (int i = 0; i < n; i++) {
        for (int j = cost[i]; j <= target; j++) {
            if (dp[j-cost[i]] >= 0)
                dp[j] = Math.max(dp[j], dp[j-cost[i]]+1);
        }
    }
    if (dp[target] < 0) return "0";
    char [] ans = new char[dp[target]]; // 采樱桃机器人数组路线那天可以想出来，今天这个路径居然没有想出来!
    int left = target;
    for (int i = 0; i < dp[target]; i++) {
        for (int j = n; j > 0; j--) {
            if (left >= cost[j-1] && dp[left] == dp[left-cost[j-1]] + 1) {
                ans[i] = (char)('0' + j);
                left -= cost[j-1];
                break;
            }
        }
    }
    return String.valueOf(ans);
}
```

1.5.21 516. Longest Palindromic Subsequence

Given a string s, find the longest palindromic subsequence's length in s. A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

```
 public int longestPalindromeSubseq(String s) {
    int n = s.length();
    int [][] dp = new int [n][n];
    dp[n-1][n-1] = 1;
    for (int i = n-2; i >= 0; i--) {
        dp[i][i] = 1;
        for (int j = i+1; j < n; j++) {
            if (s.charAt(i) == s.charAt(j))
                dp[i][j] = 2 + dp[i+1][j-1];
            else dp[i][j] = Math.max(dp[i+1][j], dp[i][j-1]);
        }
    }
    return dp[0][n-1];
}
```

1.5.22 1143. Longest Common Subsequence

Given two strings text1 and text2, return the length of their longest common subsequence. If there is no common subsequence, return 0. 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. For example, "ace" is a subsequence of "abcde". A common subsequence of two strings is a subsequence that is common to both strings.

```
public int longestCommonSubsequence(String S, String T) {
    int m = S.length();
    int n = T.length();
    int [][] dp = new int [m+1][n+1];
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            if (S.charAt(i-1) == T.charAt(j-1)) dp[i][j] = dp[i-1][j-1] + 1;
            else dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
    return dp[m][n];
}
```

1.5.23 1092. Shortest Common Supersequence - Hard

Given two strings str1 and str2, return the shortest string that has both str1 and str2 as subsequences. If there are multiple valid strings, return any of them.

A string s is a subsequence of string t if deleting some number of characters from t (possibly 0) results in the string s.

- 参考的标准答案:

```
public void longestCommonSubsequence(String S, String T) { // 标准模板, 记住
    int m = S.length();
    int n = T.length();
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            if (S.charAt(i-1) == T.charAt(j-1)) dp[i][j] = dp[i-1][j-1] + 1;
            else dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
}
int [][] dp;
public String shortestCommonSupersequence(String s, String t) {
    int m = s.length();
    int n = t.length();
    dp = new int [m+1][n+1];
    longestCommonSubsequence(s, t); // fill dp table
    int i = m, j = n;
    StringBuilder sb = new StringBuilder();
    while (i-1 >= 0 && j-1 >= 0) {
        if (s.charAt(i-1) == t.charAt(j-1)) {
            sb.append(s.charAt(i-1));
            --i;
            --j;
        } else {
            if (dp[i][j] == dp[i-1][j]) {
                sb.append(s.charAt(i-1));
                --i;
            } else {
                sb.append(t.charAt(j-1));
                --j;
            }
        }
    }
    if (i > 0) sb.append((new StringBuilder(s.substring(0, i))).reverse());
    if (j > 0) sb.append((new StringBuilder(t.substring(0, j))).reverse());
    return sb.reverse().toString();
}
```

- 自己写的

```
public String getLongestCommonSubsequence(String S, String T) { // 标准模板, 记住
    int m = S.length();
    int n = T.length();
    int [][] dp = new int [m+1][n+1];
    for (int i = 1; i <= m; i++)
        for (int j = 1; j <= n; j++)
            if (S.charAt(i-1) == T.charAt(j-1)) dp[i][j] = dp[i-1][j-1] + 1;
            else dp[i][j] = Math.max(dp[i-1][j], dp[i][j-1]);
    int i = m, j = n;
    StringBuilder sb = new StringBuilder();
    while (i-1 >= 0 && j-1 >= 0) {
        if (S.charAt(i-1) == T.charAt(j-1)) {
            sb.insert(0, S.charAt(i-1));
            --i;
            --j;
        } else {
            if (dp[i-1][j] >= dp[i][j-1]) --i;
            else --j;
        }
    }
    return sb.toString();
}
public String shortestCommonSupersequence(String s, String t) {
    int m = s.length();
    int n = t.length();
    int i = 0, j = 0;
    String sub = getLongestCommonSubsequence(s, t);
    String res = "";
    for (char c : sub.toCharArray()) {
        while (s.charAt(i) != c) {
            res += s.charAt(i);
            i++;
        }
        while (t.charAt(j) != c) {
            res += t.charAt(j);
            j++;
        }
        res += c;
        i++;
        j++;
    }
    return res + s.substring(i) + t.substring(j);
}
```

1.5.24 546. Remove Boxes - Hard

You are given several boxes with different colors represented by different positive numbers.

You may experience several rounds to remove boxes until there is no box left. Each time you can choose some continuous boxes with the same color (i.e., composed of k boxes, k >= 1), remove them and get k * k points.

Return the maximum points you can get.

```
// 定义 dp[l][r][k] 表示在 [l, r] 区间并且在后面包含了 k 个与 boxes[r] 相同颜色的 boxes 的情况下, 可以获得的最大得分, 显然题目要求的就是 dp[0][boxes.size() - 1]
// 首先将 dp[l][r][k] 的值初始化为 dp[l][r - 1][0] + (k + 1)^2, 表示首先消除 l 到 r-1 之间的 boxes, 然后将 boxes[r] 连同后面的 k 个 boxes 一起消除。
// 然后就尝试对 dp[l][r][k] 进行更新了:
// 如果在 l 到 r-1 区间内有 boxes[i] 和 boxes[r] 相同的字符, 那么可以尝试首先将区间 [i + 1, r - 1] 消除, 这样 i 就和后面的 k + 1 个 boxes 连起来了,
// 其可以获得分数就是需要进一步计算的 dp[l][i][k + 1]。
private int dfs(int [] arr, int i, int j, int  k) {
    if (i > j) return 0;
    if (dp[i][j][k] > 0) return dp[i][j][k];
    int res = dfs(arr, i, j-1, 0) + (k+1)*(k+1);
    for (int x = i; x < j; x++)
        if (arr[x] == arr[j]) {
            res = Math.max(res, dfs(arr, i, x, k+1) + dfs(arr, x+1, j-1, 0));
        }
    return dp[i][j][k] = res;
}
int [][][] dp;
int n;
public int removeBoxes(int[] boxes) {
    n = boxes.length;
    dp = new int [n][n][n];
    return dfs(boxes, 0, n-1, 0);
}
```

1.5.25 1531. String Compression II - Hard

Run-length encoding is a string compression method that works by replacing consecutive identical characters (repeated 2 or more times) with the concatenation of the character and the number marking the count of the characters (length of the run). For example, to compress the string "aabccc" we replace "aa" by "a2" and replace "ccc" by "c3". Thus the compressed string becomes "a2bc3".

Notice that in this problem, we are not adding ’1’ after single characters.

Given a string s and an integer k. You need to delete at most k characters from s such that the run-length encoded version of s has minimum length.

Find the minimum length of the run-length encoded version of s after deleting at most k characters.

```
 private int dfs(char [] s, int idx, int cnt) { // 求从下标 index 开始向后，所有长度为 count 的子序列中，编码后的最小长度
    if (cnt == 0) return 0;
    if (idx == n) return Integer.MAX_VALUE;   // 当下标越界时还未找到长度为 count 的子序列
    if (dp[idx][cnt] > 0) return dp[idx][cnt];
    int min = Integer.MAX_VALUE, leftCnt = 0;
    boolean [] vis = new boolean [26];
    for (int i = idx; i < n; i++) {
        if (vis[s[i]-'a']) continue;   // 优化：已处理过当前的字母，跳过
        if (idx > 0 && s[i] == s[idx-1]) continue;
        vis[s[i]-'a'] = true;
        leftCnt = 0;
        for (int j = i; j < n; j++) {
            if (s[j] != s[i]) continue;
            leftCnt++;
            if (cnt - leftCnt < 0) break;  // 如果左半部分长度大于子序列长度，退出
            int right = dfs(s, j+1, cnt - leftCnt);
            if (right == Integer.MAX_VALUE) continue;
            int left = String.valueOf(leftCnt).length();
            min = Math.min(min, left + right + (left == 1 && leftCnt == 1 ? 0 : 1));
        }
    }
    return dp[idx][cnt] = min;
}
int [][] dp;
int n;
public int getLengthOfOptimalCompression(String s, int k) {
    n = s.length();
    dp = new int [n][n-k+1];
    return dfs(s.toCharArray(), 0, n-k);
}
```

1. 决策类 DP 总结

   - https://leetcode-cn.com/problems/string-compression-ii/solution/jie-ti-si-kao-guo-cheng-yu-jie-fa

1.5.26 1000. Minimum Cost to Merge Stones

There are n piles of stones arranged in a row. The ith pile has stones[i] stones. A move consists of merging exactly k consecutive piles into one pile, and the cost of this move is equal to the total number of stones in these k piles. Return the minimum cost to merge all piles of stones into one pile. If it is impossible, return -1.

```
public int mergeStones(int[] stones, int k) {
    int n = stones.length;
    if ((n-1) % (k-1) != 0) return -1;
    int [][] dp = new int[n][n];
    int [] pre = new int[n+1];
    for (int i = 1; i <= n; i++)
        pre[i] = pre[i-1] + stones[i-1];
    int j = 0;
    for (int len = k; len <= n; len++) {
        for (int i = 0; i+len-1 < n; i++) {
            j = i + len -1;
            dp[i][j] = Integer.MAX_VALUE; // have to initialize it here !!!
            for (int x = i; x < j; x += k-1)
                dp[i][j] = Math.min(dp[i][j], dp[i][x] + dp[x+1][j]);
            if ((j - i) % (k - 1) == 0) // 如果总长度满足合并只剩一个数的条件，则可以再合并一次
                dp[i][j] += pre[j+1] - pre[i];
        }
    }
    return dp[0][n-1];
}
```

1.5.27 1039. Minimum Score Triangulation of Polygon

You have a convex n-sided polygon where each vertex has an integer value. You are given an integer array values where values[i] is the value of the ith vertex (i.e., clockwise order). You will triangulate the polygon into n - 2 triangles. For each triangle, the value of that triangle is the product of the values of its vertices, and the total score of the triangulation is the sum of these values over all n - 2 triangles in the triangulation. Return the smallest possible total score that you can achieve with some triangulation of the polygon.

```java
// 动态规划，递归可以使逻辑简单（本质还是动态规划）将多边形起
// 始位置设为 start，end，用一个数组 dp 来记录任意起始位置的 score
// 为了计算 dp[start][end]，我们用一个 index k 在 start 到 end 之间遍
// 历 dp[start][end] = min(dp[start][k] + dp[k][end] + A[start]
// * A[k] * A[end]) 结果为 dp[0][n - 1] 注意：相邻的 dp[i][i + 1]
// = 0，因为两条边无法组成三角形
private int dfs(int [] arr, int x, int y) {
    if (y - x < 2) return dp[x][y] = 0;
    if (dp[x][y] > 0) return dp[x][y];
    int min = Integer.MAX_VALUE;
    for (int i = x+1; i < y; i++)
        min = Math.min(min, dfs(arr, x,  i) + dfs(arr, i, y) + arr[x]*arr[i]*arr[y]);
    return dp[x][y] = min;
}
int [][] dp;
int n;
public int minScoreTriangulation(int[] arr) {
    n = arr.length;
    dp = new int [n][n];
    return dfs(arr, 0, n-1);
}
```

1.5.28 375. Guess Number Higher or Lower II - Medium

We are playing the Guessing Game. The game will work as follows:

I pick a number between 1 and n.

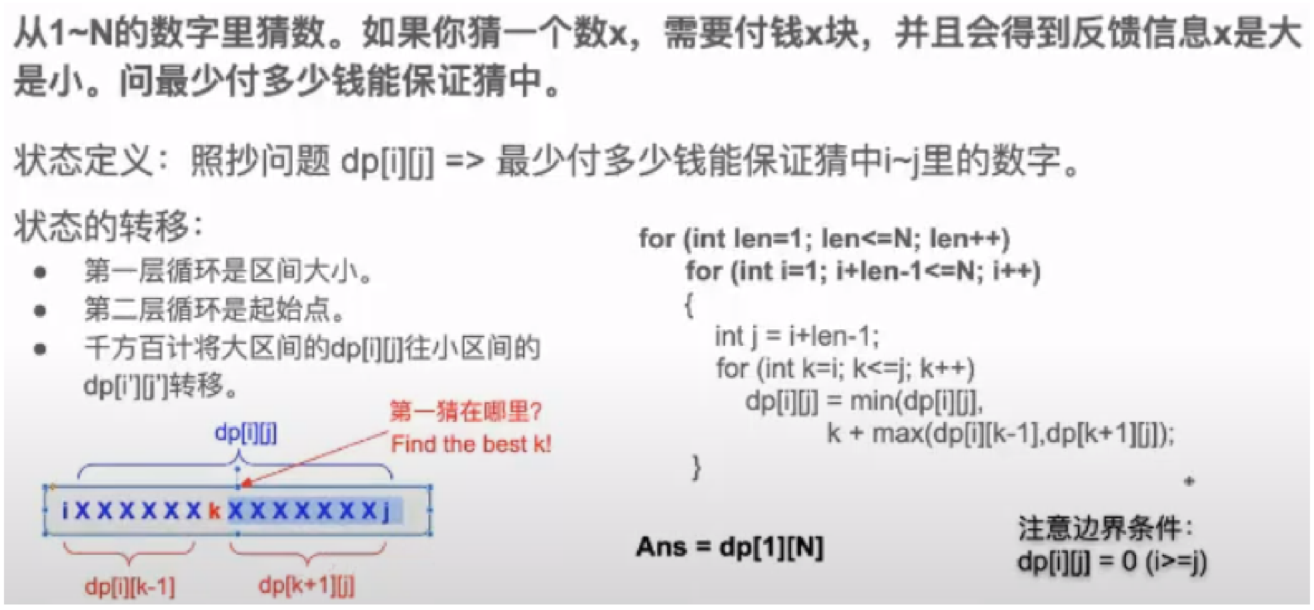
You guess a number.

If you guess the right number, you win the game.

If you guess the wrong number, then I will tell you whether the number I picked is higher or lower, and you will continue guessing.

Every time you guess a wrong number x, you will pay x dollars. If you run out of money, you lose the game.

Given a particular n, return the minimum amount of money you need to guarantee a win regardless of what number I pick.



从1~N的数字里猜数。如果你猜一个数x，需要付钱x块，并且会得到反馈信息x是大是小。问最少付多少钱能保证猜中。

状态定义：照抄问题 dp[i][j] => 最少付多少钱能保证猜中i~j里的数字。

状态的转移：

- 第一层循环是区间大小。
- 第二层循环是起始点。
- 千方百计将大区间的dp[i][j]往小区间的dp[i'][j']转移。

```
for (int len=1; len<=N; len++)
    for (int i=1; i+len-1<=N; i++)
    {
        int j = i+len-1;
        for (int k=i; k<=j; k++)
            dp[i][j] = min(dp[i][j],
                    k + max(dp[i][k-1],dp[k+1][j]);
    }
```

Ans = dp[1][N]

注意边界条件：
dp[i][j] = 0 (i>=j)

```java
private int dfs(int l, int r) {
    if (dp[l][r] > 0) return dp[l][r];
    if (l == r) return dp[l][r] = 0;
    if (l == r-1) return dp[l][r] = Math.min(l, r);
    int min = Integer.MAX_VALUE;
    for (int i = l; i <= r; i++)
        min = Math.min(min, i + Math.max((i == r ? i : dfs(i+1, r)), (i == l ? i : dfs(l, i-1))));
    return dp[l][r] = min;
}
int [][] dp;
public int getMoneyAmount(int n) {
    dp = new int[n+1][n+1];
    return dfs(1, n);
}
```

1.5.29 1478. Allocate Mailboxes - Hard

Given the array houses and an integer k. where houses[i] is the location of the ith house along a street, your task is to allocate k mailboxes in the street.

Return the minimum total distance between each house and its nearest mailbox.

The answer is guaranteed to fit in a 32-bit signed integer.

解题思路分析：

对于如何安排邮箱位置，看到很多文章说应放在中位数的位置上，比如一共有 1，2，3，4 这 4 间房屋，不论房屋间的距离是多少，如果只有一个邮箱的话，放在房间 2 处（3 也可以）最为合理。这个说法虽然正确，但实际上并不恰当。我们简单的讨论一下这个问题：

```
1.当只有 1 栋房屋，1个邮箱时，显然将邮箱放在房屋处最为合理，这时邮箱与房屋的距离为 0。
2.当有 2 栋房屋，1个邮箱时，比如房屋 1 在坐标 0 处，房屋 2 在坐标 10 处，此时如果将邮箱放在坐标 0 的话，它与两栋房屋的距离和为 10。
  放在坐标 10 的情况下距离和也为 10。另外我们可以看出，不论邮箱放在两栋房屋之间的任意位置上，它与房屋的距离和都是 10。因此
  通过此例可得出，中位数的说法虽然正确，但并不全面，不过这不影响本题解题，对于本题，我们统一将邮箱安排在房屋位置上是为
  了方便计算，因此才得出中位数的说法（本例房屋 1 和 2 都可以看做是中位数）。
3.当有 3 栋房屋，1个邮箱时，此时通过上面的例子可知，对于两侧的房子，将邮筒放在他们之间的任何位置对于结果没有任何影响，距
  离和都是两栋房子间的距离。但邮箱的位置会对中间的房子产生影响，因此，将其放置在中间房子的坐标上最为合理，这样邮箱与中
  间房屋的距离为 0，可使得全局总距离最小。而中间的房屋正是 3 个房屋的中位数。
4.当有 4 栋房屋，1个邮箱时，与上例同理，对于两侧的房子，将邮筒放在他们之间的任何位置对于结果没有任何影响，因此邮箱可以考
  虑放在中间两个房屋的任何一个位置上。另外对于中间两个房屋，不论邮箱放置在其任何一个位置上，对于总距离都不会产生影响
 （这相当于第 2 条）。
```

因此我们可以得出结论，当有 N 栋房屋，1 个邮箱时，我们将邮箱放在房屋下标的中位数上最为合理。那么，如果有多个邮箱时该怎么办？其实也不难，本题最终可以理解为，我们将一个房屋数组分割为 K 个子数组（k 为邮筒个数），每一个子数组中放置一个邮筒，求最优分割方式。这就变为了经典的动态规划 DP 问题，对于 DP 问题我习惯采用递归加记忆数组的方式，本题我们也采用递归方式讲解。

首先建立一个递归函数，参数为当前子区间开始位置 index，以及剩余未分配邮筒个数 k。起始时，子区间开始位置为下标 0，邮筒个数为题目给定的整数 k。递归时，当前子区间的开始坐标是参数 index，结束坐标范围理论上可以是当前 index 到数组末尾为止，不过这里有一处可以优化，即要保证剩下的 k-1 个邮筒都能分配出去的话，还需要至少 k-1 个子区间，也就是说除了当前子区间外还至少需要 k-1 个房屋，因此当前子区间的结束坐标范围应该是当前 index 到 length-k 为止。我们从 index 循环至 length-k，分别作为当前子区间的结束位置 end。并通过中位数方式求出当前子区间 [index, end] 放置邮筒后的距离和（后文会给出方法）。然后将 end 加一作为下一个子区间的开始位置，同时 k 值减去一作为参数传入递归子问题中继续求解。递归函数的返回值加上当前子区间的距离和即是选择当前子区间范围后的一个结果 sum。循环完所有当前子区间的结束位置 end 之后，所有 sum 中的最小值即是最优方案，也是本层递归的返回值。

接下来再为递归加上一个记忆数组。记忆数组相当于动态规划中使用到的 DP 数组。由于递归函数中存在 2 个变量，因此我们需要使用一个 2 维数组来描述该递归函数，并记录它的返回值。

最后，上文中提到需要求解子区间内放置一个邮筒后所有房屋与邮筒的距离和。这个问题没有太好的方式，只能暴力累加每个房屋与中位数房屋所在位置的距离。为了提高效率，我们可以事先计算好所有区间（排列组合）内放置一个邮筒时的距离和，方便递归中使用，也避免重复运算。这里可能有人会提出质疑，既然递归方法中已经使用了记忆数组，目的就是防止重复计算，这里为什么还担心重复计算距离和呢？原因很简单，记忆数组是二维数组，即在两个条件都满足的情况下才会使用记忆数组中的数据，比如我们计算过以下标 5 作为子区间起点，并且当前还剩 2 个油桶的递归函数返回值为 x，即 memo[4, 5]=x，再次遇到相同问题时我们可以直接返回 x。但是遇到 memo^{4,2}或者 memo^{4,3}时，我们尚未做出过计算，同样还会进入到递归函数内部，如果没有事前计算好下标 5 到 end（end 取值范围是 5 到 length-k）的距离和的话，还要重复计算一遍。

对于上述问题，还有一个更好的优化方式即再建立一个保存距离和的记忆数组，计算一个距离和记录一个，方便下次使用。

```
private int getDist(int [] arr, int i, int j) { // 求区间 start 到 end 间放置邮筒后的距离和 i: left, j: right
    if (dist[i][j] > 0) return dist[i][j];
    int m = i + (j-i)/2, v = arr[m], sum = 0;
```

[4]DEFINITION NOT FOUND.

[5]DEFINITION NOT FOUND.

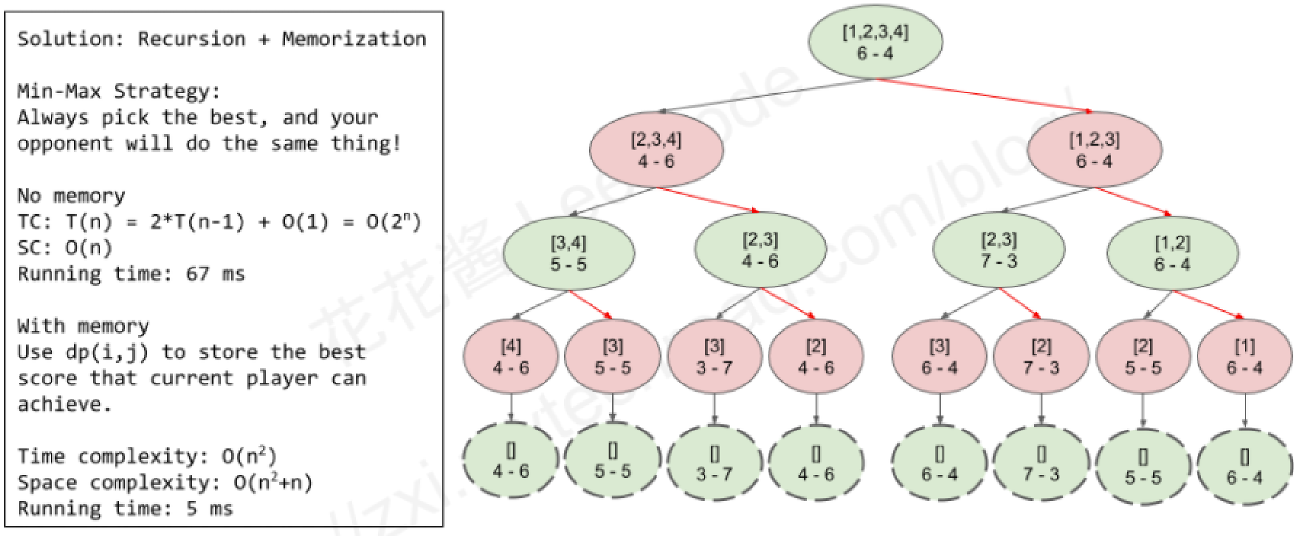
```java
    for (int k = i; k <= j; k++)
        sum += Math.abs(arr[k] - v);
    return dist[i][j] = sum;
}
private int dfs(int [] arr, int idx, int k) {  // idx: 待分割大子区间的起始坐标; k: 待分割成的子区间的个数
    if (idx == n || idx == n-k) return 0;
    if (dp[idx][k] > 0) return dp[idx][k];
    if (k == 1) return dp[idx][k] = getDist(arr, idx, n-1);
    int res = Integer.MAX_VALUE;
    for (int i = idx; i < n-(k-1); i++)
        res = Math.min(res, getDist(arr, idx, i) + dfs(arr, i+1, k-1));
    return dp[idx][k] = res;
}
int [][] dp;
int [][] dist; // 这也是一种记忆数组优化
int n;
public int minDistance(int [] houses, int k) {
    n = houses.length;
    dist = new int [n][n];
    dp = new int [n][k+1];
    Arrays.sort(houses);
    return dfs(houses, 0, k);
}
```

1.5.30 486. Predict the Winner

You are given an integer array nums. Two players are playing a game with this array: player 1 and player 2. Player 1 and player 2 take turns, with player 1 starting first. Both players start the game with a score of 0. At each turn, the player takes one of the numbers from either end of the array (i.e., nums[1] or nums[nums.length - 1]) which reduces the size of the array by 1. The player adds the chosen number to their score. The game ends when there are no more elements in the array. Return true if Player 1 can win the game. If the scores of both players are equal, then player 1 is still the winner, and you should also return true. You may assume that both players are playing optimally.

博弈类题目，使用 minMax 思想, 使自己分数最大化，对手分数尽量小，递归自顶向下求解。

该题不使用备忘机制同样能通过测试例,只不过耗时相对较长,单纯的比较取数后两 players 的分数差即可:Math.max(nums[l] - getScore(nums, l + 1, r), nums[r] - getScore(nums, l, r - 1));


```
Solution: Recursion + Memorization

Min-Max Strategy:
Always pick the best, and your
opponent will do the same thing!

No memory
TC: T(n) = 2*T(n-1) + O(1) = O(2^n)
SC: O(n)
Running time: 67 ms

With memory
Use dp(i,j) to store the best
score that current player can
achieve.

Time complexity: O(n^2)
Space complexity: O(n^2+n)
Running time: 5 ms
```

```java
private int helper( int [] arr, int i, int j) {
    if (i == j) return arr[i];
    else return Math.max(arr[i] - helper(arr, i+1, j), arr[j] - helper(arr, i, j-1));
}
public boolean PredictTheWinner(int[] nums) {
    int n = nums.length;
    if (n == 1) return true;
    return helper(nums, 0, n-1) >= 0;
}
```

1.5.31 123. Best Time to Buy and Sell Stock III

You are given an array prices where prices[i] is the price of a given stock on the ith day. Find the maximum profit you can achieve. You may complete at most two transactions. Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

```java
// k 次交易 = k 个 non-overlapping subarray
//     以这个角度去想，无非就是从两个方向扫描，
//     利用 localMin / localMax 与当前元素的差值，去构造从左边/右边扫的 dp 数组。
//     left[i] : 从最左面到 i 所能获得的最大利益（单次交易）
//     right[i] : 从 i 到最右面所能获得的最大利益（单次交易）
public int maxProfit(int[] prices) {
    int n = prices.length;
    int [] left = new int [n];
    int [] right = new int[n];
    int locMin = prices[0];
    int globalMax = Integer.MIN_VALUE;
    for (int i = 1; i < n; i++) {
        globalMax = Math.max(globalMax, Math.max(0, prices[i] - locMin));
        locMin = Math.min(locMin, prices[i]);
        left[i] = globalMax;
    }
    int locMax = prices[n-1];
    globalMax = Integer.MIN_VALUE;
    for (int i = n-2; i >= 0; i--) {
        globalMax = Math.max(globalMax, Math.max(0, locMax - prices[i]));
        locMax = Math.max(locMax, prices[i]);
        right[i] = globalMax;
    }
    globalMax = 0;
    for (int i = 0; i < n-1; i++)
        globalMax = Math.max(globalMax, left[i] + right[i+1]);
    globalMax = Math.max(globalMax, left[n-1]);
    return globalMax;
}
```

1.5.32 188. Best Time to Buy and Sell Stock IV

You are given an integer array prices where prices[i] is the price of a given stock on the ith day, and an integer k. Find the maximum profit you can achieve. You may complete at most k transactions. Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

```java
public int maxProfit(int k, int [] prices) {
    if (prices == null || prices.length == 0) return 0;
    int n = prices.length;
    int diff = 0;
    if (k >= n/2) {
        int res = 0;
        for (int i = 1; i < n; i++) {
           diff = prices[i] - prices[i-1];
            if (diff > 0) res += diff;
        }
        return res;
    }
    int [][] locMax = new int [n][k+1];
    int [][] gloMax = new int [n][k+1];
    for (int i = 1; i < n; i++) {
        diff = prices[i] - prices[i-1];
        for (int j = 1; j <= k && j * 2 <= i+1; j++) {
            locMax[i][j] = Math.max(locMax[i-1][j], gloMax[i-1][j-1]) + diff;
            gloMax[i][j] = Math.max(locMax[i][j], gloMax[i-1][j]);
        }
    }
    return gloMax[n-1][k];
}
```

1.5.33 714. Best Time to Buy and Sell Stock with Transaction Fee

You are given an array prices where prices[i] is the price of a given stock on the ith day, and an integer fee representing a transaction fee. Find the maximum profit you can achieve. You may complete as many transactions as you like, but you need to pay the transaction fee for each transaction. Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

```java
public int maxProfit(int[] prices, int fee) {
    int n = prices.length;
    int [] sold = new int[n];
    int [] hold = new int[n];
    hold[0] = -prices[0];
    for (int i = 1; i < n; i++) {
        sold[i] = Math.max(sold[i-1], hold[i-1]+prices[i]-fee);
        hold[i] = Math.max(hold[i-1], sold[i-1]-prices[i]);
    }
    return sold[n-1];
}
```

1.5.34 309. Best Time to Buy and Sell Stock with Cooldown

You are given an array prices where prices[i] is the price of a given stock on the ith day. Find the maximum profit you can achieve. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times) with the following restrictions: After you sell your stock, you cannot buy stock on the next day (i.e., cooldown one day). Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

- 感觉自己 DP 的能力还是太弱，越是这样越需要迎难而上。
- 这个题和 714. Best Time to Buy and Sell Stock with Transaction Fee 比较像。做题方法都是使用了两个数组：
- cash 该天结束手里没有股票的情况下，已经获得的最大收益
- hold 该天结束手里有股票的情况下，已经获得的最大收益
- 状态转移方程式这样的：
  - cash[i] 代表的是手里没有股票的收益，这种可能性是今天卖了或者啥也没干。max(昨天手里有股票的收益 + 今天卖股票的收益，昨天手里没有股票的收益)，即 max(sell[i - 1], hold[i - 1] + prices[i])；
  - hold[i] 代表的是手里有股票的收益，这种可能性是今天买了股票或者啥也没干，今天买股票必须昨天休息。所以为 max(今天买股票是前天卖掉股票的收益-今天股票的价格，昨天手里有股票的收益）。即 max(hold[i - 1], sell[i - 2] - prices[i])。
- 另外需要注意的是，题目说的是昨天卖了股票的话今天不能买，对于开始的第一天，不可能有卖股票的行为，所以需要做个判断。
- 该算法的时间复杂度是 O(n)，空间复杂度是 O(n)。

```
public int maxProfit(int[] prices) {
    int n = prices.length;
    int [] sold = new int [n];
    int [] hold = new int [n];
    hold[0] = -prices[0];
    for (int i = 1; i < n; i++) {   // ith: do nothing, selling hold[i-1]
        sold[i] = Math.max((i >= 2 ? sold[i-1] : 0), hold[i-1] + prices[i]); // 今天卖了股票, 或者今天什么也没有干
        hold[i] = Math.max(hold[i-1], (i >= 2 ? sold[i-2] : 0) - prices[i]); // 今天买了股票, 或者今天什么也没有干
    }
    return Math.max(sold[n-1], hold[n-1]);
}
```

1.5.35 673. Number of Longest Increasing Subsequence

Given an integer array nums, return the number of longest increasing subsequences. Notice that the sequence has to be strictly increasing.

```
public int findNumberOfLIS(int[] nums) { // dynamic programming
    int n = nums.length;
    int [][] arr = new int[n][2];
    int maxLength = 1;
    for (int i = 0; i < n; i++)
        Arrays.fill(arr[i], 1);
    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            if (nums[j] > nums[i]) {
                if (arr[i][0] + 1 > arr[j][0]) {
                    arr[j][0] = arr[i][0] +1;
                    arr[j][1] = arr[i][1];
                    maxLength = Math.max(maxLength, arr[j][0]);
                } else if (arr[i][0] + 1 == arr[j][0])
                    arr[j][1] += arr[i][1];
            }
       }
    }
    int cnt = 0;
    for (int i = 0; i < n; i++)
        if (arr[i][0] == maxLength) cnt += arr[i][1];
    return cnt;
}
```

1.5.36 1896. Minimum Cost to Change the Final Value of Expression - Hard

You are given a valid boolean expression as a string expression consisting of the characters ’1’,’0’,’&’ (bitwise AND operator),’|’ (bitwise OR operator),’(’, and ’)’.

For example, "()1|1" and "(1)&()" are not valid while "1", "(((1))|(0))", and "1|(0&(1))" are valid expressions. Return the minimum cost to change the final value of the expression.

For example, if expression = "1|1|(0&0)&1", its value is 1|1|(0&0)&1 = 1|1|0&1 = 1|0&1 = 1&1 = 1. We want to apply operations so that the new expression evaluates to 0. The cost of changing the final value of an expression is the number of operations performed on the expression. The types of operations are described as follows:

Turn a ’1’ into a ’0’. Turn a ’0’ into a ’1’. Turn a ’&’ into a ’|’. Turn a ’|’ into a ’&’. Note: ’&’ does not take precedence over ’|’ in the order of calculation. Evaluate parentheses first, then in left-to-right order.

```
private int [] getMinOperations(int va, int vb, int ca, int cb, char sign) {
    if (sign == '&') {
        if (va == 1 && vb == 1)      // 1&1, 将其中一个 1 反转为 0
            return new int [] {1, Math.min(ca, cb)};
        else if (va == 0 && vb == 0) // 0&0, 将其中一个 0 反转为 1, 并将 & 反转为 |
            return new int [] {0, Math.min(ca, cb) + 1};
        else return new int [] {0, 1}; // 1&0, 将 & 反转为 |
    } else {
        if (va == 1 && vb == 1)        // 1|1, 将其中一个 1 反转为 0, 并将 | 反转为 &
            return new int [] {1, Math.min(ca, cb) + 1};
        else if (va == 0 && vb == 0)   // 0|0, 将其中一个 0 反转为 1
            return new int [] {0, Math.min(ca, cb)};
        else return new int [] {1, 1}; // 1|0, 将 | 反转为 &
    }
}
public int minOperationsToFlip(String expression) {
    Stack<Integer> res = new Stack<>();
    Stack<Character> sgn = new Stack<>();
    Stack<Integer> cnt = new Stack<>();
    for (char c : expression.toCharArray()) {
        if (c == '(' || c == '&' || c == '|') {
            sgn.push(c);
            continue;
        } else if (c == ')') sgn.pop();
        else {
            res.push((int)(c - '0'));
            cnt.push(1);
        }
        if (res.size() > 1 && sgn.peek() != '(') {
            int [] loc = getMinOperations(res.pop(), res.pop(), cnt.pop(), cnt.pop(), sgn.pop());
            res.push(loc[0]); // expr results
            cnt.push(loc[1]); // min operations
        }
    }
    return cnt.peek();
}
```

1.5.37 823. Binary Trees With Factors

Given an array of unique integers, arr, where each integer arr[i] is strictly greater than 1. We make a binary tree using these integers, and each number may be used for any number of times. Each non-leaf node’s value should be equal to the product of the values of its children. Return the number of binary trees we can make. The answer may be too large so return the answer modulo 109 + 7.

```
public int numFactoredBinaryTrees(int[] arr) {
    int n = arr.length;
    Arrays.sort(arr);
    Map<Integer, Long> dp = new HashMap<>();
    int mod = 1_000_000_007;
    long res = 0;
    long max = 0;
    for (int i = 0; i < n; i++) {
        dp.put(arr[i], 1l);
        for (int j = 0; j < i; j++) {
            if (arr[i] % arr[j] == 0 && dp.containsKey(arr[i]/arr[j])) {
                max = dp.get(arr[i]) + dp.get(arr[j]) * dp.get(arr[i]/arr[j]);
                dp.put(arr[i], max % mod);
            }
        }
        res += dp.get(arr[i]);
        res %= mod;
    }
    return (int)(res % mod);
}
```

1.5.38 907. Sum of Subarray Minimums

Given an array of integers arr, find the sum of min(b), where b ranges over every (contiguous) subarray of arr. Since the answer may be large, return the answer modulo 109 + 7.

```
public int sumSubarrayMins(int[] arr) {
    int n = arr.length;
    // for each A[i], find k <= i <= j, so that A[i] is the min from [k,j]
    // sum += A[i] * (i-k+1) * (j-i+1)
    // so we need to find the next min to the right and to the left
    //这个过程可以简化为使用一个栈。对于被某个数从栈中弹出的数而言，它右侧第一个比它小的数就是这个数。所以我们可以对所有被弹出的数得到左侧的区间范围和右侧的区间范围。
    long [] right = new long [n];  // next smaller element index to the right
    long [] left = new long[n];    // next smaller element index to the left
    Stack<Integer> s = new Stack<>();
    for (int i = 0; i < n; i++) {
        while (!s.isEmpty() && arr[i] <= arr[s.peek()]) {
            right[s.pop()] = i-1;
        }
        s.push(i);
    }
    while (!s.isEmpty()) {
        right[s.pop()] = n-1;
    }
    s.clear();
    for (int i = n-1; i >= 0; i--) {
        while (!s.isEmpty() && arr[i] < arr[s.peek()])
            left[s.pop()] = i+1;
        s.push(i);
    }
    while (!s.isEmpty())
        left[s.pop()] = 0;
    long sum = 0;
    long leftsize = 0, rightsize = 0;
    for (int i = 0; i < n; i++) {
        leftsize = i - left[i] +1;
        rightsize = right[i] - i + 1;
        sum += arr[i] * leftsize * rightsize;
        sum %= mod;
    }
    return (int)sum;
}
int mod = 1_000_000_007;
public int sumSubarrayMins(int[] arr) {
    int n = arr.length;
    long [] left = new long[n];
    long [] right = new long[n];
    long sum = 0;
    long cnt = 0;
    int j = 0;
    for (int i = 0; i < n; i++) { // 计算左边比自身大的数的个数
        cnt = 1;
        j = i-1;
        while (j >= 0 && arr[j] >= arr[i]) {
            cnt += left[j];
            j -= left[j];
        }
        left[i] = cnt;
    }
    // 就是因为计算了两个方向，所以对于数组里面有相同元素的情况下，需要特别考虑一下。
    //     不能重复计算，也不能漏掉，
    //     具体就是一个方向的时候用 <=, 另外一个方向的时候用 <。这个在做的时候也 bug 了。
    for (int i = n-1; i >= 0; i--) { // 计算右边比自身大的数的个数
        cnt = 1;
        j = i+1;
        while (j < n && arr[j] > arr[i]) {
            cnt += right[j];
            j += right[j];
        }
        right [i] = cnt;
    }
    for (int i = 0; i < n; i++)
        sum += arr[i] * left[i] * right[i];
    return (int) (sum % mod);
}
```

1.5.39 1477. Find Two Non-overlapping Sub-arrays Each With Target Sum

Given an array of integers arr and an integer target. You have to find two non-overlapping sub-arrays of arr each with a sum equal target. There can be multiple answers so you have to find an answer where the sum of the lengths of the two sub-arrays is minimum. Return the minimum sum of the lengths of the two required sub-arrays, or return -1 if you cannot find such two sub-arrays.

```
// 找出数组中等于 target 的最小非重叠区间的长度, 用 dp[i] 表示当前 i 以及 i 之前的满足条件的最小区间长度, 状态更新规则为
//     dp[i]=min(dp[i-1],i-j+1) if sum[j,i]=target
//     答案更新规则
//     res=min(res,dp[j1]+ij+1)
public int minSumOfLengths(int[] arr, int target) {
    int n = arr.length;
    int [] dp = new int [n];
    Arrays.fill(dp, Integer.MAX_VALUE);
    int cur = 0, s = 0;
    int res = Integer.MAX_VALUE, minLen = Integer.MAX_VALUE;
    for (int i = 0; i < n; i++) {
        cur += arr[i];
        while (cur > target) {
            cur -= arr[s];
            s += 1;
        }
        if (cur == target) {
            int curLen = i - s + 1;
            if (s > 0 && dp[s-1] != Integer.MAX_VALUE)
                res = Math.min(res, curLen + dp[s-1]);
            minLen = Math.min(minLen, curLen);
        }
        dp[i] = minLen;
    }
    return res == Integer.MAX_VALUE ? -1 : res;
}
```

1.5.40 1771. Maximize Palindrome Length From Subsequences

You are given two strings, word1 and word2. You want to construct a string in the following manner: Choose some non-empty subsequence subsequence1 from word1. Choose some non-empty subsequence subsequence2 from word2. Concatenate the subsequences: subsequence1 + subsequence2, to make the string. Return the length of the longest palindrome that can be constructed in the described manner. If no palindromes can be constructed, return 0. A subsequence of a string s is a string that can be made by deleting some (possibly none) characters from s without changing the order of the remaining characters. A palindrome is a string that reads the same forward as well as backward.

```
public int longestPalindrome(String s, String t) { // 这个题目没有懂, 需要再好好看一下
    int m = s.length();
    int n = t.length();
    int mn = m + n;
    String st = s + t;
    int [][] dp = new int [mn][mn];
    for (int i = 0; i < mn; i++)
        dp[i][i] = 1;
    for (int l = 2; l <= mn; l++) {
        for (int i = 0, j = i+l-1; j < mn; i++,j++) { //
            if (st.charAt(i) == st.charAt(j))
                dp[i][j] = dp[i+1][j-1] + 2;
            else dp[i][j] = Math.max(dp[i+1][j], dp[i][j-1]);
        }
    }
    int ans = 0;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (s.charAt(i) == t.charAt(j))
                ans = Math.max(ans, dp[i][m+j]); //
        }
    }
    return ans;
}
```

1.5.41 907. Sum of Subarray Minimums

Given an array of integers arr, find the sum of min(b), where b ranges over every (contiguous) subarray of arr. Since the answer may be large, return the answer modulo 109 + 7.

```
public int sumSubarrayMins(int[] arr) {
    int n = arr.length;
    // for each A[i], find k <= i <= j, so that A[i] is the min from [k,j]
    // sum += A[i] * (i-k+1) * (j-i+1)
    // so we need to find the next min to the right and to the left
    //这个过程可以简化为使用一个栈。对于被某个数从栈中弹出的数而言，它右侧第一个比它小的数就是这个数。所以我们可以对所有被弹出的数得到左侧的区间范围和右侧的
    long [] right = new long [n];  // next smaller element index to the right
    long [] left = new long[n];    // next smaller element index to the left
    Stack<Integer> s = new Stack<>();
    for (int i = 0; i < n; i++) {
        while (!s.isEmpty() && arr[i] <= arr[s.peek()]) {
            right[s.pop()] = i-1;
        }
        s.push(i);
    }
    while (!s.isEmpty()) {
        right[s.pop()] = n-1;
    }
    s.clear();
    for (int i = n-1; i >= 0; i--) {
        while (!s.isEmpty() && arr[i] < arr[s.peek()])
            left[s.pop()] = i+1;
        s.push(i);
    }
    while (!s.isEmpty())
        left[s.pop()] = 0;
    long sum = 0;
    long leftsize = 0, rightsize = 0;
    for (int i = 0; i < n; i++) {
        leftsize = i - left[i] +1;
        rightsize = right[i] - i + 1;
        sum += arr[i] * leftsize * rightsize;
        sum %= mod;
    }
    return (int)sum;
}
int mod = 1_000_000_007;
public int sumSubarrayMins(int[] arr) {
    int n = arr.length;
    long [] left = new long[n];
    long [] right = new long[n];
    long sum = 0;
    long cnt = 0;
    int j = 0;
    for (int i = 0; i < n; i++) { // 计算左边比自身大的数的个数
        cnt = 1;
        j = i-1;
        while (j >= 0 && arr[j] >= arr[i]) {
            cnt += left[j];
            j -= left[j];
        }
        left[i] = cnt;
    }
    // 就是因为计算了两个方向，所以对于数组里面有相同元素的情况下，需要特别考虑一下。
    //     不能重复计算，也不能漏掉，
    //     具体就是一个方向的时候用 <=, 另外一个方向的时候用 <。这个在做的时候也 bug 了。
    for (int i = n-1; i >= 0; i--) { // 计算右边比自身大的数的个数
        cnt = 1;
        j = i+1;
        while (j < n && arr[j] > arr[i]) {
            cnt += right[j];
            j += right[j];
        }
        right [i] = cnt;
    }
    for (int i = 0; i < n; i++)
        sum += arr[i] * left[i] * right[i];
    return (int) (sum % mod);
}
```

1.5.42 689. Maximum Sum of 3 Non-Overlapping Subarrays

Given an integer array nums and an integer k, find three non-overlapping subarrays of length k with maximum sum and return them. Return the result as a list of indices representing the starting position of each interval (0-indexed). If there are multiple answers, return the lexicographically smallest one.

```java
public int[] maxSumOfThreeSubarrays(int[] nums, int k) {
    int n = nums.length;
    int [] pre = new int [n+1];
    for (int i = 1; i <= n; i++)
        pre[i] = pre[i-1] + nums[i-1];
    // left[i] 表示在区间 [0, i] 范围内长度为 k 且和最大的子数组的起始位置
    // right[i] 表示在区间 [i, n - 1] 范围内长度为 k 且和最大的子数组的起始位置
    int [] left = new int [n];
    int [] right = new int [n];
    int [] res = new int [3];
    Arrays.fill(right, n-k);
    for (int i = k, total = pre[k]-pre[0]; i < n; i++) {
        if (pre[i+1] - pre[i+1-k] > total) {
            left[i]= i+1-k;
            total = pre[i+1] - pre[i+1-k];
        } else left[i] = left[i-1];
    }
    for (int i = n-1-k, total = pre[n]-pre[n-k]; i >= 0; i--) {
        if (pre[i+k] - pre[i] >= total) {
            right[i] = i;
            total = pre[i+k] - pre[i];
        } else right[i] = right[i+1];
    }
    int max = Integer.MIN_VALUE;
    for (int i = k; i <= n-2*k; i++) {
        int l = left[i-1];
        int r = right[i+k];
        int total = (pre[i+k]-pre[i]) + (pre[k+l]-pre[l]) + (pre[r+k] - pre[r]);
        if (max < total) {
            max = total;
            res = new int [] {l, i, r};
        }
    }
    return res;
}
```

1.5.43 363. Max Sum of Rectangle No Larger Than K

Given an m x n matrix matrix and an integer k, return the max sum of a rectangle in the matrix such that its sum is no larger than k. It is guaranteed that there will be a rectangle with a sum no larger than k.

```java
public int maxSumSubmatrix(int[][] mat, int k) {
    int m = mat.length;
    int n = mat[0].length;
    if (m == 1 && n == 1) return mat[0][0];
    int [][] pre = new int [m][n];
    int res = Integer.MIN_VALUE;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            int t = mat[i][j];
            if (i > 0) t += pre[i-1][j];
            if (j > 0) t += pre[i][j-1];
            if (i > 0 && j > 0) t -= pre[i-1][j-1];
            pre[i][j] = t;
            for (int r = 0; r <= i; r++) {
                for (int c = 0; c <= j; c++) {
                    int d = pre[i][j];
                    if (r > 0) d -= pre[r-1][j];
                    if (c > 0) d -= pre[i][c-1];
                    if (r > 0 && c > 0) d += pre[r-1][c-1];
                    if (d <= k) res = Math.max(res, d);
                }
            }
        }
    }
    return res;
}
// 把二维数组按行或列拆成多个一维数组，然后利用一维数组的累加和来找符合要求的数字，
// 这里用了 lower_bound 来加快的搜索速度，也可以使用二分搜索法来替代。
public int maxSumSubmatrix(int[][] mat, int target) {
    int row = mat.length;
    int col = mat[0].length;
    int res = Integer.MIN_VALUE;
    boolean key = col > row ? false : true;
    int m = Math.min(row, col);
    int n = Math.max(row, col);
    int [] pre = new int [n];
    TreeSet<Integer> ts = new TreeSet<>(); //用来保存当前高度下，长度为从 0 开始到 k 位置的矩形的结果。理解 set 的含义是解决此题的关键。
    Integer tmp = 0;
    for (int i = 0; i < m; i++) { // 找从第 i 行开始一直到第 0 行这 i+1 行的可能组成的矩形长度
```

```
        Arrays.fill(pre, 0);
        for (int j = i; j >= 0; j--) {
            ts.clear();
            ts.add(0);
            int curSum = 0;
            for (int k = 0; k < n; k++) {
                if (key)
                    pre[k] += mat[k][j];
                else pre[k] += mat[j][k];
                curSum += pre[k];
                 // * 因为要满足  (sum-set 中的元素) <=target,
                 // * 而且 sum-set 中的元素的值要尽可能的大,
                 // * 所以也就是再求小于等于 sum-target 中满足条件的元素的最小的一个
                 // * 正好 TreeSet 中提供了这个方法 ceil(), 可以很方便的找出这个元素
                tmp = ts.ceiling(curSum - target);
                if (tmp != null) res = Math.max(res, curSum - tmp);
                ts.add(curSum);
            }
        }
    }
    return res;
}
```

1.5.44 805. Split Array With Same Average

You are given an integer array nums. You should move each element of nums into one of the two arrays A and B such that A and B are non-empty, and average(A) == average(B). Return true if it is possible to achieve that and false otherwise. Note that for an array arr, average(arr) is the sum of all the elements of arr over the length of arr.

```
public boolean splitArraySameAverage(int[] nums) {
    int n = nums.length;
    int m = n / 2;
    int sum = Arrays.stream(nums).sum();
    boolean poss = false;
    for (int i = 1; i <= m; i++)
        if (sum * i % n == 0) {
            poss = true;
            break;
        }
    if (!poss) return false;
    List<Set<Integer>> ls = new ArrayList<>();
    for (int i = 0; i <= m; i++)
        ls.add(new HashSet<Integer>());
    ls.get(0).add(0);    // 这种构建子序列和的方法, 要学习一下
    for (int v : nums)  // for each element in A, we try to add it to sums[i] by joining sums[i - 1]
        for (int i = m; i >= 1; i--)
            for (int t : ls.get(i-1))
                ls.get(i).add(t + v);
    for (int i = 1; i <= m; i++) {
        if (sum * i % n == 0 && ls.get(i).contains(sum * i / n))
            return true;
    }
    return false;
}
```

1.5.45 1981. Minimize the Difference Between Target and Chosen Elements

You are given an m x n integer matrix mat and an integer target. Choose one integer from each row in the matrix such that the absolute difference between target and the sum of the chosen elements is minimized. Return the minimum absolute difference. The absolute difference between two numbers a and b is the absolute value of a - b.

```
// DP + BitSet : 这里面有个小问题需要挑出来
// 使用一个 DP 数组存下当前行和之前行每行选一个数可能构成的和,
// 在本题中, 可以使用 BitSet (简介) 来存储之前行可以组成的和 (由于所有数的最大值为 70, 而行数最大也为 70, 故 BitSet 最大的位数即为 4900)。
// 对于当前行, 遍历 BitSet 已经 set 过的位 (即代表之前行可能组成的和), 然后加上当前数, set 新的和
// 最后遍历 BitSet, 求出当前位与 target 的最小值
public int minimizeTheDifference(int[][] mat, int target) {
    int m = mat.length;
    int n = mat[0].length;
    BitSet sum = new BitSet(); // 遍历每一行, 存下当前行和之前行可能组成的和
    for (int i = 0; i < n; i++) // 初始时存下第一行
        sum.set(mat[0][i]);
    for (int i = 1; i < m; i++) {
        BitSet newSum = new BitSet(); // 用来存新的和
        for (int j = 0; j < n; j++) {
            // 注意: 要遍历 BitSet 中的真实位, 请使用以下循环: previousSetBit() 方法 用于查找在指定的起始索引上或之前是否存在任何真位
            // for (int i = bs.length(); (i = bs.previousSetBit(i-1)) >= 0; ) {
            //     // operate on index i here
            // }
            for (int k = sum.length(); (k = sum.previousSetBit(k-1)) >= 0; ) {
                newSum.set(k+mat[i][j]);
            }
        }
        sum = newSum;
    }
    int ans = 4900;
    for (int k = sum.length(); (k = sum.previousSetBit(k-1)) >= 0;) {
        int diff = Math.abs(k - target);
        ans = Math.min(ans, diff);
    }
    return ans;
}
public int minimizeTheDifference(int[][] mat, int target) {
    int m = mat.length;
    int n = mat[0].length;
    int diff = Integer.MAX_VALUE, limit = 4900;
    int [] dp = new int[limit];
    for (int i = 0; i < n; i++) // 相当于是手工实现 java BitSet
        dp[mat[0][i]] = 1;
    for (int i = 1; i < m; i++) {
        int [] tmp = new int [limit];
        for (int v = limit-1; v >= 0; v--) {
            if (dp[v] == 0) continue;
            for (int j = 0; j < n; j++) {
                if (v + mat[i][j] < limit)
                    tmp[v+mat[i][j]] = 1;
            }
        }
        System.arraycopy(tmp, 0, dp, 0, dp.length);
    }
    for (int i = 0; i < limit; i++)
        if (dp[i] > 0) diff = Math.min(diff, Math.abs(i-target));
    return diff;  // min difference
}
```

1.5.46 805. Split Array With Same Average

You are given an integer array nums. You should move each element of nums into one of the two arrays A and B such that A and B are non-empty, and average(A) == average(B). Return true if it is possible to achieve that and false otherwise. Note that for an array arr, average(arr) is the sum of all the elements of arr over the length of arr.

```
    //     1）如果一个长度为 n 的数组可以被划分为 A 和 B 两个数组，我们假设 A 的长度小于 B 并且 A 的大小是 k，那么：total_sum / n == A_sum / k == B_sum / (n - k
    //     2）如果经过第一步的验证，发现确实有符合条件的 k，那么我们在第二步中，就试图产生 k 个子元素的所有组合，并且计算他们的和。这里的思路就有点类似于背包问题了，
    // 3）有了 sums[n - 1][k]，我们就检查 sums[n - 1][k] 中是否包含 (total_sum * k / n)。一旦发现符合条件的 k，就返回 true，否则就返回 false。
    // 在递推公式中我们发现，sums[i][j] 仅仅和 sums[i - 1][j]，sums[i][j - 1] 有关，所以可以进一步将空间复杂度从 O(n^2*M) 降低到 O(n*M)，其中 M 是 n 中的所有元
public boolean splitArraySameAverage(int[] nums) {
    int n = nums.length;
    int m = n / 2;
    int sum = Arrays.stream(nums).sum();
    boolean poss = false;
    for (int i = 1; i <= m; i++) {
        if (sum * i % n == 0) {
            poss = true;
            break;
        }
    }
    if (!poss) return false;
    List<Set<Integer>> ls = new ArrayList<>();
    for (int i = 0; i <= m; i++)
        ls.add(new HashSet<Integer>());
    ls.get(0).add(0);    // 这种构建子序列和的方法，要学习一下
    for (int v : nums) { // for each element in A, we try to add it to sums[i] by joining sums[i - 1]
        for (int i = m; i >= 1; i--) {
            for (int t : ls.get(i-1)) {
                ls.get(i).add(t + v);
            }
        }
    }
    // System.out.println("ls.size(): " + ls.size());
    // for (int z = 0; z < ls.size(); ++z) {
    //     for (Integer x : ls.get(z))
    //         System.out.print(x + ", ");
    //     System.out.print("\n");
    //     System.out.print("\n ");
    // }
    for (int i = 1; i <= m; i++) {
        if (sum * i % n == 0 && ls.get(i).contains(sum * i / n))
```

```
            return true;
    }
    return false;
}
private boolean helper(int [] arr, int curSum, int cur, int start) {
    if (cur == 0) return curSum == 0;
    if (arr[start] > curSum / cur) return false;
    for (int i = start; i < arr.length - cur + 1; i++) {
        if (i > start && arr[i] == arr[i-1]) continue;
        if (helper(arr, curSum - arr[i], cur-1, i+1)) return true;
    }
    return false;
}
public boolean splitArraySameAverage(int[] nums) {
    int n = nums.length;
    int m = n / 2;
    int sum = Arrays.stream(nums).sum();
    boolean poss = false;
    for (int i = 1; i <= m; i++) {
        if (sum * i % n == 0) {
            poss = true;
            break;
        }
    }
    if (!poss) return false;
    Arrays.sort(nums);
    for (int i = 1; i <= m; i++)
        if (sum * i % n == 0 && helper(nums, sum * i / n, i, 0)) return true;
    return false;
}
bool splitArraySameAverage(vector<int>& A) {  // https://www.cnblogs.com/grandyang/p/10285531.html
    int n = A.size(), m = n / 2, sum = accumulate(A.begin(), A.end(), 0);
    bool possible = false;
    for (int i = 1; i <= m && !possible; ++i) {
        if (sum * i % n == 0) possible = true;
    }
    if (!possible) return false;
    bitset<300001> bits[m + 1] = {1};
    for (int num : A) {
        for (int i = m; i >= 1; --i) {
            bits[i] |= bits[i - 1] << num;
        }
    }
    for (int i = 1; i <= m; ++i) {
        if (sum * i % n == 0 && bits[i][sum * i / n]) return true;
    }
    return false;
}
```

1.5.47 801. Minimum Swaps To Make Sequences Increasing - Hard

You are given two integer arrays of the same length nums1 and nums2. In one operation, you are allowed to swap nums1[i] with nums2[i].

For example, if nums1 = [1,2,3,8], and nums2 = [5,6,7,4], you can swap the element at i = 3 to obtain nums1 = [1,2,3,4] and nums2 = [5,6,7,8]. Return the minimum number of needed operations to make nums1 and nums2 strictly increasing. The test cases are generated so that the given input always makes it possible.

An array arr is strictly increasing if and only if $arr^1 < arr^2 < arr^5 < \ldots < arr[arr.length - 1]$.

```
// 设 dp[0][i] 表示不交换 A[i] 和 B[i] 在下标 i 的交换次数
// 设 dp[1][i] 表示交换 A[i] 和 B[i] 在下标 i 的交换次数
// 可以看到交换与否只取决与前一个状态, 可以将空间复杂度压缩到 O(1)
//     时间复杂度为 O(n), 空间复杂度为 O(1)
public int minSwap(int[] a, int[] b) {
    int n = a.length;
    int [][] dp = new int [2][n];
    for (int [] row : dp)
        Arrays.fill(row, Integer.MAX_VALUE);
    dp[0][0] = 0;
    dp[1][0] = 1;
    for (int i = 1; i < n; i++) {
        if (a[i] > a[i-1] && b[i] > b[i-1]) {
            dp[0][i] = Math.min(dp[0][i], dp[0][i-1]);    // 不需要交换不用增加交换次数
            dp[1][i] = Math.min(dp[1][i], dp[1][i-1] + 1);// 如果要交换前一个也必须交换才能满足递增的条件
        }
        if (a[i] > b[i-1] && b[i] > a[i-1]) {
            dp[0][i] = Math.min(dp[0][i], dp[1][i-1]);    // 表示 i - 1 位置发生交换
            dp[1][i] = Math.min(dp[1][i], dp[0][i-1] + 1);// 表示在 i - 1 不换的基础上, i 发生了交换
        }
    }
```

```java
    return Math.min(dp[0][n-1], dp[1][n-1]);
}
```

1.5.48 837. New 21 Game - Medium

Alice plays the following game, loosely based on the card game "21".

Alice starts with 0 points and draws numbers while she has less than k points. During each draw, she gains an integer number of points randomly from the range [1, maxPts], where maxPts is an integer. Each draw is independent and the outcomes have equal probabilities.

Alice stops drawing numbers when she gets k or more points.

Return the probability that Alice has n or fewer points.

Answers within 10-5 of the actual answer are considered accepted.

```java
// When the draws sum up to K, it stops, calculate the possibility K<=sum<=N.
//     Think about one step earlier, sum = K-1, game is not ended and draw largest card W.
//     K-1+W is the maximum sum could get when game is ended. If it is <= N, then for sure the possiblity when games end ans sum <= N is 1.
//     Because the maximum is still <= 1.
//     Otherwise calculate the possibility sum between K and N.
//     Let dp[i] denotes the possibility of that when game ends sum up to i.
//     i is a number could be got equally from i - m and draws value m card.
//     Then dp[i] should be sum of dp[i-W] + dp[i-W+1] + ... + dp[i-1], devided by W.
//     We only need to care about previous W value sum, accumlate winSum, reduce the possibility out of range.
//     Time Complexity: O(N).
//     Space: O(N).
public double new21Game(int n, int k, int w) { // k : threshold
    if (k == 0 || n >= (k + w)) return 1.0;
    if (k > n) return 0;
    double [] dp = new double [n+1];
    dp[0] = 1.0;
    double winSum = 1;
    double res = 0;
    for (int i = 1; i <= n; i++) {
        dp[i] = winSum / w;
        if (i < k) winSum += dp[i];
        else res += dp[i];
        if (i >= w) winSum -= dp[i-w];
    }
    return res;
}
```

1.5.49 1105. Filling Bookcase Shelves - Medium

You are given an array books where books[i] = [thicknessi, heighti] indicates the thickness and height of the ith book. You are also given an integer shelfWidth.

We want to place these books in order onto bookcase shelves that have a total width shelfWidth.

We choose some of the books to place on this shelf such that the sum of their thickness is less than or equal to shelfWidth, then build another level of the shelf of the bookcase so that the total height of the bookcase has increased by the maximum height of the books we just put down. We repeat this process until there are no more books to place.

Note that at each step of the above process, the order of the books we place is the same order as the given sequence of books.

For example, if we have an ordered list of 5 books, we might place the first and second book onto the first shelf, the third book on the second shelf, and the fourth and fifth book on the last shelf. Return the minimum possible height that the total bookshelf can be after placing shelves in this manner.

```java
// 求摆放前 i 本书需要的最小高度，首先需要求摆放前 i-1 书需要的最小高度，以此类推，最初需要计算的是摆放第 0 本书需要的最小高度，也就是 0。
// 根据前 i-1 本书求前 i 书需要的最小高度的思路是:
// 尝试 将第 i 本书放在前 i-1 本书的下面
// 以及 将前 i-1 本书的最后几本和第 i 本书放在同一层两种方案，看哪种方案高度更小就用哪种方案，依次求出摆放前 1,⋯,n 本书需要的最小高度。
public int minHeightShelves(int[][] books, int shelfWidth) {
    int [] dp = new int [books.length + 1];
    for (int i = 1; i < dp.length; i++) { // 依次求摆放前 i 本书的最小高度
        int width = books[i-1][0];
        int height = books[i-1][1];
        dp[i] = dp[i-1] + height;
        // 将前 i - 1 本书从第 i - 1 本开始放在与 i 同一层，直到这一层摆满或者所有的书都摆好
        for (int j = i-1; j > 0 && width + books[j-1][0] <= shelfWidth; j--) {
            height = Math.max(height, books[j-1][1]); // 每层的高度由最高的那本书决定
            width += books[j-1][0];
            dp[i] = Math.min(dp[i], dp[j-1] + height);// 选择高度最小的方法
        }
    }
    return dp[books.length];
}
```

1.5.50 1997. First Day Where You Have Been in All the Rooms - Medium

There are n rooms you need to visit, labeled from 0 to n - 1. Each day is labeled, starting from 0. You will go in and visit one room a day.

Initially on day 0, you visit room 0. The order you visit the rooms for the coming days is determined by the following rules and a given 0-indexed array nextVisit of length n:

Assuming that on a day, you visit room i, if you have been in room i an odd number of times (including the current visit), on the next day you will visit a room with a lower or equal room number specified by nextVisit[i] where 0 <= nextVisit[i] <= i; if you have been in room i an even number of times (including the current visit), on the next day you will visit room (i + 1) mod n. Return the label of the first day where you have been in all the rooms. It can be shown that such a day exists. Since the answer may be very large, return it modulo 109 + 7.

```java
public int firstDayBeenInAllRooms(int [] nextVisit) {
    int n = nextVisit.length, mod = (int)1e9 + 7;
    long [] dp = new long [n];
    dp[0] = 0;
    for (int i = 1; i < n; i++)
        dp[i] = (2 * dp[i-1] % mod + mod - dp[nextVisit[i-1]] + 2) % mod;
    return (int)dp[n-1];
}
```

1.5.51 943. Find the Shortest Superstring - Hard

Given an array of strings words, return the smallest string that contains each string in words as a substring. If there are multiple valid strings of the smallest length, return any of them.

You may assume that no string in words is a substring of another string in words.

- 深搜 + 记忆数组 + 裁枝

```java
private void dfs(int [] tmp, int idx, int curCost, boolean [] vis, int path) {
    if (idx == n) {
        if (curCost > maxCost) {
            maxCost = curCost;
            best = Arrays.copyOf(tmp, n); // best = tmp.clone();
        }
        return;
    }
    for (int i = 0; i < n; i++)
        if (!vis[i]) {
            int tmpCost = idx == 0 ? 0 : curCost + cost[tmp[idx-1]][i];
            int tmpPath = (path | (1 << i));
            int maxCost = dp[tmpPath][i];
            if (maxCost > 0 && tmpCost <= maxCost) continue;
            tmp[idx] = i;
            dp[tmpPath][i] = tmpCost; // need to remember the res
            vis[i] = true;
            dfs(tmp, idx+1, tmpCost, vis, tmpPath);
            vis[i] = false;
        }
}
int n, maxCost = Integer.MIN_VALUE; // 最长公共子串的长度和，越大越好
int [][] dp;
int [][] cost;
int [] best;
public String shortestSuperstring(String[] words) {
    n = words.length;
    cost = new int [n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            if (i == j) continue;
            for (int k = 1; k <= words[i].length() && k <= words[j].length(); k++) {
                if (words[i].substring(words[i].length()-k).equals(words[j].substring(0, k)))
                    cost[i][j] = k;
            }
        }
    dp = new int [1 << n][n];
    best = new int [n];
    dfs(new int [n], 0, 0, new boolean [n], 0);
    String res = words[best[0]];
    for (int i = 1; i < n; i++) {
        int costVal = cost[best[i-1]][best[i]];
        res += words[best[i]].substring(costVal);
    }
    return res;
}
```

- 动态规划

```java
public String shortestSuperstring(String[] words) {
    int n = words.length;
    String [][] dp = new String [1 << n][n];
    int [][] overlap = new int [n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            if (i == j) continue;
            for (int k = Math.min(words[i].length(), words[j].length()); k > 0; k--) {
                if (words[i].substring(words[i].length()-k).equals(words[j].substring(0, k))) {
                    overlap[i][j] = k;
                    break;
                }
            }
        }
    for (int i = 0; i < n; i++)
        dp[1 << i][i] = words[i];
    for (int mask = 1; mask < (1 << n); mask++)
        for (int j = 0; j < n; j++) {
            if ((mask & (1 << j)) == 0) continue;
            for (int i = 0; i < n; i++) {
                if (i == j || (mask & (1 << i)) == 0) continue;
                String tmp = dp[mask ^ (1 << j)][i] + words[j].substring(overlap[i][j]);
                if (dp[mask][j] == null || tmp.length() < dp[mask][j].length()) // str.isEmpty()
                    dp[mask][j] = tmp;
            }
        }
    int last = (1 << n) - 1;
    String res = dp[last][0];
    for (int i = 1; i < n; i++)
        if (dp[last][i].length() < res.length())
            res = dp[last][i];
    return res;
}
```

1.5.52 964. Least Operators to Express Number - Hard

Given a single positive integer x, we will write an expression of the form x (op1) x (op2) x (op3) x ... where each operator op1, op2, etc. is either addition, subtraction, multiplication, or division (+, -, *, or /). For example, with x = 3, we might write 3 * 3 / 3 + 3 - 3 which is a value of 3.

When writing such an expression, we adhere to the following conventions:

The division operator (/) returns rational numbers. There are no parentheses placed anywhere. We use the usual order of operations: multiplication and division happen before addition and subtraction. It is not allowed to use the unary negation operator (-). For example, "x - x" is a valid expression as it only uses subtraction, but "-x + x" is not because it uses negation. We would like to write an expression with the least number of operators such that the expression equals the given target. Return the least number of operators used.

博主看了一会儿，发现没思路就直接放弃了，直奔论坛上找解法。这里直接参考 donggua$_{fu}$ 大神的解法吧，首先处理 edge cases，当 x 等于 target 的话，不用加任何运算符，返回 0 即可。若 x 大于 target，比如 x=5，target=3，我们其实可以迅速的求出运算符的个数，因为 5 比 3 大，要凑 3 就只能先变成 1，这里就有两种变法，一种是全部都变成 1，然后来凑 3，即 5/5 + 5/5 + 5/5，这时的运算符个数是 target * 2 -1，因为加号的个数总是比除号少一个。另一种凑法就是 5 - 5/5 - 5/5，这时候的运算符个数是 (x - target) * 2，此时的加号和除号的个数相同，均为 x 和 target 的差值。

接下来就要处理 x 小于 target 的情况了，此时由于不知道 x 到底比 target 小多少，若差距太大的话，肯定不能用加号，所以应该先用乘号来让 x 变大，直到刚好大于等于 target 停止，并每次增加次数 cnt。若此时 sum 正好等于 target，太幸运了，直接返回 cnt。但通常情况下 sum 会大于 target，此时 sum - target 的差值就需要另行计算了。这里差值跟 target 的大小关系又要分为两种情况来讨论，当 sum - target < target 时，比如 x=5，sum=25，target=15，则 sum - target=10，就是说现在已经乘到了 25，但需要再减去 10，这个差值 10 可以再次调用原函数来计算，此时新的 target 代入 10 即可，记得返回值要加上 cnt。当然之后还是要再计算一下另一种凑的方法，由于 sum 超过了 target，所以回退一个 x，变成 sum / x，此时小于 target，那么它们的差值 target - (sum / x) 就可以通过再次调用函数来计算，注意这里加上 cnt 之后还要减去 1，因为回退了一个 x，少了一个乘号。最终二者的较小值即为所求，记得要加上个 1，以为多加了个运算符，参见代码如下：

```java
public int leastOpsExpressTarget(int x, int target) {
    if (x == target) return 0;
    if (x > target) return Math.min(target*2-1, (x-target)*2);
    int cnt = 0;
    long sum = x;
    while (sum  < target) {
        sum *= x;
        ++cnt;
    }
    if (sum == target) return cnt;
    int min = Integer.MAX_VALUE, max = Integer.MIN_VALUE;
    // int tmp = sum - target; // -
    if (sum - target < target)
        min = leastOpsExpressTarget(x, (int)(sum - target)) + cnt;
    max = leastOpsExpressTarget(x, (int)(target - (sum / x))) + cnt - 1;
    return Math.min(min, max) + 1; // -
}
```

- 和 race car 那道题类似。注意到，符号的添加就是对数字进行 -x^i 的操作，最后要减到 0，k = logx(t)，有两种方式，可以先到 t 前面的数字，2^k, 或者 t 后面的数字 2$^{(k+1)}$。

注意，2^k k

看了花花酱的题解，感觉更像是 bfs。cost 小的点先扩展。这个再看一下

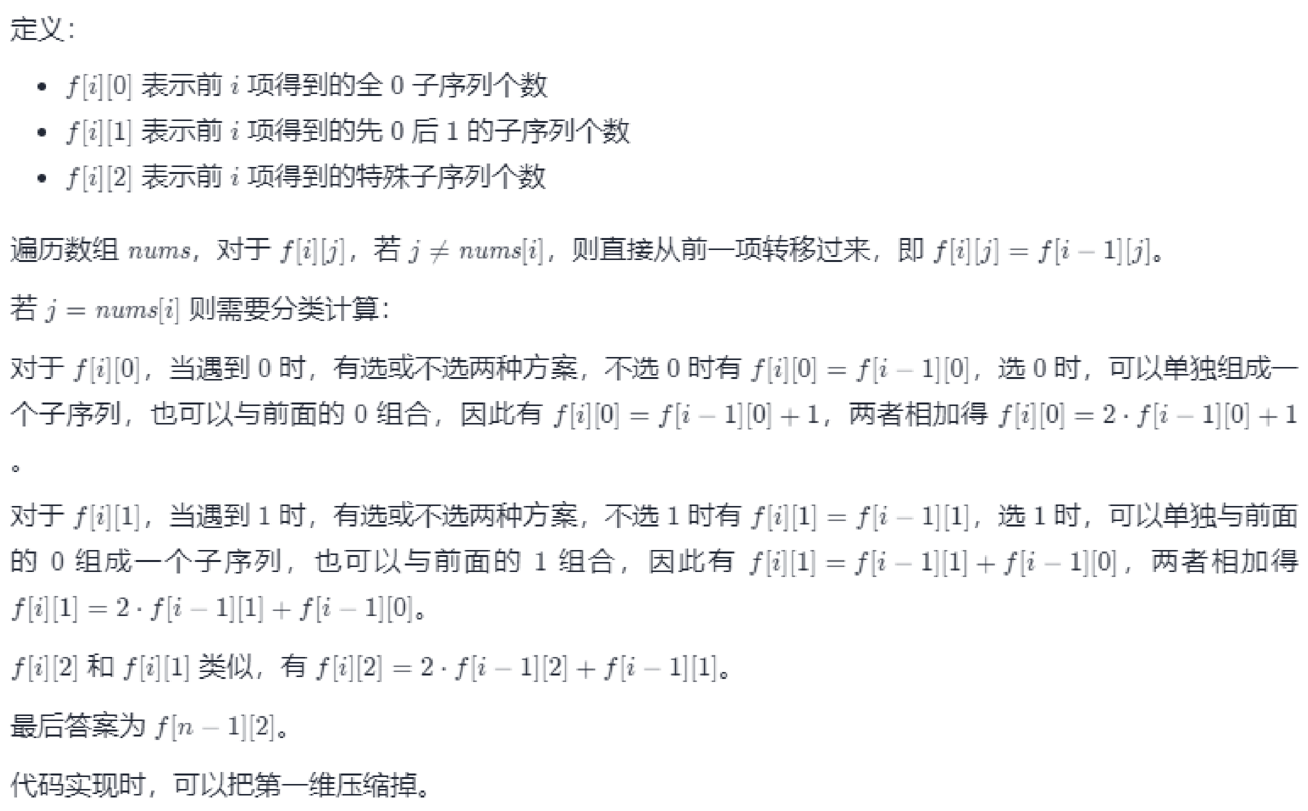
```
int leastOpsExpressTarget(int x, int target) {
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> que;
    unordered_set<int> s;
    que.emplace(0, target);
    while(!que.empty()) {
        int cost = que.top().first;
        int t = que.top().second;
        que.pop();
        if (t == 0) return cost-1;
        if (s.count(t)) continue;
        s.insert(t);
        int k = log(t) / log(x);
        int l = t - pow(x, k);
        que.emplace(cost+(k == 0 ? 2 : k), l);
        int r = pow(x, k+1) - t;
        que.emplace(cost+k+1, r);
    }
    return -1;
}
```

1.5.53 1955. Count Number of Special Subsequences - Hard

A sequence is special if it consists of a positive number of 0s, followed by a positive number of 1s, then a positive number of 2s.

For example, [0,1,2] and [0,0,1,1,1,2] are special. In contrast, [2,1,0], ^{2}, and [0,1,2,0] are not special. Given an array nums (consisting of only integers 0, 1, and 2), return the number of different subsequences that are special. Since the answer may be very large, return it modulo 109 + 7.

A subsequence of an array is a sequence that can be derived from the array by deleting some or no elements without changing the order of the remaining elements. Two subsequences are different if the set of indices chosen are different.



定义：

- $f[i][0]$ 表示前 i 项得到的全 0 子序列个数
- $f[i][1]$ 表示前 i 项得到的先 0 后 1 的子序列个数
- $f[i][2]$ 表示前 i 项得到的特殊子序列个数

遍历数组 $nums$，对于 $f[i][j]$，若 $j \neq nums[i]$，则直接从前一项转移过来，即 $f[i][j] = f[i-1][j]$。

若 $j = nums[i]$ 则需要分类计算：

对于 $f[i][0]$，当遇到 0 时，有选或不选两种方案，不选 0 时有 $f[i][0] = f[i-1][0]$，选 0 时，可以单独组成一个子序列，也可以与前面的 0 组合，因此有 $f[i][0] = f[i-1][0] + 1$，两者相加得 $f[i][0] = 2 \cdot f[i-1][0] + 1$
。

对于 $f[i][1]$，当遇到 1 时，有选或不选两种方案，不选 1 时有 $f[i][1] = f[i-1][1]$，选 1 时，可以单独与前面的 0 组成一个子序列，也可以与前面的 1 组合，因此有 $f[i][1] = f[i-1][1] + f[i-1][0]$，两者相加得 $f[i][1] = 2 \cdot f[i-1][1] + f[i-1][0]$。

$f[i][2]$ 和 $f[i][1]$ 类似，有 $f[i][2] = 2 \cdot f[i-1][2] + f[i-1][1]$。

最后答案为 $f[n-1][2]$。

代码实现时，可以把第一维压缩掉。

```java
public int countSpecialSubsequences(int[] arr) { // 去找个降维的参考一下
    int mod = (int)1e9 + 7;
    int n = arr.length;
    long [][] dp = new long [n][3];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < 3; j++) {
            if (arr[i] != j) dp[i][j] = (i == 0 ? 0 : dp[i-1][j]);
            else
                if (j == 0)
                    dp[i][j] = (i == 0 ? 0 : dp[i-1][j]) * 2 % mod + 1;
                else
                    dp[i][j] = ((i == 0 ? 0 : dp[i-1][j]) * 2 % mod + (i == 0 ? 0 : dp[i-1][j-1])) % mod;
        }
    return (int)dp[n-1][2];
}
```

1.5.54 446. Arithmetic Slices II - Subsequence - Hard

Given an integer array nums, return the number of all the arithmetic subsequences of nums.

A sequence of numbers is called arithmetic if it consists of at least three elements and if the difference between any two consecutive elements is the same.

For example, [1, 3, 5, 7, 9], [7, 7, 7, 7], and [3, -1, -5, -9] are arithmetic sequences. For example, [1, 1, 2, 5, 7] is not an arithmetic sequence. A subsequence of an array is a sequence that can be formed by removing some elements (possibly none) of the array.

For example, [2,5,10] is a subsequence of [1,2,1,2,4,1,5,10]. The test cases are generated so that the answer fits in 32-bit integer.

1. 解题思路与分析 这道题是之前那道 Arithmetic Slices 的延伸，那道题比较简单是因为要求等差数列是连续的，而这道题让我们求是等差数列的子序列，可以跳过某些数字，不一定非得连续，那么难度就加大了，但还是需要用动态规划 Dynamic Progrmming 来做。

   好，既然决定要用 DP 了，那么首先就要确定 dp 数组的定义了，刚开始我们可能会考虑使用个一维的 dp 数组，然后 dp[i] 定义为范围为 [0, i] 的子数组中等差数列的个数。定义的很简单，OK，但是基于这种定义的状态转移方程却十分的难想。我们想对于 (0, i) 之间的任意位置 j，如何让 dp[i] 和 dp[j] 产生关联呢？是不是只有 A[i] 和 A[j] 的差值 diff，跟 A[j] 之前等差数列的差值相同，才会有关联，所以差值 diff 是一个很重要的隐藏信息 Hidden Information，我们必须要在 dp 的定义中考虑进去。所以一维 dp 数组是罩不住的，必须升维，但是用二维 dp 数组的话，差值 diff 那一维的范围又是个问题，数字的范围是整型数，所以差值的范围也很大，为了节省空间，我们建立一个一维数组 dp，数组里的元素

不是数字，而是放一个 HashMap，建立等差数列的差值和当前位置之前差值相同的数字个数之间的映射。我们遍历数组中的所有数字，对于当前遍历到的数字，又从开头遍历到当前数字，计算两个数字之差 diff，如果越界了不做任何处理，如果没越界，我们让 dp[i] 中 diff 的差值映射自增 1，因为此时 A[i] 前面有相差为 diff 的 A[j]，所以映射值要加 1。然后我们看 dp[j] 中是否有 diff 的映射，如果有的话，说明此时相差为 diff 的数字至少有三个了，已经能构成题目要求的等差数列了，将 dp[j][diff] 加入结果 res 中，然后再更新 dp[i][diff]，这样等遍历完数组，res 即为所求。

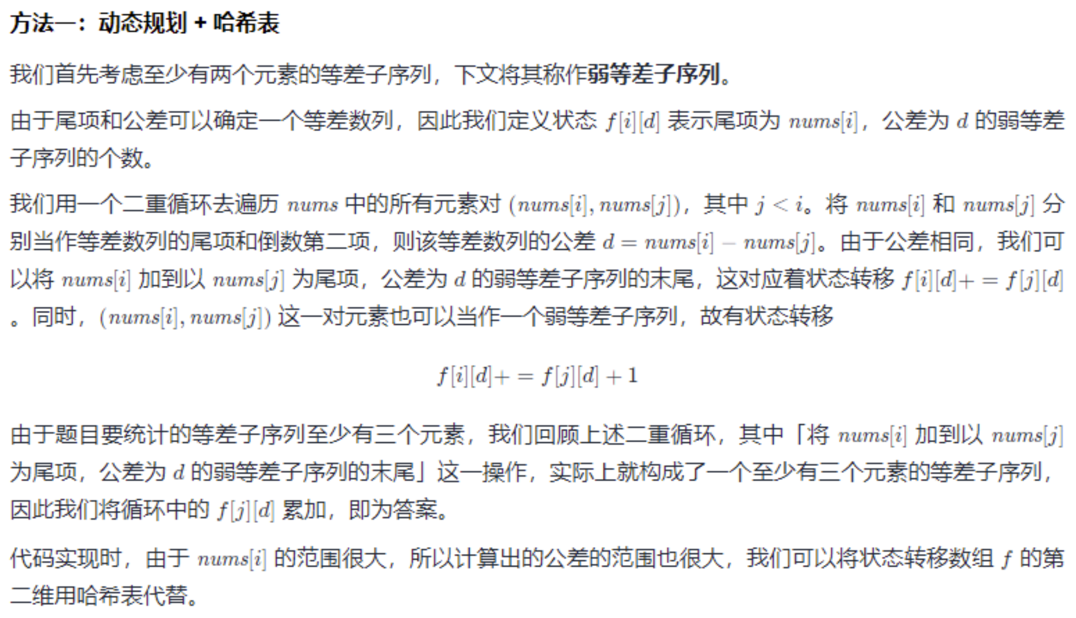
我们用题目中给的例子数组 [2，4，6，8，10] 来看，因为 2 之前没有数字了，所以我们从 4 开始，遍历前面的数字，是 2，二者差值为 2，那么在 dp^2的 HashMap 就可以建立 2->1 的映射，表示 4 之前有 1 个差值为 2 的数字，即数字 2。那么现在 i=2 指向 6 了，遍历前面的数字，第一个数是 2，二者相差 4，那么在 dp^5的 HashMap 就可以建立 4->1 的映射，第二个数是 4，二者相差 2，那么先在 dp^5的 HashMap 建立 2->1 的映射，由于 dp^2的 HashMap 中也有差值为 2 的映射，2->1，那么说明此时至少有三个数字差值相同，即这里的 [2 4 6]，我们将 dp^2中的映射值加入结果 res 中，然后当前 dp^5中的映射值加上 dp^2中的映射值。这应该不难理解，比如当 i=3 指向数字 8 时，j=2 指向数字 6，那么二者差值为 2，此时先在 dp^3建立 2->1 的映射，由于 dp^5中有 2->2 的映射，那么加上数字 8 其实新增了两个等差数列 [2,4,6,8] 和 [4,6,8]，所以结果 res 加上的值就是 dp[j][diff]，即 2，并且 dp[i][diff] 也需要加上这个值，才能使得 dp^3 中的映射变为 2->3 ，后面数字 10 的处理情况也相同，这里就不多赘述了，最终的各个位置的映射关系如下所示：

```
2     4     6     8     10
     2->1  4->1  6->1  8->1
           2->2  4->1  6->1
                 2->3  4->2
                       2->4
```

最终累计出来的结果是跟上面红色的数字相关，分别对应着如下的等差数列：

```
2->2: [2,4,6]
2->3: [2,4,6,8]    [4,6,8]
4->2: [2,6,10]
2->4: [2,4,6,8,10]    [4,6,8,10]    [6,8,10]
```

- Both time and space complexities are O(n^2)



方法一：动态规划 + 哈希表

我们首先考虑至少有两个元素的等差子序列，下文将其称作**弱等差子序列**。

由于尾项和公差可以确定一个等差数列，因此我们定义状态 $f[i][d]$ 表示尾项为 $nums[i]$，公差为 d 的弱等差子序列的个数。

我们用一个二重循环去遍历 $nums$ 中的所有元素对 $(nums[i], nums[j])$，其中 $j < i$。将 $nums[i]$ 和 $nums[j]$ 分别当作等差数列的尾项和倒数第二项，则该等差数列的公差 $d = nums[i] - nums[j]$。由于公差相同，我们可以将 $nums[i]$ 加到以 $nums[j]$ 为尾项，公差为 d 的弱等差子序列的末尾，这对应着状态转移 $f[i][d] += f[j][d]$。同时，$(nums[i], nums[j])$ 这一对元素也可以当作一个弱等差子序列，故有状态转移

$$f[i][d] += f[j][d] + 1$$

由于题目要统计的等差子序列至少有三个元素，我们回顾上述二重循环，其中「将 $nums[i]$ 加到以 $nums[j]$ 为尾项，公差为 d 的弱等差子序列的末尾」这一操作，实际上就构成了一个至少有三个元素的等差子序列，因此我们将循环中的 $f[j][d]$ 累加，即为答案。

代码实现时，由于 $nums[i]$ 的范围很大，所以计算出的公差的范围也很大，我们可以将状态转移数组 f 的第二维用哈希表代替。

Define the type of the difference as Integer type instead of Long. This is because there is no valid arithmetic subsequence slice that can have difference out of the Integer value range. But we do need a long integer to filter out those invalid cases.

Preallocate the HashMap to avoid reallocation to deal with extreme cases.

Refrain from using lambda expressions inside loops.

```java
public int numberOfArithmeticSlices(int [] arr) {
    int n = arr.length, ans = 0;
    Map<Integer, Integer> [] dp = new HashMap[n];
    dp[0] = new HashMap<>();
    for (int i = 1; i < n; i++) {
        dp[i] = new HashMap<>();
        for (int j = 0; j < i; j++) {
            long diff = (long)arr[i] - arr[j];
            if (diff > Integer.MAX_VALUE || diff < Integer.MIN_VALUE) continue;
            int dif = (int)diff;
            dp[i].put(dif, dp[i].getOrDefault(dif, 0) + 1);      // 这里先更新上
            if (dp[j].containsKey(dif)) {
                ans += dp[j].get(dif);    // 更新结果
                dp[i].put(dif, dp[i].get(dif) + dp[j].get(dif)); // 再加上之前累积的
            }
        }
    }
    return ans;
}
```

1.5.55 639. Decode Ways II - Hard

A message containing letters from A-Z can be encoded into numbers using the following mapping:

'A' -> "1" 'B' -> "2" . . . 'Z' -> "26" To decode an encoded message, all the digits must be grouped then mapped back into letters using the reverse of the mapping above (there may be multiple ways). For example, "11106" can be mapped into:

"AAJF" with the grouping (1 1 10 6) "KJF" with the grouping (11 10 6) Note that the grouping (1 11 06) is invalid because "06" cannot be mapped into 'F' since "6" is different from "06".

In addition to the mapping above, an encoded message may contain the '*' character, which can represent any digit from '1' to '9' ('0' is excluded). For example, the encoded message "1*" may represent any of the encoded messages "11", "12", "13", "14", "15", "16", "17", "18", or "19". Decoding "1*" is equivalent to decoding any of the encoded messages it can represent.

Given a string s consisting of digits and '*' characters, return the number of ways to decode it.

Since the answer may be very large, return it modulo 109 + 7.

1. 解题思路与分析 给定一个只含数字的长 n nn 的字符串 s ss，再给定一个对应规则，每个大写字母 ch 可以对应一个数字 ch - 'A' + 1。问该 s ss 有多少种不同的解码方式。s ss 中可能含有'*'，这个符号可以对应除了 0 00 以外的任意一位数。答案模 $10^9 + 7$ 后返回。

   思路是动态规划。设 f [i] f[i]f[i] 是 s ss 的长 i ii 的前缀的解码方式数，那么可以按照最后一位（或者两位）是解码成什么字母来分类进行累加。

```java
public int numDecodings(String t) {
    int mod = (int)1e9 + 7;
    int n = t.length();
    char [] s = t.toCharArray();
    System.out.println(Arrays.toString(s));
    int [] dp = new int [Math.max(2, n+1)];
    dp[0] = 1;
    dp[1] = s[0] == '*' ? 9 : s[0] == '0' ? 0 : 1;
    for (int i = 2; i <= n; i++) {
        System.out.println("i: " + i);
        for (int j = 1; j <= 26; j++) { // 枚举 s 的长 i 前缀的末尾可以解码为哪个大写字母
            char c = s[i-1];
            if (j <= 9) { // 如果是要解码为 A 到 I, 那么最后一个数字得单独解码
                if (c == '*' || c == '0' + j)
                    dp[i] += dp[i-1];
            } else {      // 否则最后两个数字得一起解码
                char p = s[i-2];
                int x = j % 10, y = j / 10;
                if ((p == '*' || p == y+ '0') && ((c == '*' && x != 0) || c == x + '0'))
                    dp[i] += dp[i-2];
            }
            dp[i] %= mod;
        }
    }
    return dp[n];
}
```

2. 解题思路与分析

   定义 dp[i] 是 nums 前 i 个字符可以得到的解码种数，假设之前的字符串是 abcx，现在新加入了 y，则有以下 5 种情况：

```
如果 x=='0'，且 y=='0'，无法解码，返回 0；
如果只有 x=='0'，则 y 只能单独放在最后，不能与 x 合并(不能以 0 开头)，此时有：dp[i] = dp[i-1]
如果只有 y=='0'，则 y 不能单独放置，必须与 x 合并，并且如果合并结果大于 26，返回 0，否则有：dp[i] = dp[i-2]
```

```
如果 xy<=26: 则 y 可以“单独”放在 abcx 的每个解码结果之后后，并且如果 abcx 以 x 单独结尾，此时可以合并 xy 作为结尾，而这种解码种数就是 abc 的解码结果，此
如果 xy>26: 此时 x 又不能与 y 合并，y 只能单独放在 dp[i]的每一种情况的最后，此时有：dp[i+1] = dp[i]
```

```java
    public int numDecodings(String s) {
        char[] arr = s.toCharArray();
        int[] dp = new int[s.length()+1];
        dp[0] = 1;
        dp[1] = arr[0]=='0'?0:1;
        if(s.length()<=1) return dp[1];
        for(int i=2;i<=s.length();i++){
            int n = (arr[i-2]-'0')*10+(arr[i-1]-'0');
            if(arr[i-1]=='0' && arr[i-2]=='0'){
                return 0;
            }else if(arr[i-2]=='0'){
                dp[i] = dp[i-1];
            }else if(arr[i-1]=='0'){
                if(n>26) return 0;
                dp[i] = dp[i-2];
            }else if(n>26){
                dp[i] = dp[i-1];
            }else{
                dp[i] = dp[i-1]+dp[i-2];
            }
        }
        return dp[dp.length-1];
    }
```

1.5.56 629. K Inverse Pairs Array - Hard

For an integer array nums, an inverse pair is a pair of integers [i, j] where 0 <= i < j < nums.length and nums[i] > nums[j].

Given two integers n and k, return the number of different arrays consist of numbers from 1 to n such that there are exactly k inverse pairs. Since the answer can be huge, return it modulo 109 + 7.

1. 解题思路与分析 比较容易辨别出来是一道 DP 的题目，但是确实算是比较难的了，下面是参考网上的代码之后我的理解。定义 dp[n][k] 表示从 1 到 n 构成的数中含有 k 个逆序对的个数，则我们可以推导出 dp[n][k] 和 dp[n - 1][i] 之间的递推关系：

   如果我们把 n 放在最后一位，则所有的 k 个逆序对均来自于前 n - 1 个数所构成的逆序对，和 n 无关；

   如果我们把 n 放在倒数第二位，则有 1 个逆序对和 n 有关，有 k - 1 个逆序对来自前 n - 1 个数所构成的逆序对；

   ……

   如果我们把 n 放在第一位，则有 n-1 个逆序对和 n 有关，k - (n - 1) 个逆序对来自前 n - 1 个数所构成的逆序对。

   所以：dp[n][k] = dp[n-1][k]+dp[n-1][k-1]+dp[n-1][k-2]+···+dp[n-1][k+1-n+1]+dp[n-1][k-n+1]。但问题是 k - (n - 1) 有可能为负数，也就是说根据 n 和 k 的不同，上面的式子有可能从某个项之后就不合法了，我们这里先写出来占位，从而得到下面两个式子：

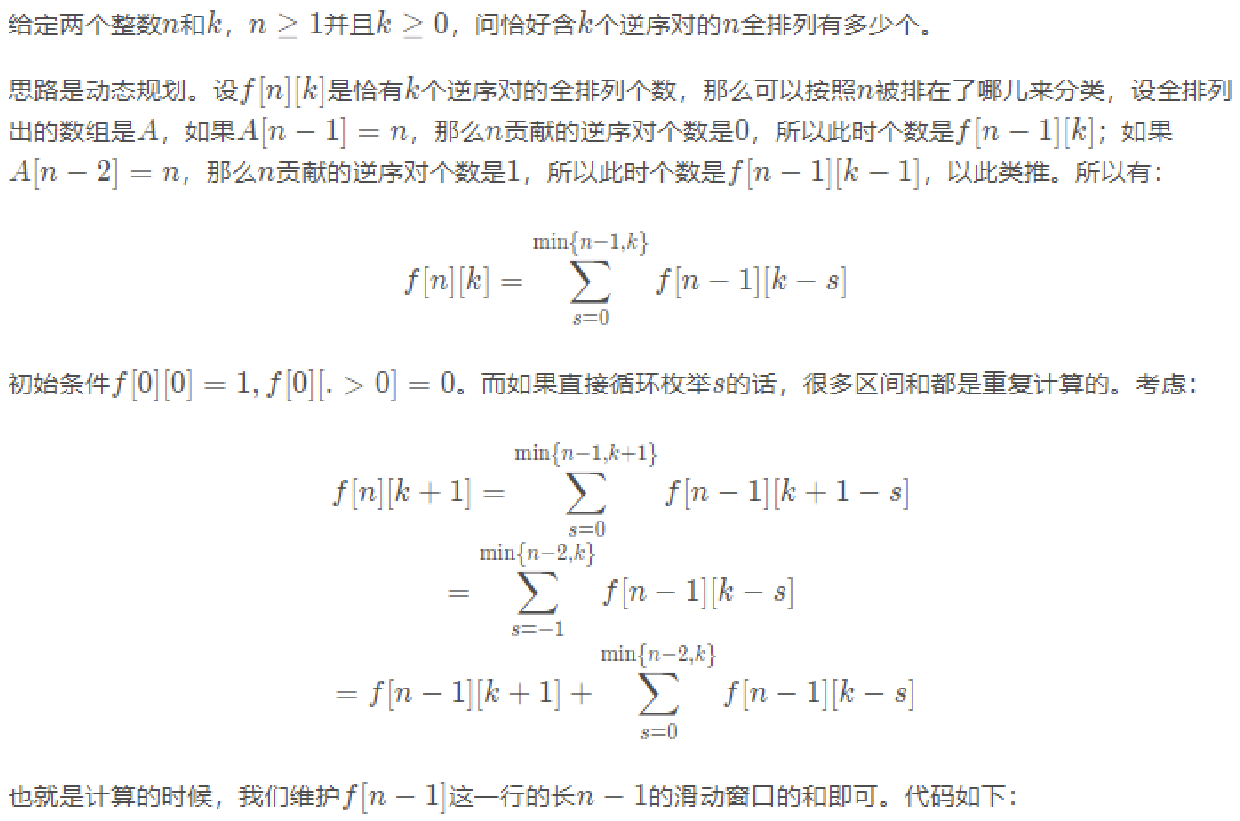
```
dp[n][k]     = dp[n-1][k] + dp[n-1][k-1] + dp[n-1][k-2] + ··· + dp[n-1][k + 1-n + 1] + dp[n-1][k-n + 1] // A
dp[n][k + 1] = dp[n-1][k + 1] + dp[n-1][k] + dp[n-1][k-1] + dp[n-1][k-2] + ··· + dp[n-1][k + 1-n + 1]   // B
dp[n][k+1] - dp[n][k] = dp[n-1][k+1] - dp[n-1][k-n+1]; // B-A
dp[n][k+1] = dp[n][k] + dp[n-1][k+1] - dp[n-1][k-n+1]; // 移项
// 将 k+1 换回成 k, 可以得到:
dp[n][k] = dp[n][k-1] + dp[n - 1][k] - dp[n-1][k-n]
```

   把上面两个式子相减可以推导出：dp[n][k+1] = dp[n][k]+dp[n-1][k+1]-dp[n-1][k+1-n]。这样就可以写出代码了。

   当然由于 dp[n][k] 只和 dp[n][x]，dp[n-1][x] 有关，所以该代码还可以进一步将空间复杂度从 O(nk) 降低到 O(k)。时间复杂度是 O(nk)。

```java
public int kInversePairs(int n, int k) {
    int mod = (int)1e9 + 7;
    long [][] dp = new long [n+1][k+1];
    dp[0][0] = 1;
    for (int i = 1; i <= n; i++) {
        dp[i][0] = 1;
        for (int j = 1; j <= k; j++)  {
            dp[i][j] = dp[i][j-1] + dp[i-1][j];
            if (j >= i)
                dp[i][j] -= dp[i-1][j-i];
            dp[i][j] = (dp[i][j] + mod) % mod;
        }
    }
    return (int)dp[n][k];
}
```

2. 解题思路与分析



给定两个整数n和k，$n \geq 1$并且$k \geq 0$，问恰好含k个逆序对的n全排列有多少个。

思路是动态规划。设$f[n][k]$是恰有k个逆序对的全排列个数，那么可以按照n被排在了哪儿来分类，设全排列出的数组是A，如果$A[n-1]=n$，那么n贡献的逆序对个数是0，所以此时个数是$f[n-1][k]$；如果$A[n-2]=n$，那么n贡献的逆序对个数是1，所以此时个数是$f[n-1][k-1]$，以此类推。所以有：

$$f[n][k] = \sum_{s=0}^{\min\{n-1,k\}} f[n-1][k-s]$$

初始条件$f[0][0]=1, f[0][.>0]=0$。而如果直接循环枚举s的话，很多区间和都是重复计算的。考虑：

$$\begin{aligned} f[n][k+1] &= \sum_{s=0}^{\min\{n-1,k+1\}} f[n-1][k+1-s] \\ &= \sum_{s=-1}^{\min\{n-2,k\}} f[n-1][k-s] \\ &= f[n-1][k+1] + \sum_{s=0}^{\min\{n-2,k\}} f[n-1][k-s] \end{aligned}$$

也就是计算的时候，我们维护$f[n-1]$这一行的长$n-1$的滑动窗口的和即可。代码如下：

```
public int kInversePairs(int n, int k) {
    int mod = (int)1e9 + 7;
    int [][] dp = new int [n+1][k+1];
    dp[0][0] = 1;
    for (int i = 1; i <= n; i++) {
        long sum = 0;
        for (int j = 0; j <= k; j++) {
            sum += dp[i-1][j];
            if (j >= i)
                sum -= dp[i-1][j-i];
            dp[i][j] = (int)(sum % mod);
        }
    }
    return (int)dp[n][k];
}
```

1.5.57 1787. Make the XOR of All Segments Equal to Zero - Hard

You are given an array nums and an integer k. The XOR of a segment [left, right] where left <= right is the XOR of all the elements with indices between left and right, inclusive: nums[left] XOR nums[left+1] XOR . . . XOR nums[right].

Return the minimum number of elements to change in the array such that the XOR of all segments of size k is equal to zero.

1. 解题思路与分析

解题思路：

容易发现，最后得到的数组一定满足：

- $a_1 = a_{k+1} = a_{2k+1} = \dots$
- $a_2 = a_{k+2} = a_{2k+2} = \dots$
- $\dots$

因此可以考虑将同一组中的数放在一起进行考虑。

首先，我们对每一组分别进行计数。

接下来，我们用动态规划求解。$dp[val]$表示处理到当前组时，异或值为val的最小修改次数。显然初值（考虑第一组之前）为：

- $dp[0] = 0$
- $dp[val] = \infty \ (val \neq 0)$

在转移时，我们有两种选择：

1. 将当前组的数全都改为同一个值，此时不管我们之前如何选择，都可以利用这一个值的选择得到任何一个异或值，那么我们自然应该选择之前的代价中最小的那个，此时的代价为$size[i] + \min dp[val]$，新状态的异或值可以为$[0, 2^D)$中的任意一个值。这一转移需要进行K次。
2. 使用当前组中的某一个值val'。此时我们需要枚举之前的异或值val（或枚举达到的目标值，二者是等价的），此时的代价为$size[i] - cnt[i][val'] + dp[val]$，新状态的异或值为$val \oplus val'$。这一转移最多需要进行$N$次。

- 时间复杂度$\mathcal{O}(2^D \cdot (K + N))$。其中$D = 10$。
- 空间复杂度$\mathcal{O}(N + 2^D)$。

- 根据题目特点，最后所有长度为 k 的区间异或结果等于零，可推出得到的数组满足：

```
a1 = ak+1 = a2k+1 = ⋯
a2 = ak+2 = a2k+2 = ⋯
```

因此可将数组中的元素按上述规律，每间隔 k 个的数字为一组进行分组。

在此基础上，设计一个动态规划数组 dp[j]，表示到当前第 i 组为止，所有元素异或到对应数字 j 时的更改次数。则对第 i 组 dp[j] 的状态转移方程可能为：

j 可由某一数值和当前组中的某个数 num 异或得到，newDp[j] = dp[j & num] + size[i] - 组中 num 的数量

j 可通过和任意数字异或得到，newDp[j] = 前一 dp 中最小的改变次数 + size[i]

完成 k 个组的动态规划后，dp[1]就是所求的解。

```java
public int minChanges(int[] arr, int k) {
    int n = arr.length;
    List<Map<Integer, Integer>> group = new ArrayList<>(); // 存储 k 个组、各组中各个数字数量
    int [] size = new int [k];                             // 各组大小，它们会有可能不同吗？
    for (int i = 0; i < k; i++) {
        Map<Integer, Integer> map = new HashMap<>();
        for (int j = i; j < n; j += k) {
            map.put(arr[j], map.getOrDefault(arr[j], 0) + 1);
            size[i]++;
        }
        group.add(map);
    }
    int range = 1 << 10;          //  题中 nums[i] < 2^10，为的是遍历所有可能更改值，以取最小
    int [] dp = new int [range];  // 当前组异或到对应数字时的更改次数
    Arrays.fill(dp, Integer.MAX_VALUE);
    dp[0] = 0;
    for (int i = 0; i < k; i++) {
        int minVal = Arrays.stream(dp).min().getAsInt();
        int [] newDp = new int [range];       //  遍历每一个个组，暴搜、取最小
        Arrays.fill(newDp, minVal + size[i]); // 变为当前组中不存在数字的改变次数：之前的最小改变次数 + 当前组元素个数
        for (int j = 0; j < range; j++) {
            if (dp[j] == Integer.MAX_VALUE) continue; // 对第 0 组, 保证 dp[i] 是异或到 0 的最小改变数
            Iterator it = group.get(i).entrySet().iterator();
            while (it.hasNext()) {
                Map.Entry en = (Map.Entry)it.next();
                int num = (int)en.getKey(), v = (int)en.getValue();
                int xorNum = num ^ j;
                newDp[xorNum] = Math.min(newDp[xorNum], dp[j] + size[i] - v);
            }
        }
        dp = Arrays.copyOf(newDp, range); // 将遍历到当前组、累积较优解的 newDp 复制入全局最优解 dp 数组中
    }
    return dp[0];
}
```

1.5.58 1735. Count Ways to Make Array With Product - Hard 乘积为 K 的质因子数排列组合的总个数：分解质因子

You are given a 2D integer array, queries. For each queries[i], where queries[i] = [ni, ki], find the number of different ways you can place positive integers into an array of size ni such that the product of the integers is ki. As the number of ways may be too large, the answer to the ith query is the number of ways modulo 109 + 7.

Return an integer array answer where answer.length == queries.length, and answer[i] is the answer to the ith query.

1.5.59 解题思路与分析

题意分析

回顾题意：对于一个查询 (n, k) 而言，我们需要找出 n 个整数的**一个排列**，使得它们的乘积为 k。

首先对 l 做质因数分解。设其全部的质因子为 $p_1, p_2, ..., p_m$，则会有

$$k = p_1^{c1} \cdot p_2^{c2} ... \cdot p_m^{cm}$$

考虑质因子 p_1 ：整数 k 中共有 c_1 个质因子 p_1。那么，如果要找出乘积为 k 的 n 个正整数，那么这 n 个数中一共也应当有 c_1 个质因子 p_1。其中，每个数最少有 0 个p_1，最多有 c_1 个 p_1。这也就意味着，我们要将 c_1 个 p_1 在这 n 个正整数之间「分配」。

于是，问题转化成了下面的子问题形式：「给定一个正整数 c_1，找出所有长度为 n 的非负整数的排列，使得它们的加和为 c_1」。

转化后的子问题是一个动态规划类型的问题。设 $dp[i][j]$ 代表「给定一个正整数 j，满足长度为 i，且加和为 j」的非负整数排列的数量。为了求解 $dp[i][j]$，我们考虑排列中的第一个整数，它的取值范围为 $[0, j]$，若它取值为 k，则余下 $i - 1$ 个整数的加和需要为 $j - k$，对应的方案数目为 $dp[i - 1][j - k]$。因此，

$$dp[i][j] = \Sigma_{k=0}^{k=j} dp[i-1][j-k]$$

在解决完毕子问题之后，原问题就迎刃而解了。为了分配 c_1 个 p_1，共有 $dp[n][c_1]$ 种方法；为了分配 c_2 个 p_2，共有 $dp[n][c_2]$ 种方法。因此，总的方法数目为

$$\Pi_{k=1}^{k=m} dp[n][c_m]$$

算法细节

首先，我们需要遍历数组，找到所有 n, k 的最大值 $maxn, maxk$。此外，还要找出「加和」的最大值 $cmax$。根据前面的描述，$cmax$ 即为质因数数量的最大值。而由于质因数最小为 2，$maxk$ 最大为 10^4，因此质因数数量不会超过 $\log_2 10^4 < 15$。

在找出每个变量的上界之后，就可以在 $O(\text{maxn} \cdot maxc^2)$ 的时间内预处理 dp 数组。

随后，我们还要预先求出 $[2, maxn]$ 之间的全部质数，复杂度为 $O(maxn)$。

最后，我们就可以回答每个查询了。

```java
// 在找出每个变量的上界之后，就可以在 O(maxn}{maxc}^2) 的时间内预处理 dp 数组。
//     随后，我们还要预先求出 [2, maxn] 之间的全部质数，复杂度为 O(maxn)。
public int[] waysToFillArray(int[][] q) {
    int n = q.length;
    int mod = (int)1e9 + 7;
    int maxC = 15;          // 找出「加和」的最大值 cmax。根据前面的描述，cmax 即为质因数数量的最大值。
    int maxN = 0, maxK = 0; // 而由于质因数最小为 22，maxkmaxk 最大为 10^4 因此质因数数量不会超过 log_2 10^4 < 15
    for (int i = 0; i < n; i++) { // 需要遍历数组，找到所有 n, kn,k 的最大值 maxn,maxk
        maxN = Math.max(maxN, q[i][0]);
        maxK = Math.max(maxK, q[i][1]);
    }
    long [][] dp = new long [maxN + 1][maxC + 1]; // dp[i][j] 代表给定一个正整数 jj，满足长度为 ii，且加和为 jj 的非负整数排列的数量。
    for (int i = 1; i <= maxC; i++)
        dp[1][i] = 1;
    for (int i = 1; i <= maxN; i++)
        dp[i][0] = 1;
    for (int i = 2; i <= maxN; i++)
        for (int j = 1; j <= maxC; j++)
            for (int k = 0; k <= j; k++) { // 为了求解 dp[i][j]，我们考虑排列中的第一个整数，它的取值范围为 [0,j][0,j]，
                dp[i][j] += dp[i-1][j-k];  // 若它取值为 kk，则余下 i-1i1 个整数的加和需要为 j-kjk，对应的方案数目为 dp[i1][jk]
                dp[i][j] %= mod;
            }
    int [] isPrime = new int [maxK + 1]; // 分解乘积的质因子
    Arrays.fill(isPrime, 1);
    List<Integer> primes = new ArrayList<>();
    for (int i = 2; i <= maxK; i++) {
        if (isPrime[i] == 1)
            primes.add(i);
        for (int j = i*2; j <= i*i && j <= maxK; j += i) // 最大乘积为 maxK 的数组, 分解出小的质因子了, 那么凡是小质因子的乘积倍数的数都不是质数
            isPrime[j] = 0;
    }
    int [] ans = new int [n];
    for (int i = 0; i < n; i++) {
        int m = q[i][0], k = q[i][1];
        List<Integer> cs = new ArrayList<>(); // 乘积 k 的质因子表
        for (int p : primes) {
            if (p > k) break;
            int cnt = 0, left = k;
            while (left % p == 0) {
                left /= p;
                cnt++;
            }
            if (cnt > 0) cs.add(cnt); // 乘积 k 中各质因子的个数 (指数)
        }
        long res = 1;
        for (int c : cs) {
            res *= dp[m][c]; // 数组长度为 n, 数组和为质因子 c 的指数个数的所有可能的分布数合数, 各质因子之间个数之间相乘
            res %= mod;
        }
        ans[i] = (int)res;
    }
    return ans;
}
```

- 下面这种方法理解得还不是很透

```
public int[] waysToFillArray(int[][] queries) {
    int[] result = new int[queries.length];
    int resultIdx = 0;
    Combination combination = new Combination(10030, 20);
    for (int[] q : queries) {
        int n = q[0]; // 长度为 n
        int k = q[1]; // 乘积为 k
        long product = 1L;
        for (int power : getPrimeFactors(k).values()) {
            // power 个球, 分到 n 个位置, 每个位置可以为空
            // 等价于: (power+n) 个球, 分到 n 个位置, 每个位置不能为空
            // Why? 等价后得到一种分法, 每组减去 1, 就是原来的解
            // 插板法可得 C (power + n - 1, n - 1) = C (power + n - 1, power)
            product = (product * combination.get(n + power - 1, power)) % mod;
        }
        result[resultIdx++] = (int)(product);
    }
    return result;
}
long mod = (int)1e9 + 7;
public HashMap<Integer, Integer> getPrimeFactors(int n) {
    HashMap<Integer, Integer> map = new HashMap();
    for(int i = 2; i <= n; i++) {
        if (n % i == 0) {
            int cnt = 0;
            while (n % i == 0) {
                cnt++;
                n = n / i;
            }
            map.put(i, cnt);
        }
    }
    return map;
}
class Combination {
    long[][] c;
    Combination (int n, int m) {
        c = new long[n + 1][m + 1];
        c[0][0] = 1;
        for(int i = 1; i <= n; i++){
            c[i][0] = 1;
            for(int j = 1; j <= m; j++)
                c[i][j] = (c[i-1][j-1] + c[i-1][j]) % mod;
        }
    }
    public long get(int n, int m) {
        return c[n][m];
    }
}
```

1.5.60 1359. Count All Valid Pickup and Delivery Options - Hard

Given n orders, each order consist in pickup and delivery services.
Count all valid pickup/delivery possible sequences such that delivery(i) is always after of pickup(i).
Since the answer may be too large, return it modulo $10^9 + 7$.

1. 解题思路与分析

   就是总共有 2N 个位置，每次放一两个，还剩下多少个位置可以合理占用

```
public int countOrders(int n) {
    int mod = (int)1e9 + 7;
    int spots = n * 2;
    long ans = 1;
    for (int i = n; i >= 2; i--) {
        ans = (ans * spots * (spots - 1) / 2l) % mod;
        spots -= 2;
    }
    return (int)ans;
}
```

1.6 BitMask 掩码相关的

1.7 第二次刷 DP: 下面的题目只需要注意细节，思路没问题

1.7.1 322. Coin Change

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money. Return 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.

```
public int coinChange(int[] coins, int amount) {
    if (amount == 0) return 0;
    int n = coins.length;
    int [] dp = new int [amount + 1];
    Arrays.fill(dp, amount + 1);
    dp[0] = 0;
    for (int i = 0; i <= amount; i++) {
        for (int v : coins) {
            if (i - v < 0) continue;
            dp[i] = Math.min(dp[i], dp[i-v] + 1);
        }
    }
    return dp[amount] == amount + 1 ? -1 : dp[amount];
}
```