

LeetCode Online Coding Interview Questions – 错题集

deepwaterooo

March 31, 2022

Contents

Chapter 1

Bit Manipulations

1.1 基本概念：原码、反码、与补码（对负数尤其重要）

1.1.1 1、原码：

一个正数，按照绝对值大小转换成的二进制数；一个负数按照绝对值大小转换成的二进制数，然后最高位补 1，称为原码。

比如 00000000 00000000 00000000 00000101 是 5 的原码。

10000000 00000000 00000000 00000101 是 -5 的原码。

备注：

比如 byte 类型，用 2^8 来表示无符号整数的话，是 0 - 255 了；如果有符号，最高位表示符号，0 为正，1 为负，那么，正常的理解就是 -127 至 +127 了。这就是原码了，值得一提的是，原码的弱点，有 2 个 0，即 +0 和 -0（10000000 和 00000000）；还有就是，进行异号相加或同号相减时，比较笨蛋，先要判断 2 个数的绝对值大小，然后进行加减操作，最后运算结果的符号还要与大的符号相同；于是，反码产生了。

1.1.2 2、反码

正数的反码与原码相同，负数的反码为对该数的原码除符号位外各位取反 [每一位取反 (除符号位)]。

取反操作指：原为 1，得 0；原为 0，得 1。（1 变 0；0 变 1）

比如：正数 00000000 00000000 00000000 00000101 的反码还是 00000000 00000000 00000000 00000101

负数 10000000 00000000 00000000 00000101 的反码则是 11111111 11111111 11111111 11111010。

反码是相互的，所以也可称：10000000 00000000 00000000 00000101 和 11111111 11111111 11111111 11111010 互为反码。

备注：还是有 +0 和 -0，没过多久，反码就成为了过滤产物，也就是，后来补码出现了。

1.1.3 3、补码

正数的补码与原码相同，负数的补码为对该数的原码除符号位外各位取反，然后在最后一位加 1。

比如：10000000 00000000 00000000 00000101 的补码是：11111111 11111111 11111111 11111010。

那么，补码为：

11111111 11111111 11111111 11111010 + 1 = 11111111 11111111 11111111 11111011

备注：1、从补码求原码的方法跟原码求补码是一样的，也可以通过完全逆运算来做，先减一，再取反。

2、补码却规定 0 没有正负之分

所以，-5 在计算机中表达为：11111111 11111111 11111111 11111011。转换为十六进制：0xFFFFFFF5。

1.2 数组中不重复的两个元素

给定一个整数数组 `nums`，其中恰好有两个元素只出现一次，其余所有元素均出现两次。找出只出现一次的那两个元素。

输入：[1,2,1,3,2,5]

输出：[3,5]

复制代码将所有元素进行异或运算，得到两个不重复元素的异或值，也就是这两个元素中不相同的部分为 1 的数，`n & (-n)` 得到 `n` 的位级表示中最低的那一位 1，这个 1 只可能来自两个不重复元素中的一个（就算重复数的二进制数中也有可能包含这个 1，但通过 `x ^= num` 的异或操作便消除）。

isolate rightmost 1-bit which is different for x and y	bitmask	0	0	0	0	0	0	0
	x = 1	0	0	0	0	0	0	1
	y = 2	0	0	0	0	0	1	0
	a = 3	0	0	0	0	0	1	1
	a = 3	0	0	0	0	0	1	1
	bitmask = bitmask ^ x ^ y ^ a ^ a	0	0	0	0	0	1	1
	diff = bitmask & (-bitmask)	0	0	0	0	0	0	1

1.3 371. Sum of Two Integers

Given two integers a and b, return the sum of the two integers without using the operators + and

一开始自己想的如果两个数都是正数，那么很简单，运用 XOR ^ 去找出所有的单一的 1。接着运用 AND & 去找出所有重复的 1；重复的 1 就相当于 carryover，需要进位。然后运动 << 把重复的 1 给进位就可以了，最后直接 OR 一下就等于答案（这是错的，需要每次循环来判断新的进位）。但是发现这个是能运用于两个正数，研究来研究去，不会算带负数的，所以放弃网上找答案。

发现答案不和我的两个正数之和算法一样嘛！唯一不同的就是答案是把算出的答案继续带回 function 直到 carry 等于 0；

通过例子来看一下：

a = 5, b = 1:

a: 101

b: 001

根据我最初的算法：（错误的）

sum = a ^ b = 100

carry = a & b = 001 这里这个 1 就是需要进位的

carry = 001 << 1 = 010

最后把 sum 100 和 carry 010 OR 一下就等于 110 = 6。

但是答案的做法却是把 sum 和 carry 在带回 function 继续算直至 carry = 0，我们来看一下例子：

a = 5, b = 1:

a = 101

b = 001

sum = 100

carry = 010

带回

a = 100

b = 010

sum = 110

carry = 000 这里等于 0 了，所以结束，我的理解是，答案的做法是把 carryover 带回去，和 sum 比较，如果这一次没有继续需要进位的数字了，就可以结束，否则继续下一轮；换一句话就是，答案是把每一轮的 sum 和 carryover 拿出来，下一轮继续加一起看一看有没有新的需要进位的地方，所以明显我之前的做法是错的，我只考虑了一轮而已，实际上是每一轮都有可能新的需要进位的地方。

那新的问题又来了，为啥负数也可以，这里的负数是 2 's complement:

比如说 -5 = 1111 1111 1111 1111 1111 1111 1011

为何-5 是这样：首先把上面的 bits -1

1111 1111 1111 1111 1111 1111 1010


```
// Think about (a&b) ^ (a&c). Can you simplify this expression?
// It is equal to a&(b^c).
// Then, (A[i]&B[0])^(A[i]&B[1])... = A[i]&(B[0]^B[1]^arr[2]...).
// Let bXorSum = (B[0]^B[1]^B[2]...),
// aXorSum = (A[0]^A[1]^A[2]...) so the final answer is
// (bXorSum&A[0]) ^ (bXorSum&A[1]) ^ (bXorSum&A[2]) ^ ... = bXorSum & aXorSum.
public int getXORSum(int[] a, int[] b) {
    int m = a.length;
    int n = b.length;
    int aXorSum = a[0], bXorSum = b[0];
    for (int i = 1; i < m; i++)
        aXorSum ^= a[i];
    for (int i = 1; i < n; i++)
        bXorSum ^= b[i];
    return aXorSum & bXorSum;
}
```

1.3.3 982. Triples with Bitwise AND Equal To Zero 平生不识 TwoSum，刷尽 LeetCode 也枉然

Given an integer array nums, return the number of AND triples.

An AND triple is a triple of indices (i, j, k) such that:

$0 \leq i < \text{nums.length}$ $0 \leq j < \text{nums.length}$ $0 \leq k < \text{nums.length}$ $\text{nums}[i] \& \text{nums}[j] \& \text{nums}[k] == 0$, where $\&$ represents the bitwise-AND operator.

```
// ‘平生不识 TwoSum，刷尽 LeetCode 也枉然’ 还好不至于哭死呀.....
public int countTriplets(int[] arr) {
    Map<Integer, Integer> m = new HashMap<>();
    int v = 0, res = 0;
    for (int i = 0; i < arr.length; i++)
        for (int j = 0; j < arr.length; j++) {
            v = arr[i] & arr[j];
            m.put(v, m.getOrDefault(v, 0) + 1);
        }
    for (int i = 0; i < arr.length; i++)
        for (int k : m.keySet())
            if ((arr[i] & k) == 0) res += m.get(k);
    return res;
}

public int countTriplets(int[] arr) { // 这种方法执行起来效率更高一点儿
    int res = 0, v = 0;
    int[] cnt = new int[1 << 16];
    Arrays.fill(cnt, -1);
    for (int a : arr)
        for (int b : arr) {
            v = a & b;
            if (cnt[v] == -1) {
                cnt[v] = 0;
                for (int c : arr)
                    if ((v & c) == 0) ++cnt[v];
            }
            res += cnt[v];
        }
    return res;
}
```

1.3.4 187. Repeated DNA Sequences - Medium

The DNA sequence is composed of a series of nucleotides abbreviated as 'A', 'C', 'G', and 'T'.

For example, "ACGAATTCCG" is a DNA sequence. When studying DNA, it is useful to identify repeated sequences within the DNA.

Given a string s that represents a DNA sequence, return all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule. You may return the answer in any order.

有人说上面 native 方法超时是因为字符串存储浪费了太多的空间和时间，因此可以考虑用整数存储，即二进制方法。这个思路非常简单，这里一共有四个字母：A，C，G，T。我们转换整数的思路如下：

```
A = 00, C = 01, G = 10, T = 11
int key = 0, key = key << 2 | code(A|C|G|T)
```

这样我们就很容易把一个字符串转换为整数了，上面公式不清楚的话，可以直接看转换代码：

```
private static int hashCode(String s) {
    int hash = 0;
    for (int i = 0; i < s.length(); i++)
        hash = hash << 2 | mapInteger(s.charAt(i));
    return hash;
}

private static int mapInteger(char c) {
    switch (c) {

```

```

        case 'A': return 0;
        case 'C': return 1;
        case 'G': return 2;
        case 'T': return 3;
        default: return 0;
    }
}

public List<String> findRepeatedDnaSequences(String s) {
    List<String> res = new ArrayList<>();
    if (s == null || s.length() == 0) return res;
    Set<Integer> si = new HashSet<>();
    for (int i = 0; i <= s.length()-10; i++) {
        String substr = s.substring(i, i+10);
        Integer key = hashCode(substr);
        if (si.contains(key) && !res.contains(substr))
            res.add(substr);
        else si.add(key);
    }
    return res;
}

```

1.3.5 1915. Number of Wonderful Substrings - Medium

A wonderful string is a string where at most one letter appears an odd number of times.

For example, "ccjjc" and "abab" are wonderful, but "ab" is not. Given a string word that consists of the first ten lowercase English letters ('a' through 'j'), return the number of wonderful non-empty substrings in word. If the same substring appears multiple times in word, then count each occurrence separately.

A substring is a contiguous sequence of characters in a string.

```

public long wonderfulSubstrings(String word) {
    int n = word.length(), mask = 0, cur = 0;
    long res = 0, cnt = 0;
    Map<Integer, Integer> m = new HashMap<>();
    m.put(0, 1);
    for (int i = 0; i < n; i++) {
        mask ^= (1 << (word.charAt(i) - 'a'));
        res += m.getOrDefault(mask, 0);
        m.put(mask, m.getOrDefault(mask, 0) + 1);
        for (int j = 0; j < 10; j++) {
            cur = mask ^ (1 << j);
            res += m.getOrDefault(cur, 0);
        }
    }
    return res;
}

```

1.3.6 782. Transform to Chessboard- Hard

You are given an $n \times n$ binary grid board. In each move, you can swap any two rows with each other, or any two columns with each other.

Return the minimum number of moves to transform the board into a chessboard board. If the task is impossible, return -1.

A chessboard board is a board where no 0's and no 1's are 4-directionally adjacent.

我们发现对于长度为奇数的棋盘，各行的 0 和 1 个数不同，但是还是有规律的，每行的 1 的个数要么为 $n/2$ ，要么为 $(n+1)/2$ ，这个规律一定要保证，不然无法形成棋盘。

还有一个很重要的规律，我们观察题目给的第一个例子，如果我们只看行，我们发现只有两种情况 0110 和 1001，如果只看列，只有 0011 和 1100，我们发现不管棋盘有多长，都只有两种情况，而这两种情况下各位上是相反的，只有这样的矩阵才有可能转换为棋盘。那么这个规律可以衍生出一个规律，就是任意一个矩形的四个顶点只有三种情况，要么四个 0，要么四个 1，要么两个 0 两个 1，不会有其他的情况。那么四个顶点亦或在一起一定是 0，所以我们判断只要亦或出了 1，一定是不对的，直接返回 -1。之后我们来统计首行和首列中的 1 个数，因为我们要让其满足之前提到的规律。统计完了首行首列 1 的个数，我们判断如果其小于 $n/2$ 或者大于 $(n+1)/2$ ，那么一定无法转为棋盘。我们还需要算下首行和首列跟棋盘位置的错位的个数，虽然 01010 和 10101 都可以是正确的棋盘，我们先默认跟 10101 比较好了，之后再做优化处理。

最后的难点就是计算最小的交换步数了，这里要分 n 的奇偶来讨论。如果 n 是奇数，我们必须得到偶数个，为啥呢，因为我们之前统计的是跟棋盘位置的错位的个数，而每次交换行或者列，会修改两个错位，所以如果是奇数就无法还原为棋盘。举个例子，比如首行是 10001，如果我们跟棋盘 10101 比较，只有一个错位，但是我们是无法通过交换得到 10101 的，所以我们必须要交换得到 01010，此时的错位是 4 个，而我们通过 $n - \text{rowDiff}$ 正好也能得到 4，这就是为啥我们需要偶数个错位。如果 n 是偶数，那么就不会出现这种问题，但是会出现另一个问题，比如我们是 0101，这本身就是正确的棋盘排列了，但是由于我们默认是跟 1010 比较，那么我们会得到 4 个错位，所以我们应该跟 $n - \text{rowDiff}$ 比较取较小值。列的处理跟行的处理完全一样。最终我们把行错位个数跟列错位个数相加，再除以 2，就可以得到最小的交换次数了，之前说过了每交换一次，可以修复两个错位，参见代码如下：

```

public int movesToChessboard(int[][] bd) { // bd: board
    int n = bd.length, rowSum = 0, colSum = 0, rowDiff = 0, colDiff = 0;

```

```

for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        if ((bd[0][0] ^ bd[i][0] ^ bd[0][j] ^ bd[i][j]) > 0) return -1;
for (int i = 0; i < n; i++) {
    rowSum += bd[0][i];
    colSum += bd[i][0];
    rowDif += bd[i][0] == i % 2 ? 1 : 0; //
    colDif += bd[0][i] == i % 2 ? 1 : 0; //
}
if (rowSum < n/2 || rowSum > (n+1)/2) return -1;
if (colSum < n/2 || colSum > (n+1)/2) return -1;
if (n % 2 == 1) {
    if (rowDif % 2 == 1) rowDif = n - rowDif;
    if (colDif % 2 == 1) colDif = n - colDif;
} else {
    rowDif = Math.min(rowDif, n - rowDif);
    colDif = Math.min(colDif, n - colDif);
}
return (rowDif + colDif) / 2;
}

```

- 方法一：分维度计算【通过】

思路

首先需要思考的是一次交换之后，棋盘会发生什么变化。为了简单起见，这里用交换列来做例子。在对任意两列进行交换之后，可以看到列交换是不会改变任意两行之间的状态的，简单的来说如果这两行原本就相同，列交换之后这两行依旧相同，如果这两行本来就不同，列交换之后也还是不同。由于最终的棋盘只有两种不同的行，最初的棋盘也一定只有两种不同的行，否则不管怎么做列交换都不会得到最终的棋盘。

之后再来看棋盘行的规律，棋盘有两种行，这两种行每一位都互相不同。同时对于每一行来说，一定有一半为 1，一半为 0（如果长度为奇数，会多一个 1 或多一个 0）。对于棋盘的列也是同样的规律。

可以观察到，先换行再换列跟先换列再换行结果是一样的。在这里先将所有的行调到正确的位置，再将所有的列调到正确的位置。

考虑到只有两种不同的行，可以分别用 0, 1 对其表示。要达成最终的棋盘实际上等价于将棋盘的行表示成 0, 1 相隔的状态。假设在将棋盘的行用 0, 1 表示之后得到数组为 [0, 1, 1, 1, 0, 0]，那么只需求这个数组变成 [0, 1, 0, 1, 0, 1] 和 [1, 0, 1, 0, 1, 0] 的代价，之后取其中最小的代价就好了。同理，对列也是如此，这就将二维问题变成了两个一维问题。

算法

首先需要确认是否有且只有两种行（列）存在，且这两种行（列）的 0, 1 排布合法，如果不符合条件直接返回 -1。之后需要生成理想的行（列）的状态（即 0, 1 相隔的数组排列），对于每种理想状态，计算其与初始状态之间变换的代价。举个例子，对于 [0, 1, 1, 1, 0, 0] 初始状态来说，有两种理想状态，分别是 [0, 1, 0, 1, 0, 1] 和 [1, 0, 1, 0, 1, 0]，对于 [0, 1, 1, 1, 0] 初始状态只有一种理想状态 [1, 0, 1, 0, 1]。

在 Java 实现中，用整型来表示每行。之后将其与 0b010101010101....01 进行异或来计算初始状态转换到理想状态的代价。为了代码简洁，这里统一使用 0xAAAAAAAA 和 0x55555555，为了不引入额外的转换代价，还需要根据行的长度 N 生成 0b00...0011...11 掩码与结果做与运算。

```

public int movesToChessboard(int[][] board) {
    int N = board.length;
    // count[code] = v, where code is an integer
    // that represents the row in binary, and v
    // is the number of occurrences of that row
    Map<Integer, Integer> count = new HashMap();
    for (int[] row: board) {
        int code = 0;
        for (int x: row)
            code = 2 * code + x;
        count.put(code, count.getOrDefault(code, 0) + 1);
    }
    int k1 = analyzeCount(count, N);
    if (k1 == -1) return -1;
    // count[code], as before except with columns
    count = new HashMap();
    for (int c = 0; c < N; ++c) {
        int code = 0;
        for (int r = 0; r < N; ++r)
            code = 2 * code + board[r][c];
        count.put(code, count.getOrDefault(code, 0) + 1);
    }
    int k2 = analyzeCount(count, N);
    return k2 >= 0 ? k1 + k2 : -1;
}

public int analyzeCount(Map<Integer, Integer> count, int N) {
    // Return -1 if count is invalid
    // Otherwise, return number of swaps required
    if (count.size() != 2) return -1;
    List<Integer> keys = new ArrayList(count.keySet());
    int k1 = keys.get(0), k2 = keys.get(1);
    // If lines aren't in the right quantity
    if (!(count.get(k1) == N/2 && count.get(k2) == (N+1)/2) &&
        !(count.get(k2) == N/2 && count.get(k1) == (N+1)/2))

```

```

    return -1;
    // If lines aren't opposite
    if ((k1 ^ k2) != (1 << N) - 1)
        return -1;
    int Nones = (1 << N) - 1;
    int ones = Integer.bitCount(k1 & Nones); // bitCount 统计二进制中 1 的个数
    int cand = Integer.MAX_VALUE;
    if (N%2 == 0 || ones * 2 < N) // zero start
        cand = Math.min(cand, Integer.bitCount(k1 ^ 0xAAAAAAAA & Nones) / 2);
    if (N%2 == 0 || ones * 2 > N) // ones start
        cand = Math.min(cand, Integer.bitCount(k1 ^ 0x55555555 & Nones) / 2);
    return cand;
}

```

1.3.7 1803. Count Pairs With XOR in a Range - Hard

Given a (0-indexed) integer array `nums` and two integers `low` and `high`, return the number of nice pairs.

A nice pair is a pair (i, j) where $0 \leq i < j < \text{nums.length}$ and $\text{low} \leq (\text{nums}[i] \text{ XOR } \text{nums}[j]) \leq \text{high}$.

• 1. 暴力算法

直接暴力计算，利用 $\text{num}^{\text{num2}} = i$ 等效于 $\text{num}^i = \text{num2}$ 的特点，先统计当前各个数字出现的次数，再将当前数字和 $[\text{low}, \text{high}]$ 范围内的数字进行异或运算，将其结果对应的出现次数相加。

```

public int countPairs(int[] arr, int low, int high) {
    int[] freq = new int[20002]; // nums[i] <= 20000
    for (int v : arr)
        freq[v]++;
    int val = 0, res = 0;
    for (int v : arr) {
        for (int i = low; i <= high; i++) {
            val = v ^ i; // num ^ i = num2 <==> num ^ num2 = i
            if (val <= 20000)
                res += freq[val]; // v^val=i 中当前 v 对应 val 出现的次数
        }
        freq[v]--; // 当前 v 所有组合已统计，减去出现次数，避免重复
    }
    return res;
}

```

• 2. 字典树 (Trie 树)

(1) 思路在上述算法的基础上，结合字典树方法快速统计。在依次将 `nums` 中数字加入字典树的同时，搜索和该数字异或值在 $[0, \text{high}]$ 和 $[0, \text{low} - 1]$ 范围内数字 `num2` 的个数并相减，就是符合异或值为 $[\text{low}, \text{high}]$ 区间内的数字个数。

```

public class Trie {
    static final int H = 14; // 2^15=32768, 15 位二进制足够计算
    int cnt;
    Trie[] next;
    public Trie() {
        this.cnt = 0;
        this.next = new Trie[2];
    }
    public void insert(int va) { // 插入数值
        Trie r = this;
        for (int i = H; i >= 0; i--) {
            int bit = (va >> i) & 1;
            if (r.next[bit] == null)
                r.next[bit] = new Trie();
            r = r.next[bit];
            r.cnt++;
        }
    }
    public int search(Trie r, int digit, int v, int range) { // 搜索和 v 异或值在 [0, range] 范围内的数字 num2 的个数
        if (r == null) return 0;
        if (digit < 0) return r.cnt;
        int vb = (v >> digit) & 1; // v 和 range 在该位的值
        int vr = (range >> digit) & 1;
        if (vr == 1) { // range 在该位为 1
            if (vb == 0) // num 在该位为 0, num2 该位为 0 的部分全部满足，为 1 的部分继续判断
                return r.next[0] == null ? search(r.next[1], digit-1, v, range) : r.next[0].cnt + search(r.next[1], digit-1, v, range);
            else // v 在该位为 1, num2 该位为 1 的部分全部满足，为 0 的部分继续判断
                return (r.next[1] == null ? search(r.next[0], digit-1, v, range) : r.next[1].cnt + search(r.next[0], digit-1, v, range));
        }
        return search(r.next[vb], digit-1, v, range); // range 在该位 vr 为 0, num2 该位必须和 num 一致
    }
}

private Trie root;
public int countPairs(int[] arr, int low, int high) {
    int n = arr.length, maxHeight = 14; // 2^15=32768, 15 位二进制足够计算
    int res = 0;
    root = new Trie();
    for (int v : arr) {

```

```

    res += root.search(root, maxHeight, v, high) - root.search(root, maxHeight, v, low-1); // 这里的脑袋好难转呀...
    root.insert(v);
}
return res;
// for (int v : arr)
//     root.insert(v);
// for (int v : arr)
//     res += root.search(root, maxHeight, v, high) - root.search(root, maxHeight, v, low-1);
// return res / 2; // 如果按这种写法, 就得 / 2, 智商呢 ?!!
}

```

1.3.8 1734. Decode XORed Permutation - Medium

There is an integer array perm that is a permutation of the first n positive integers, where n is always odd.

It was encoded into another integer array encoded of length n - 1, such that encoded[i] = perm[i] XOR perm[i + 1]. For example, if perm = [1,3,2], then encoded = [2,1].

Given the encoded array, return the original array perm. It is guaranteed that the answer exists and is unique.

结合 n 为奇数的特点, 先对 encoded 数组中下标为奇数的元素进行异或, 得到第 2 到 n 个数的异或值;

因为整数数组是前 n 个正整数的排列, 再对 1 到 n 进行异或, 得到全部数的异或值;

上述二者进行异或即可得到第 1 个数, 然后依次求解获得其他数字, 得到原始数组。

```

public int[] decode(int[] encoded) {
    int n = encoded.length + 1;
    int xor = 0, vFrom2 = 0;
    for (int i = 1; i < n-1; i += 2) // 记录第 2 到 n 个数的异或值
        vFrom2 = vFrom2 ^ encoded[i]; // (a[1]^a[2])^(a[3]^a[4])^...^(a[n-2]^a[n-1])
    for (int i = 1; i <= n; i++) // a[0]^a[1]^a[2]^...^a[n-1]
        xor ^= i;
    int [] arr = new int [n];
    arr[0] = xor ^ vFrom2;
    for (int i = 1; i < n; i++)
        arr[i] = arr[i-1] ^ encoded[i-1];
    return arr;
}

```

1.3.9 957. Prison Cells After N Days - Medium

There are 8 prison cells in a row and each cell is either occupied or vacant.

Each day, whether the cell is occupied or vacant changes according to the following rules:

If a cell has two adjacent neighbors that are both occupied or both vacant, then the cell becomes occupied. Otherwise, it becomes vacant. Note that because the prison is a row, the first and the last cells in the row can't have two adjacent neighbors.

You are given an integer array cells where cells[i] = 1 if the ith cell is occupied and cells[i] = 0 if the ith cell is vacant, and you are given an integer n.

Return the state of the prison after n days (i.e., n such changes described above).

```

Input: cells = [0,1,0,1,1,0,0,1], N = 7
Output: [0,0,1,1,0,0,0,0]
Explanation: The following table summarizes the state of the prison on each day:
Day 0: [0, 1, 0, 1, 1, 0, 0, 1]
Day 1: [0, 1, 1, 0, 0, 0, 0, 0]
Day 2: [0, 0, 0, 0, 1, 1, 1, 0]
Day 3: [0, 1, 1, 0, 0, 1, 0, 0]
Day 4: [0, 0, 0, 0, 0, 1, 0, 0]
Day 5: [0, 1, 1, 1, 0, 1, 0, 0]
Day 6: [0, 0, 1, 0, 1, 1, 0, 0]
Day 7: [0, 0, 1, 1, 0, 0, 0, 0]

```

博主最开始做的时候, 看题目标记的是 Medium, 心想应该不需要啥特别的技巧, 于是就写了一个暴力破解的, 但是超时了 Time Limit Exceeded. 给了一个超级大的 N, 不得不让博主怀疑是否能够直接遍历 N, 又看到了本题的标签是 Hash Table, 说明了数组的状态可能是会有重复的, 就是说可能是有一个周期循环的, 这样就完全没有必要每次都算一遍。正确的做法的应该是建立状态和当前 N 值的映射, 一旦当前计算出的状态在 HashMap 中出现了, 说明周期找到了, 这样就可以通过取余来快速的缩小 N 值。为了使用 HashMap 而不是 TreeMap, 这里首先将数组变为字符串, 然后开始循环 N, 将当前状态映射为 N-1, 然后新建了一个长度为 8, 且都是 0 的字符串。更新的时候不用考虑首尾两个位置, 因为前面说了, 首尾两个位置一定会变为 0。更新完成了后, 便在 HashMap 查找这个状态是否出现过, 是的话算出周期, 然后 N 对周期取余。最后再把状态字符串转为数组即可, 参见代码如下:

```

vector<int> prisonAfterNDays(vector<int>& cells, int N) {
    vector<int> res;
    string str;
    for (int num : cells) str += to_string(num);
    unordered_map<string, int> m;
    while (N > 0) {
        m[str] = N--;
        string cur(8, '0');

```

```

    for (int i = 1; i < 7; ++i) {
        cur[i] = (str[i - 1] == str[i + 1]) ? '1' : '0';
    }
    str = cur;
    if (m.count(str)) {
        N %= m[str] - N;
    }
}
for (char c : str) res.push_back(c - '0');
return res;
}

```

下面的解法使用了 TreeMap 来建立状态数组和当前 N 值的映射，这样就不用转为字符串了，写法是简单了一点，但是运行速度下降了许多，不过还是在 OJ 许可的范围之内，参见代码如下：

```

vector<int> prisonAfterNDays(vector<int>& cells, int N) {
    map<vector<int>, int> m;
    while (N > 0) {
        m[cells] = N--;
        vector<int> cur(8);
        for (int i = 1; i < 7; ++i) {
            cur[i] = (cells[i - 1] == cells[i + 1]) ? 1 : 0;
        }
        cells = cur;
        if (m.count(cells)) {
            N %= m[cells] - N;
        }
    }
    return cells;
}

```

- 下面这种解法是看 lee215 大神的帖子中说的这个循环周期是 1, 7, 或者 14, 知道了这个规律后，直接可以在开头就对 N 进行缩小处理，取最大的周期 14，使用 $(N-1) \% 14 + 1$ 的方法进行缩小，至于为啥不能直接对 14 取余，是因为首尾可能会初始化为 1，而一旦 N 大于 0 的时候，返回的状态首尾一定是 0。为了不使得正好是 14 的倍数的 N 直接缩小为 0，所以使用了这么个小技巧，参见代码如下：

```

vector<int> prisonAfterNDays(vector<int>& cells, int N) {
    for (N = (N - 1) % 14 + 1; N > 0; --N) {
        vector<int> cur(8);
        for (int i = 1; i < 7; ++i) {
            cur[i] = (cells[i - 1] == cells[i + 1]) ? 1 : 0;
        }
        cells = cur;
    }
    return cells;
}

public int[] prisonAfterNDays(int[] arr, int n) {
    int m = 8, cnt = 0;
    int [] tmp = arr.clone();
    while (cnt < (n % 14 == 0 ? 14 : n % 14)) {
        Arrays.fill(tmp, 0);
        for (int i = 1; i < m-1; i++)
            tmp[i] = 1 - (arr[i-1] ^ arr[i+1]);
        arr = tmp.clone();
        ++cnt;
    }
    return arr;
}

```

- 还有一个大神级的思路

since N might be pretty large, so we can't starting from times 1 to times N, No matter what the rules are, the states might be reappear after a certain times of proceeding(because we have fixed number of different states.)

but for different initial state, it might take different steps to reach back to this same state.

so we need to calculate the length of that. and based on N, we can get what we want after N steps.

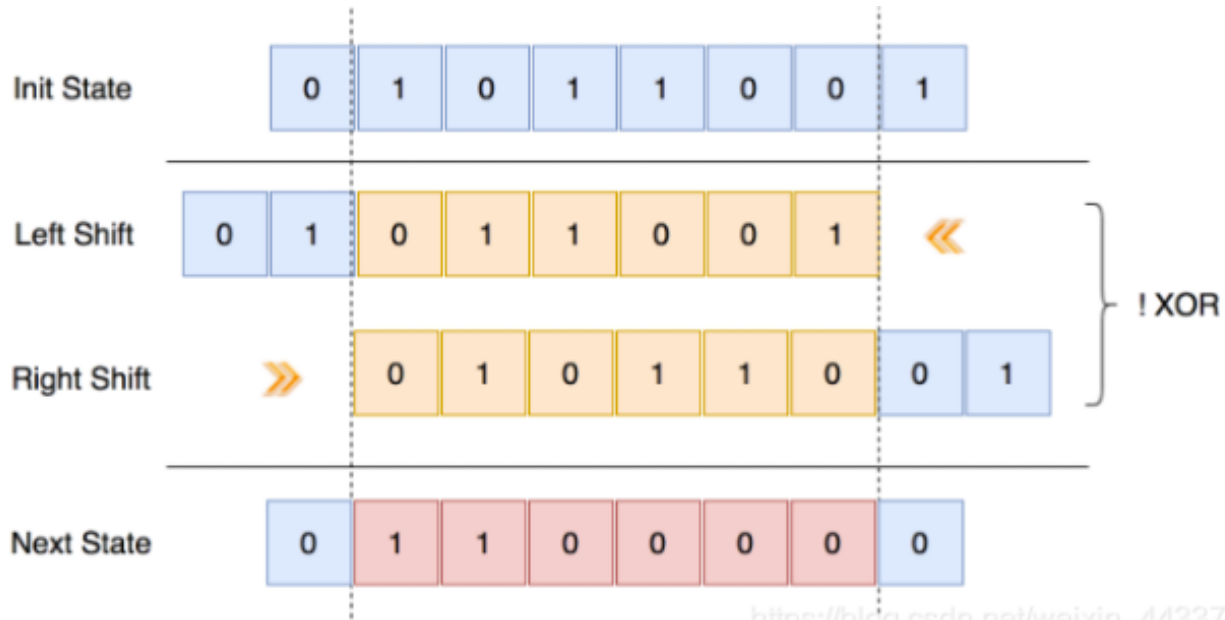
This is the method called fast-forward.

and if the number of possible states is very large, say 10^{10} , and it's even larger than N, then calculate the length of repetitive pattern is not acceptable. but in this problem, there will be 2^8 number of possible states. so we can calculate the length of cycle.

however, think twice about it. each time we need to check if this is a repetitive pattern of initial state. this is time consuming.

Solution2:

we have a better solution, in stead of change each digit at a time for each transaction, we use bit map, based on the follow rule:



https://blog.csdn.net/weixin_44337444

```
public int[] prisonAfterNDays(int[] cells, int N) {
    HashMap<Integer, Integer> seen = new HashMap<>();
    boolean isFastForwarded = false;
    // step 1). convert the cells to bitmap
    int stateBitmap = 0x0;
    for (int cell : cells) {
        stateBitmap <<= 1;
        stateBitmap = (stateBitmap | cell);
    }
    // step 2). run the simulation with hashmap
    while (N > 0) {
        if (!isFastForwarded) {
            if (seen.containsKey(stateBitmap)) {
                // the length of the cycle is seen[state_key] - N
                N %= seen.get(stateBitmap) - N;
                isFastForwarded = true;
            } else {
                seen.put(stateBitmap, N);
            }
            // check if there is still some steps remained,
            // with or without the fast forwarding.
            if (N > 0) {
                N -= 1;
                stateBitmap = this.nextDay(stateBitmap);
            }
        }
        // step 3). convert the bitmap back to the state cells
        int ret[] = new int[cells.length];
        for (int i = cells.length - 1; i >= 0; i--) {
            ret[i] = (stateBitmap & 0x1);
            stateBitmap = stateBitmap >> 1;
        }
        return ret;
    }
}

protected int nextDay(int stateBitmap) {
    stateBitmap = ~(stateBitmap << 1) ^ (stateBitmap >> 1);
    // set the head and tail to zero
    stateBitmap = stateBitmap & 0x7e;
    return stateBitmap;
}
```

Chapter 2

Bit Masks

2.0.1 总结一下

对于一个含有 N 个元素的集合，其总共包含 2^N 个子集，因此有个掩码的可能，每一个掩码表示一个子集。事实上，每一个掩码就是一个用二进制表示的整数，比如 1001 就是 9。

Bitmasking 是为每个掩码分配一个值（即为每个子集分配一个值），然后使用已经计算出的掩码值来计算新掩码的值。通常，我们的主要目标是为整个集合（即掩码 11111111）计算值。

要计算子集 X 的值，我们要么以各种可能的方式删除元素，并将获得的子集的值，来计算 X 的值或解。这意味着的值必须已经计算过，因此我们需要考虑掩码计算的先后顺序。

最容易想到就是自然序：按相应数字的递增顺序遍历并计算掩码所对应的解。同样，我们一般从空的子集 X 开始，然后以各种可能的方式添加元素，并使用解已知的子集的值来计算 X 的值/解。

掩码常见的操作和表示： $\text{bit}(i, \text{mask})$ 表示取掩码的第 i 位 $\text{count}(\text{mask})$ 表示掩码中非零位的个数 $\text{first}(\text{mask})$ 表示掩码中最低非零位的数目 $\text{set}(i, \text{mask})$ 表示设置掩码中的第 i 位 $\text{check}(i, \text{mask})$ 表示检查掩码中的第 i 位

而在基于状态压缩的动态规划中，我们常用到以下四种计算操作：

- 若当前状态为 S ，对 S 有下列操作。
 - 判断第 i 位是否为 0: $(S \& (1 \ll i)) == 0$ ，意思是将 1 左移 i 位与 S 进行与运算后，看结果是否为零。
 - 将第 i 位设置为 1: $S | (1 \ll i)$ ，意思是将 1 左移 i 位与 S 进行或运算。
 - 将第 i 位设置为 0: $S \& \sim(1 \ll i)$ ，意思是将 S 与第 i 位为 0，其余位为 1 的数进行与运算；
 - 取第 i 位的值: $S \& (1 \ll i)$

2.0.2 1655. Distribute Repeating Integers - Hard

You are given an array of n integers, `nums`, where there are at most 50 unique values in the array. You are also given an array of m customer order quantities, `quantity`, where `quantity[i]` is the amount of integers the i th customer ordered. Determine if it is possible to distribute `nums` such that:

The i th customer gets exactly `quantity[i]` integers, The integers the i th customer gets are all equal, and Every customer is satisfied. Return true if it is possible to distribute `nums` according to the above conditions.

1. 解题思路与分析: dfs 回塑

```
private boolean backTracking(int [] arr, int [] quantity, int idx) {
    if (idx < 0) return true;
    Set<Integer> vis = new HashSet<>();
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] < quantity[idx] || vis.contains(arr[i])) continue; // 去杂去重
        vis.add(arr[i]);
        arr[i] -= quantity[idx];
        if (backTracking(arr, quantity, idx-1)) return true;
        arr[i] += quantity[idx];
    }
    return false;
}

public boolean canDistribute(int[] nums, int[] quantity) {
    Map<Integer, Integer> map = new HashMap<>();
    for (Integer v : nums)
        map.put(v, map.getOrDefault(v, 0) + 1);
    int [] arr = new int [map.size()];
    int i = 0;
    for (Integer val : map.values()) arr[i++] = val;
    Arrays.sort(quantity); // decreasing frequency: 是一种裁枝优化
    return backTracking(arr, quantity, quantity.length-1);
}
```

- 不用 map 的操作相对快一点儿

```
public boolean canDistribute(int[] nums, int[] quantity) {
    int[] counts = new int[1001]; // compress the states first
    int n = 0;
    for (int i: nums) {
        counts[i]++;
        if (counts[i] == 1) n++;
    }
    nums = new int[n];
    int j = 0;
    for (int i: counts)
        if (i > 0) nums[j++] = i;
    return distribute(nums, quantity, 0);
}
private boolean distribute(int[] nums, int[] quantity, int idx) {
    if (i == quantity.length) return true;
    int q = quantity[idx];
    Set<Integer> used = new HashSet<>();
    for (int j = 0; j < nums.length; j++) {
        int k = nums[j];
        if (k < q || used.contains(k)) continue;
        nums[j] -= q;
        used.add(k); // Avoid duplicates. TLE without it.
        if (distribute(nums, quantity, i+1)) return true;
        nums[j] += q;
    }
    return false;
}
```

2. 解题思路与分析: 状态压缩 DP 首先, 容易发现 nums 的具体取值是不重要的: 只有每个取值出现的次数是重要的。因此, 我们构造 nums 的频次数组 cnt, 代表了原数组中每个数字出现的次数。

例如, 在数组 [3,2,2,5] 中, 只有数字 2 出现了 2 次, 故频次数组为 [1,2,1] (其顺序无关紧要)。

考虑到订单数目最多为 10, 故使用状态压缩动态规划解决本题: 用一个 $0 - 2^{10} (=1024)$ 的整数代表 mm 个顾客的一个子集。随后, 用 dp[i][j] 表示: cnt 数组中的前 i 个元素, 能否满足顾客的子集合 j 的订单需求。

考虑 dp[i][j] 时, 为了满足子集 j 的需求, 我们可以让 cnt[i] 满足 j 的某个子集 ss, 并让 cnt[0..i-1] 满足子集 js。对于特定的某个子集 ss 而言, 该种方案如果可行, 必然有 dp[i-1][js] 为 true, 且子集 s 的订单需求总和不超过 cnt[i]。

因此, 当且仅当能找到这样的子集 s 时, dp[i][j]=true。

```
public boolean canDistribute(int[] a, int[] quantity) {
    Map<Integer, Integer> map = new HashMap<>();
    for (Integer v : a)
        map.put(v, map.getOrDefault(v, 0) + 1);
    int n = map.size(), idx = 0, m = quantity.length, r = 1 << m;
    int[] cnt = new int[n];
    for (Integer v : map.values()) cnt[idx++] = v;
    int[] sum = new int[r];
    for (int i = 1; i < r; i++)
        for (int j = 0; j < m; j++)
            if (((i >> j) & 1) == 1) {
                int left = i - (1 << j);
                sum[i] = sum[left] + quantity[j];
                break;
            }
    boolean[][] dp = new boolean[n][r]; // dp[i][j] 表示: cnt 数组中的前 i 个元素, 能否满足顾客的子集合 j 的订单需求
    for (int i = 0; i < n; i++)
        dp[i][0] = true;
    for (int i = 0; i < n; i++) // 遍历 cnt 数组
        for (int j = 0; j < r; j++) { // 遍历客户组合子集
            if (i > 0 && dp[i-1][j]) {
                dp[i][j] = true;
                continue;
            }
            for (int k = j; k != 0; k = ((k-1) & j)) { // 子集 s 枚举, 详见 https://oi-wiki.org/math/bit/#_14
                int pre = j - k; // 前 i-1 个元素需要满足子集 prev = j-s
                boolean last = (i == 0) ? (pre == 0) : dp[i-1][pre]; // cnt[0..i-1] 能否满足子集 prev
                boolean need = sum[k] <= cnt[i]; // cnt[i] 能否满足子集 s
                if (last && need) {
                    dp[i][j] = true;
                    break;
                }
            }
        }
    return dp[n-1][r-1];
}
```

2.0.3 1659. Maximize Grid Happiness - Hard

You are given four integers, m, n, introvertsCount, and extrovertsCount. You have an m x n grid, and there are two types of people: introverts and extroverts. There are introvertsCount introverts and extrovertsCount extroverts.

You should decide how many people you want to live in the grid and assign each of them one grid cell. Note that you do not have to have all the people living in the grid.

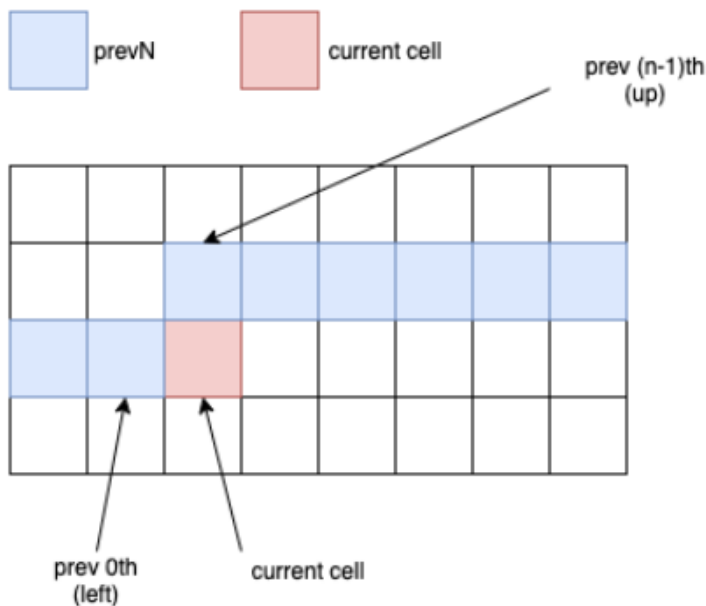
The happiness of each person is calculated as follows:

Introverts start with 120 happiness and lose 30 happiness for each neighbor (introvert or extrovert). Extroverts start with 40 happiness and gain 20 happiness for each neighbor (introvert or extrovert). Neighbors live in the directly adjacent cells north, east, south, and west of a person's cell.

The grid happiness is the sum of each person's happiness. Return the maximum possible grid happiness.

Key Notes:

- **Just DFS+Memo**
- For each cell we have 3 states -> **0: empty, 1: intro, 2: extro**
 - Only previous N bits matter (previous 1 is **left**, previous N is **up**)
 - Naturally, we use **Ternary** to construct our prevN bit mask
 - There are totally $3^5 = 243$ combinations, meaning the prevN value is bounded by 243.
- For each cell: try all possibilities: 0, 1 or 2
 - 0: do nothing, move our prevN bit mask with new 0 bit.
 - 1: happiness 120, move our prevN bit mask with new 1 bit.
 - 2: happiness 40, move our prevN bit mask with new 2 bit.
 - Check **up** and **left** cells to determine how many extra happiness need to be **added/subtracted**.



```
public int getMaxGridHappiness(int m, int n, int introvertsCount, int extrovertsCount) {
    this.m = m; this.n = n;
    dp = new Integer[m][n][introvertsCount+1][extrovertsCount+1][243]; // 3^5=243
    return dfs(0, 0, introvertsCount, extrovertsCount, 0);
}
Integer dp[][][][][];
int m, n, mod = 243;
private int dfs(int i, int j, int inCnt, int exCnt, int preMask) {
    if (inCnt == 0 && exCnt == 0) return 0;
    if (j == n) return dfs(i+1, 0, inCnt, exCnt, preMask); // 有到达边界, 直接看下一行
    if (i == m) return 0;
    if (dp[i][j][inCnt][exCnt][preMask] != null) return dp[i][j][inCnt][exCnt][preMask];
    int up = getBit(preMask, n-1), left = getBit(preMask, 0);
    int ans = dfs(i, j+1, inCnt, exCnt, setBit(preMask, 0)); // 不安排任何人在这个格
    if (inCnt > 0) { // 考虑安放一个内向的人在这个格子, 算最优可能性
        int addon = 120, cur = setBit(preMask, 1);
        if (i >= 1 && up != 0) { // up: 1 / 2
            addon -= 30; // 当前格内向的人, 上面有人他不开心
            if (up == 1) addon -= 30; // 当前格上一格, 如果是内向的人, 他不开心
            else addon += 20; // 当前格上一格, 如果是外向的人, 他很开心
        }
        if (j >= 1 && left != 0) {
            addon -= 30;
            if (left == 1) addon -= 30;
            else addon += 20;
        }
    }
}
```

```

    }
    ans = Math.max(ans, addon + dfs(i, j+1, inCnt-1, exCnt, cur));
}
if (exCnt > 0) { // 考虑安放一个外向的人在这个格子, 算最优可能性
    int addon = 40, cur = setBit(preMask, 2);
    if (i >= 1 && up != 0) {
        addon += 20;
        if (up == 1) addon -= 30;
        else addon += 20;
    }
    if (j >= 1 && left != 0) {
        addon += 20;
        if (left == 1) addon -= 30;
        else addon += 20;
    }
    ans = Math.max(ans, addon + dfs(i, j+1, inCnt, exCnt-1, cur));
}
return dp[i][j][inCnt][exCnt][preMask] = ans;
}
private int getBit(int v, int i) {
    v /= (int)Math.pow(3, i);
    return v % 3;
}
private int setBit(int v, int i) {
    return (v * 3 + i) % mod;
}
}

```

• 简洁版的代码

```

private int statemax = 1, mod = 0, R = 0, C = 0;
private int [][][][] dp;
public int getMaxGridHappiness(int m, int n, int introvertsCount, int extrovertsCount) {
    R = m;
    C = n;
    for (int i = 0; i < n; i++) statemax *= 3;
    mod = statemax / 3;
    dp = new int[m][n][introvertsCount + 1][extrovertsCount + 1][statemax];
    return dfs(0, 0, introvertsCount, extrovertsCount, 0);
}
private int dfs(int x, int y, int in, int ex, int last){
    if (x == R) return 0;
    if (y == C) return dfs(x + 1, 0, in, ex, last);
    if (dp[x][y][in][ex][last] != 0) return dp[x][y][in][ex][last];
    int res = dfs(x, y + 1, in, ex, last % mod * 3); // 不安排
    if (in != 0) { // 安排内向
        int addon = 120, up = last / mod, left = last % 3;
        if (x - 1 >= 0 && up != 0) {
            addon -= 30;
            addon += up == 1 ? -30 : 20;
        }
        if (y - 1 >= 0 && left != 0) {
            addon -= 30;
            addon += left == 1 ? -30 : 20;
        }
        res = Math.max(res, addon + dfs(x, y + 1, in - 1, ex, last % mod * 3 + 1));
    }
    if (ex != 0) { // 安排外向
        int addon = 40, up = last / mod, left = last % 3;
        if (x - 1 >= 0 && up != 0) {
            addon += 20;
            addon += up == 1 ? -30 : 20;
        }
        if (y - 1 >= 0 && left != 0) {
            addon += 20;
            addon += left == 1 ? -30 : 20;
        }
        res = Math.max(res, addon + dfs(x, y + 1, in, ex - 1, last % mod * 3 + 2));
    }
    return dp[x][y][in][ex][last] = res;
}
}

```

1. 解题思路与分析: 有一种状压, 叫做滑动窗口状压 (112ms)

```

public int getMaxGridHappiness(int m, int n, int introvertsCount, int extrovertsCount) {
    int r = (int)Math.pow(3, n), rless = (int)Math.pow(3, n-1);
    int [][] offset = {{0, 0, 0}, {0, -60, -10}, {0, -10, 40}};
    int [][][][] dp = new int [m*n+1][introvertsCount+1][extrovertsCount+1][r]; // [m*n+1]: +1 为的是去处理 dp[i+1][j][k][cur] 的 i = m*n-1 格的特殊
    for (int i = m*n-1; i >= 0; i--) { // i: idx, coordinates
        int x = i / n, y = i % n;
        for (int j = 0; j <= introvertsCount; j++)
            for (int k = 0; k <= extrovertsCount; k++) {
                for (int pre = 0; pre < r; pre++) { // pre 就是前 n 个格子的状态 (三进制)
                    int cur = (pre * 3) % r;
                    if (j > 0) { // 当前格安放一个内向的人
                        int dif = 120 + (y != 0 ? 1 : 0) * offset[1][pre % 3] + offset[1][pre / rless];
                        dp[i][j][k][pre] = Math.max(dp[i][j][k][pre], dif + dp[i+1][j-1][k][cur + 1]);
                    }
                    if (k > 0) { // 当前格安放一个外向的人

```

```

        int dif = 40 + (y != 0 ? 1 : 0) * offset[2][pre % 3] + offset[2][pre / rless];
        dp[i][j][k][pre] = Math.max(dp[i][j][k][pre], dif + dp[i+1][j][k-1][cur + 2]);
    }
    dp[i][j][k][pre] = Math.max(dp[i][j][k][pre], dp[i+1][j][k][cur]); // 当前格不放置任何人
}
}
}
return dp[0][introvertsCount][extrovertsCount][0];
}

```

2. 解题思路与分析: 其它

The maximum score of any sub grid is determined by following variables:

1. Introverts count.
2. Extroverts count.
3. The starting cell of the sub grid.
4. The introverts/extroverts placement status of previous N cells where n is the total column of the grid.

For 4, we only need to store last N cells because the above cell is N away and the left cell is one away from current cell if we flatten the grid to a 1d array.

```

xxxx
xxxX

```

e.g. We only need four previous cells to calculate the score for Cell C.

So we can use a rolling string to store the state. E.g. if current state is "1111" and we pick extrovert for current cell then the state becomes "1112".

Put all pieces together, we have following as the overall state and use a map to store the maximum score for any particular state:
idx + PrevNCellstate + Introverts Count + Extroverts Count

Time complexity: $m * n * 3^n * \text{IntroCount} * \text{ExtroCount}$

Space complexity: $m * n * 3^n * \text{IntroCount} * \text{ExtroCount}$

```

public int getMaxGridHappiness(int m, int n, int introvertsCount, int extrovertsCount) {
    Map<String, Integer> memo = new HashMap<>();
    return helper("0".repeat(n), 0, m, n, introvertsCount, extrovertsCount, memo);
}
private int helper(String state, int idx, int m, int n, int inCnt, int exCnt, Map<String, Integer> memo) {
    if (inCnt == 0 && exCnt == 0 || idx == m*n) return 0;
    String key = idx + state + inCnt + exCnt;
    if (memo.containsKey(key)) return memo.get(key);
    int i = idx / n, j = idx % n;
    int max = 0;
    if (inCnt > 0) { // case 1: place an introvert in this cell if possible.
        int curVal = 120;
        if (i > 0) curVal = calScore(state.charAt(0)-'0', 1, curVal);
        if (j > 0) curVal = calScore(state.charAt(state.length()-1)-'0', 1, curVal);
        max = Math.max(max, helper(state.substring(1)+"1", idx+1, m, n, inCnt-1, exCnt, memo) + curVal);
    }
    if (exCnt > 0) { // case 2: place an extrovert in this cell if possible.
        int curVal = 40;
        if (i > 0) curVal = calScore(state.charAt(0)-'0', 2, curVal);
        if (j > 0) curVal = calScore(state.charAt(state.length()-1)-'0', 2, curVal);
        max = Math.max(max, helper(state.substring(1)+"2", idx+1, m, n, inCnt, exCnt-1, memo) + curVal);
    }
    // case 3: Do not place any person.
    max = Math.max(max, helper(state.substring(1)+"0", idx+1, m, n, inCnt, exCnt, memo)); // 不要忘记这种选择
    memo.put(key, max);
    return max;
}
private int calScore(int i, int j, int v) {
    if (i == 1 && j == 1) return v - 60;
    if (i == 2 && j == 2) return v + 40;
    if (i == 1 && j == 2 || i == 2 && j == 1) return v - 10;
    return v;
}

```

- 还有一种其它语言写的，行与行之间以行为单位进行动态规划状态转移的，可以搜出来参考一下

2.0.4 1755. Closest Subsequence Sum - Hard 分成两半: 由 $O(2^N)$ 降为 $O(N \log N)$

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

You want to choose a subsequence of nums such that the sum of its elements is the closest possible to goal. That is, if the sum of the subsequence's elements is sum, then you want to minimize the absolute difference $\text{abs}(\text{sum} - \text{goal})$.

Return the minimum possible value of $\text{abs}(\text{sum} - \text{goal})$.

Note that a subsequence of an array is an array formed by removing some elements (possibly all or none) of the original array.

```
public int minAbsDifference(int[] arr, int goal) {
    int n = arr.length;
    List<Integer> lsum = new ArrayList<>();
    List<Integer> rsum = new ArrayList<>();
    lsum.add(0);
    rsum.add(0);
    for (int i = 0; i <= n/2; i++) { // 这种生成和的方式, i < n/2
        int size = lsum.size();
        for (int j = 0; j < size; j++)
            lsum.add(lsum.get(j) + arr[i]);
    }
    for (int i = n/2+1; i < n; i++) { // int i = n/2 同样可以, 只是左右大小的细微差别
        int size = rsum.size();
        for (int j = 0; j < size; j++)
            rsum.add(rsum.get(j) + arr[i]);
    }
    TreeSet<Integer> rightSumSet = new TreeSet<>(rsum);
    Set<Integer> leftSumSet = new HashSet<>(lsum);
    int ans = Math.abs(goal);
    for (int v : leftSumSet) {
        int b = goal - v;
        Integer lower = rightSumSet.floor(b); // 对 treeset 的这几个函数总是记不住
        Integer higher = rightSumSet.ceiling(b);
        if (lower != null)
            ans = Math.min(ans, Math.abs(goal-v-lower));
        if (higher != null)
            ans = Math.min(ans, Math.abs(goal-v-higher));
    }
    return ans;
}
```

- 另一种我都怀疑是不是自己写出来的, 居然会忘了。。。。。

```
// 要把这种工具方法像写 binarySearch 一样随手拈来随手就敲才行: 仍然超时, 最后两个不过
public void getSum(List<Integer> li, int [] arr, int sum, int l, int r) {
    if (l >= r) { //
        li.add(sum);
        return;
    }
    getSum(li, arr, sum + arr[l], l+1, r); // choose and add idx l to sum
    getSum(li, arr, sum, l+1, r); // skip idx l, move directly to next element
}
public int minAbsDifference(int[] arr, int goal) {
    int n = arr.length, m = arr.length / 2;
    List<Integer> l = new ArrayList<>();
    List<Integer> r = new ArrayList<>();
    getSum(l, arr, 0, 0, m); // m
    getSum(r, arr, 0, m, n); // m ? 这里反而想不明白了?
    Collections.sort(l);
    Collections.sort(r);
    int i = 0, j = r.size()-1, cur = 0;
    int minDiff = Integer.MAX_VALUE;
    while (i < l.size() && j >= 0) {
        cur = l.get(i) + r.get(j) - goal;
        if (cur > 0) {
            minDiff = Math.min(minDiff, cur);
            j--;
        } else if (cur < 0) {
            minDiff = Math.min(minDiff, -cur);
            i++;
        }
    }
    else return 0;
}
return minDiff;
}
```

- Horowitz and Sahni's Subset Sum | comments | links

– <https://leetcode.com/problems/closest-subsequence-sum/discuss/1055432/Java-252ms-or-47.4-MB-or-Horowitz-Subset-Sum-or-comments-or-links>

- 好像还有一个类似提交 python 关于子集的位操作的 java 方法, 回头再找来参考一下

2.0.5 2035. Partition Array Into Two Arrays to Minimize Sum Difference - Hard 上一题: 分成两半的套娃题 todo: + binarySearch 等解法

You are given an integer array `nums` of $2 * n$ integers. You need to partition `nums` into two arrays of length `n` to minimize the absolute difference of the sums of the arrays. To partition `nums`, put each element of `nums` into one of the two arrays.

Return the minimum possible absolute difference.

```

public int minimumDifference(int[] nums) {
    int n = nums.length;
    int sum = Arrays.stream(nums).sum();
    TreeSet<Integer>[] sets = new TreeSet[n/2+1]; // 数组, 而不是 hashMap, 这个应该关系不是很大
    for (int i = 0; i < (1 << (n / 2)); ++i) { // 一次遍历, 而不是 n 次遍历
        int curSum = 0;
        int m = 0; // element Cnts
        for (int j = 0; j < n / 2; ++j)
            if ((i & (1<<j)) != 0) {
                curSum += nums[j]; // 左半部分
                m++;
            }
        if (sets[m] == null) sets[m] = new TreeSet<Integer>();
        sets[m].add(curSum);
    }

    int res = Integer.MAX_VALUE;
    for (int i = 0; i < (1 << (n / 2)); ++i) {
        int curSum = 0;
        int m = 0;
        for (int j = 0; j < n / 2; ++j)
            if ((i & (1<<j)) != 0) {
                curSum += nums[n/2 + j]; // 遍历计算右半部分的和: 边遍历, 边解决问题
                m++;
            }
        int target = (sum - 2 * curSum) / 2;
        Integer left = sets[n/2-m].floor(target), right = sets[n/2-m].ceiling(target);
        if (left != null)
            res = Math.min(res, Math.abs(sum - 2 * (curSum + left.intValue())));
        if (right != null)
            res = Math.min(res, Math.abs(sum - 2 * (curSum + right.intValue())));
        if (res == 0) return 0;
    }
    return res;
}

```

- 看一下另一个写法

```

public int minimumDifference(int[] a) {
    int n = a.length, ans = Integer.MAX_VALUE;
    int [] left = Arrays.copyOfRange(a, 0, n/2);
    int [] right = Arrays.copyOfRange(a, n/2, n);
    Map<Integer, TreeSet<Integer>> lsum = new HashMap<>();
    Map<Integer, TreeSet<Integer>> rsum = new HashMap<>();
    for (int i = 1; i < (1 << (n/2)); i++) { // 这里一次遍历把左右两边的都算出来了
        int sumLeft = 0, sumRight = 0, cnt = 0;
        for (int j = 0; j < n/2; j++) {
            cnt += (i >> j) & 1;
            if (((i >> j) & 1) == 1) {
                sumLeft += left[j];
                sumRight -= right[j];
            } else {
                sumLeft -= left[j];
                sumRight += right[j];
            }
        }
        lsum.computeIfAbsent(cnt, z -> new TreeSet<>()).add(sumLeft);
        rsum.computeIfAbsent(cnt, z -> new TreeSet<>()).add(sumRight);
    }
    for (int i = 1; i <= n/2; i++)
        for (Integer lv : lsum.get(i)) {
            if (rsum.get(i).contains(-lv)) return 0;
            Integer ceiling = rsum.get(i).ceiling(-lv);
            Integer floor = rsum.get(i).floor(-lv);
            if (ceiling != null)
                ans = Math.min(ans, Math.abs(ceiling + lv));
            if (floor != null)
                ans = Math.min(ans, Math.abs(floor + lv));
        }
    // 暴力匹配会超时
    // for (int i = 1; i < n/2; i++)
    //     for (int lv : lsum.get(i))
    //         for (int rv : rsum.get(i))
    //             ans = Math.min(ans, Math.abs(lv - rv));
    return ans;
}

```

2.0.6 956. Tallest Billboard: 折腰轨半第三次: 重要的题型重复三遍

You are installing a billboard and want it to have the largest height. The billboard will have two steel supports, one on each side. Each steel support must be an equal height. You are given a collection of rods that can be welded together. For example, if you have rods of lengths 1, 2, and 3, you can weld them together to make a support of length 6. Return the largest possible height of your billboard installation. If you cannot support the billboard, return 0.

1. 解题思路与分析: dfs 记忆化搜索? 没想到吧。。。。。

对于每一根钢筋 x ，我们会写下 $+x$ 、 $-x$ 或者 0 。我们的目标是最终得到结果 0 并让正数之和最大。我们记所有写下的正数之和为 $score$ 。例如， $+1 +2 +3 -6$ 的 $score$ 为 6 。

因为 $sum(rods)$ 的大小限制，就说明可以利用这个性质。事实上，如果之前已经写下了一些数字，那么就不需要考虑这些数字是如何得到的。例如， $rods = [1, 2, 2, 3]$ ，我们可以用 3 种方法得到和为 3 ，但只考虑最终的 $score$ 为 3 。数字之和的上界是 10001 ，因为只有 $[-5000, 5000]$ 区间内的整数是可能的值。

算法

$dp[i][s]$ 表示当我们可以使用 $rods[j]$ ($j \geq i$) 时能得到的最大 $score$ ，由于之前写下的数字和为 s （不统计在 $score$ 内）。例如， $rods = [1, 2, 3, 6]$ ，可以有 $dp^{1,2} = 5$ ，在写下 1 之后，可以写下 $+2 +3 -6$ 使得剩下的 $rods[i:]$ 获得 $score$ 为 5 。边界情况： $dp[rods.length][s]$ 是 0 当 $s == 0$ ，剩余情况为 $-infinity$ 。递推式为 $dp[i][s] = \max(dp[i+1][s], dp[i+1][s-rods[i]], rods[i] + dp[i+1][s+rods[i]])$ 。

```
public int tallestBillboard(int[] rods) { // dp: dfs 记忆化搜索？从来不曾认出来过。。。dfs 记忆化搜索 真是全能呀。。。终于会写这个题了。。。。。
    int N = rods.length;
    // "dp[n][x]" will be stored at dp[n][5000+x]
    // Using Integer for default value null
    dp = new Integer[N][10001];
    return (int)dfs(rods, 0, 5000);
}
Integer[][] dp;
public int dfs(int[] rods, int i, int s) {
    if (i == rods.length)
        return s == 5000 ? 0 : Integer.MIN_VALUE / 3;
    if (dp[i][s] != null) return dp[i][s];
    int ans = dfs(rods, i+1, s);
    ans = Math.max(ans, dfs(rods, i+1, s-rods[i]));
    ans = Math.max(ans, rods[i] + dfs(rods, i+1, s+rods[i]));
    dp[i][s] = ans;
    return ans;
}
```

2. 解题思路与分析: 折腰轨半搜索（和上面两个题比一比）

暴力搜索的复杂度可以用“折半搜索”优化。在这个问题中，我们有 3^N 种可行方案，对于每个钢筋 x 可以考虑 $+x$ 、 $-x$ ，或者 0 ，我们要让暴力的速度更快。

我们可以让前 $3^{\{N/2\}}$ 和后一半分开来考虑，然后再合并他们。例如，如果有钢筋 $[1, 3, 5, 7]$ ，那么前两根钢筋可以构成九种状态： $[0+0, 0+3, 0-3, 1+0, 1+3, 1-3, -1+0, -1+3, -1-3]$ ，后两根钢筋也可以构成九种状态。

我们对每个状态记录正数之和，以及负数绝对值之和。例如， $+1 +2 -3 -4$ 记为 $(3, 7)$ 。同时记状态的 δ 为两者之差 $3-7$ ，所以这个状态的 δ 为 -4 。

我们的目标是将两个状态合并，使得 δ 之和为 0 。 $score$ 是所有正数之和，我们希望获得最高的 $score$ 。对于每个 δ 我们只会记录具有最高 $score$ 的状态。

算法

将钢筋分成左右两半：左侧和右侧。

对于每一半，暴力计算可达的所有状态，如上定义。然后针对所有状态，记录下 δ 和最大的 $score$ 。

然后我们有左右两半的 $[(\delta, score)]$ 信息。我们找到 δ 为 0 时最大的 $score$ 和。

```
public int tallestBillboard(int[] rods) { // 这个非 dp 的解法亲民贴近刷题老百姓了。。。。看得还有点儿稀里糊涂
    int n = rods.length;
    Map<Integer, Integer> leftDelta = make(Arrays.copyOfRange(rods, 0, n/2));
    Map<Integer, Integer> rightDelta = make(Arrays.copyOfRange(rods, n/2, n));
    int ans = 0;
    for (int d : leftDelta.keySet())
        if (rightDelta.containsKey(-d))
            ans = Math.max(ans, leftDelta.get(d) + rightDelta.get(-d));
    return ans;
}
public Map<Integer, Integer> make (int [] a) {
    Point [] dp = new Point [60000]; // 3^10 = 59049
    int idx = 0;
    dp[idx++] = new Point(0, 0);
    for (Integer v : a) {
        int stop = idx;
        for (int i = 0; i < stop; i++) {
            Point p = dp[i];
            dp[idx++] = new Point(p.x + v, p.y); // 要么加入第一根长棒
            dp[idx++] = new Point(p.x, p.y + v); // 要么加入第二根长棒
        }
    }
    Map<Integer, Integer> ans = new HashMap<>();
    for (int i = 0; i < idx; i++) {
        int a = dp[i].x, b = dp[i].y;
        // k, v: v 对应 score, 即每个 delta 所对应的最大正数和, 即 dp[i].x 的值
        // score 是所有正数之和, 我们希望获得最高的 score。
    }
}
```

¹DEFINITION NOT FOUND.

```

        ans.put(a-b, Math.max(ans.getDefault(a-b, 0), a)); // 对于每个 delta 我们只会记录具有最高 score 的状态
    }
    return ans;
}

```

- 之前看别人的，还需要理解消化，官方 dp 解也需要消化一下

<https://leetcode-cn.com/problems/tallest-billboard/solution/zui-gao-de-yan-gao-pai-by-leetcode/>

```

// // https://blog.csdn.net/luke2834/article/details/89457888 // 这个题目要多写几遍
public int tallestBillboard(int[] rods) { // 写得好奇呀
    int n = rods.length;
    int sum = Arrays.stream(rods).sum();
    System.out.println("sum: " + sum);
    int [][] dp = new int [2][(sum + 1) << 1]; // (sum + 1) * 2
    for (int i = 0; i < 2; i++)
        Arrays.fill(dp[i], -1);
    dp[0][sum] = 0;
    for (int i = 0; i < n; i++) {
        int cur = i & 1, next = (i & 1) ^ 1; // 相当于是滚动数组: [0, 1]
        for (int j = 0; j < dp[cur].length; j++) {
            if (dp[cur][j] == -1) continue;
            dp[next][j] = Math.max(dp[cur][j], dp[next][j]); // update to max
            dp[next][j+rods[i]] = Math.max(dp[next][j+rods[i]], dp[cur][j] + rods[i]);
            dp[next][j-rods[i]] = Math.max(dp[next][j-rods[i]], dp[cur][j] + rods[i]);
        }
    }
    return dp[rods.length & 1][sum] >> 1; // dp[n&1][sum] / 2
}

```

- 这里详细纪录一下生成过程，记住这个方法

```

int [] a = new int [] {1, 2, 3};
sum: 6
i: 0
-1, -1, -1, -1, -1, -1, 0, -1, -1, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, 1, 0, 1, -1, -1, -1, -1, -1, -1,
i: 1
-1, -1, -1, 3, 2, 3, 0, 3, 2, 3, -1, -1, -1, -1, -1,
-1, -1, -1, -1, -1, 1, 0, 1, -1, -1, -1, -1, -1, -1,
i: 2
-1, -1, -1, 3, 2, 3, 0, 3, 2, 3, -1, -1, -1, -1, -1,
6, 5, 6, 3, 6, 5, 6, 5, 6, 3, 6, 5, 6, -1,
r: 3

```

- 另一种写法

```

// 1. 所有的状态全集: dp[len][sum+1], len = length of array, sum = sum of the array, 代表两边共有的高度
// 2. state transfer:
// a. 忽略当前 dp[i][j] = max(dp[i-1][j], dp[i][j])
// b. 加入到 higher 一侧 dp[i][j+h] = max(dp[i][j+h], dp[i-1][j])
// c. 加入到 lower 一侧 lower = abs(j-h); dp[i][lower] = max(dp[i][lower], dp[i][j] + min(j, h)); 其中 min(j, h) 为新增高度
private void dfs(int [] arr, int idx) {
    int cur = arr[idx];
    if (dp[idx][cur] != -1) return;
    if (idx == 0) {
        dp[idx][cur] = 0; // add
        dp[idx][0] = 0; // ignore
        return;
    }
    dfs(arr, idx-1);
    int lower = 0;
    for (int i = 0; i < dp[idx].length-cur; i++) {
        if (dp[idx-1][i] < 0) continue;
        dp[idx][i] = Math.max(dp[idx][i], dp[idx-1][i]); // 1: ignore
        dp[idx][i+cur] = Math.max(dp[idx][i+cur], dp[idx-1][i]); // 2: add to higher
        lower = Math.abs(i - cur); // 3: add to lower
        dp[idx][lower] = Math.max(dp[idx][lower], dp[idx-1][i] + Math.min(i, cur));
    }
}
int [][] dp;
int n;
public int tallestBillboard(int[] rods) {
    int n = rods.length;
    int sum = Arrays.stream(rods).sum();
    dp = new int [n][sum+1];
    for (int i = 0; i < n; i++)
        Arrays.fill(dp[i], -1);
    dfs(rods, n-1);
    return dp[n-1][0];
}

```

- 这里详细纪录一下生成过程，记住这个方法

```

int [] a = new int [] {1, 2, 3};
i: 0
0, 0, -1, -1, -1, -1, -1,
0, -1, 0, -1, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1,
i: 1
0, 0, -1, -1, -1, -1, -1,
0, 1, 0, 0, -1, -1, -1,
-1, -1, -1, -1, -1, -1, -1,
i: 0
0, 0, -1, -1, -1, -1, -1,
0, 1, 0, 0, -1, -1, -1,
0, -1, -1, 0, -1, -1, -1,
i: 1
0, 0, -1, -1, -1, -1, -1,
0, 1, 0, 0, -1, -1, -1,
0, 1, 2, 0, 1, -1, -1,
i: 2
0, 0, -1, -1, -1, -1, -1,
0, 1, 0, 0, -1, -1, -1,
0, 2, 2, 0, 1, 0, -1,
i: 3
0, 0, -1, -1, -1, -1, -1,
0, 1, 0, 0, -1, -1, -1,
3, 2, 2, 0, 1, 0, 0,
r: 3

```

```

// 定义一个数对键值: (i,j): i 表示两个子序列的累加和差值的绝对值, j 表示这个差值下, 子序列中累加和的最大值, 定义一个 dp 的 map 存放前 m 个数的所有子序列的累加和
// 新建一个 HashMap temp 用于存放第 m 个数对之前子序列累加和和只差的状态转移结果
// 对于新到来的 rod, 只能够有 3 中情况:
// 1. rod 不加入任何列表:
// 从 dp 中拿出每个子序列的差值 k 的累加和最大值 v1, 每个结果与 temp 中相应 k 的累加和最大值 v2 比较, 如果 v1>v2, 那么更新 temp 中 k 对应的最大值
// 2. rod 加入累加和较大的序列
// 从 dp 中拿出每个子序列的差值 k 的累加和最大值 v1, 并加上 rod, 这时差值变成 k + rod, 累加和最大值变成 v1+ rod, 每个结果与 temp 中相应 k + rod 的累加和最大值比较
// 3. rod 加入累加和较小的序列
// 从 dp 中拿出每个子序列的差值 k 的累加和最大值 v1, 累加和较小的子序列加上了 rod, 那么和累加和较大的子序列之差为 k-rod, k-rod 为负数时, 说明累加和较小的子序列的累加和最大值 v1 加上 rod 后大于累加和较大的子序列的累加和最大值, 那么更新 temp 中 k-rod 的累加和最大值
public int tallestBillboard(int[] rods) {
    int n = rods.length;
    Map<Integer, Integer> dp = new HashMap<>();
    dp.put(0, 0);
    for (int rod : rods) {
        System.out.println("\nrod: " + rod);
        Map<Integer, Integer> tmp = new HashMap<>();
        dp.forEach((k, v) -> {
            if (tmp.getOrDefault(k, -1) < v) tmp.put(k, v);
            if (tmp.getOrDefault(k+rod, -1) < v+rod) tmp.put(k + rod, v+rod);
            int dis = k - rod;
            int larger = Math.max(v, v-dis);
            dis = Math.abs(dis);
            if (tmp.getOrDefault(dis, -1) < larger) tmp.put(dis, larger);
        });
        dp = tmp;
    }
    return dp.get(0);
}

```

2.0.7 1982. Find Array Given Subset Sums - Hard

You are given an integer n representing the length of an unknown array that you are trying to recover. You are also given an array `sums` containing the values of all 2^n subset sums of the unknown array (in no particular order).

Return the array `ans` of length n representing the unknown array. If multiple answers exist, return any of them.

An array `sub` is a subset of an array `arr` if `sub` can be obtained from `arr` by deleting some (possibly zero or all) elements of `arr`. The sum of the elements in `sub` is one possible subset sum of `arr`. The sum of an empty array is considered to be 0.

Note: Test cases are generated such that there will always be at least one correct answer.

本题的简化版（`sums` 中所有元素均为非负数）是一道非常经典的题目。本题在原题基础上进行了拓展，是一道兼具思考性和趣味性的好题。

所有元素均为非负数

该限制条件下可以用归纳法求解。在下述解法中，“从 multiset 里去除元素 x ”指的是去除一个 x ，而不是去除所有 x 。

1. 设 S 表示 `sums` 中所有元素组成的 multiset。
2. 首先将 0 从 S 中去除，此时 S 中的最小值即为 `ans` 中的最小值。
3. 若我们已经推出了 `ans` 中最小的 k 个元素，那么我们从 S 中把这 k 个元素所有子集的和去除。此时 S 中的最小值即为 `ans` 中的第 $(k+1)$ 小值。

复杂度 $\mathcal{O}(2^n n)$ 。

所有元素可以为任意整数

解法

1. 令 $m < 0$ 表示 S 中的最小值，将 S 中所有元素增加 $-m$ 变成另一个 multiset S' 。
2. 对 S' 按 `sums` 中所有元素均为非负数的方法求解，得到答案 `tmp`。
3. 寻找 `tmp` 中的任意一个子集，使得该子集的和等于 m 。把子集中的所有元素变为相反数（乘以 -1 ）后得到的 `ans` 即为最终答案。

复杂度仍为 复杂度 $\mathcal{O}(2^n n)$ 。

证明

令 A 表示“标准答案”的 multiset，令 A^- 表示 A 中所有负数组成的 multiset，显然有 $\sum A^- = m$ 。

观察 $T \subseteq A$ ，可以发现 $\sum T - m = \sum (T \oplus A^-)$ ，其中 \oplus 是这样一种集合运算：若 A^- 中的一个元素 a 也存在于 T 中，则将其从 T 中去除；否则将 $-a$ 加入 T 。因此原问题和转化后的问题的子集具有一一对应的映射关系，其中一个问题有解，另一个问题也有解。

1. 解题思路与分析 求子集和可以用二进制状压 DP

以 i 结尾的子集和 $dp[mask \mid 1 \ll i] = dp[mask] + ans[i]$ ， $(0 \leq mask < 1 \ll i)$

最后找与偏移量相等的子集和的时候还能复用到

```
public int[] recoverArray(int n, int[] sums) {
    int offset = Math.min(0, Arrays.stream(sums).min().getAsInt());
    offset = -offset;
    TreeMap<Integer, Integer> m = new TreeMap<>(); // java 没有 multiset, 使用 TreeMap 代替 {子集和: 出现的次数}, 次数为 0 则删除掉
    for (Integer v : sums) {
        v += offset;
        m.put(v, m.getOrDefault(v, 0) + 1);
    }
    m.put(0, m.get(0) - 1); // 先 (试图) 移除空子集 0: 后面会统一移除
    int [] dp = new int [1 << n]; // 状态压缩, 存储已移除掉的子集, 初始化 dp[0]=0 可省略
    int [] ans = new int [n];
    for (int i = 0; i < n; i++) {
        // 延迟删除策略, 最小值次数为 0, 则抛弃掉此最小值
        while (m.firstEntry().getValue() == 0) m.pollFirstEntry();
        ans[i] = m.firstKey();
        for (int mask = 0; mask < 1 << i; mask++) {
            dp[mask | 1 << i] = dp[mask] + ans[i];
            m.put(dp[mask | 1 << i], m.get(dp[mask | 1 << i]) - 1); // 减少 cnt, 为的是接下来的移除
        }
    }
    for (int mask = 0; ++mask)
        if (dp[mask] == offset) {
```

```

    for (int i = 0; i < n; i++)
        if ((mask & 1 << i) != 0)
            ans[i] = -ans[i];
    return ans;
}

```

- 官方题解：可以参考一下 <https://leetcode-cn.com/problems/find-array-given-subset-sums/solution/cong-zi-ji-de-he-huan-yuan-shu-zu-by-lee-aj8o/>

2.0.8 1815. Maximum Number of Groups Getting Fresh Donuts - Hard

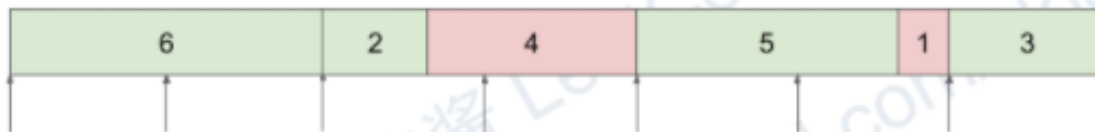
There is a donuts shop that bakes donuts in batches of `batchSize`. They have a rule where they must serve all of the donuts of a batch before serving any donuts of the next batch. You are given an integer `batchSize` and an integer array `groups`, where `groups[i]` denotes that there is a group of `groups[i]` customers that will visit the shop. Each customer will get exactly one donut.

When a group visits the shop, all customers of the group must be served before serving any of the following groups. A group will be happy if they all get fresh donuts. That is, the first customer of the group does not receive a donut that was left over from the previous group.

You can freely rearrange the ordering of the groups. Return the maximum possible number of happy groups after rearranging the groups.

[6,2,4,5,1,3]

4 groups start with a new batch => ans = 4



Key observation:

1. If groups have sizes that are factors of batch size, we can put it in the front and all of them will be happy. And it is optimal.
2. For groups of sizes that are larger than batch size, we can use $n \% k$ as the new group size which will not affect the final results.



1. 解题思路与分析: // 回塑 DFS + 记忆化搜索

```

public int maxHappyGroups(int batchSize, int [] groups) {
    this.batchSize = batchSize;
    n = groups.length;
    cnt = new int [batchSize]; // group size 30 is too large for backtracking WITHOUT modifications
    for (Integer v : groups) // 预处理: 压缩输入以便回塑
        cnt[v % batchSize] ++;
    // take out the 2 remainder's min groups if their sum is batchSize.
    // it still works but slow without this step
    // Note: < batchSize / 2 to avoid when i is batchSize / 2 it will subtract itself
    int ans = cnt[0], min = 0; // 优化: 通过这里的优化, 将回塑的数据规模大副减小
    for (int i = 1; i < batchSize / 2; i++) {
        min = Math.min(cnt[i], cnt[batchSize-i]);
        cnt[i] -= min;
        cnt[batchSize-i] -= min;
        ans += min;
    }
    return dfs(0, n - cnt[0]) + ans;
}

Map<String, Integer> dp = new HashMap<>();
int [] cnt;
int n, batchSize;
private int dfs(int sum, int leftOver) {
    if (leftOver == 0) return 0;
    String key = Arrays.toString(cnt);
    if (dp.containsKey(key)) return dp.get(key);
    int ans = 0;

```

```

for (int i = 1; i < batchSize; i++) {
    if (cnt[i] == 0) continue;
    cnt[i]--;
    ans = Math.max(ans, dfs(sum + i, leftOver-1) + (sum % batchSize == 0 ? 1 : 0));
    cnt[i]++;
    if ((sum + i) % batchSize == 0) break; // 当前顾客恰好能买完一批甜甜圈，已是最优情况之一，可剪枝 <<<=====
}
dp.put(key, ans);
return ans;
}

```

- 另一种 DP 超时的做法，掌握这个方法

```

// Time complexity: O(n*2n) TLE TLE TLE
// Space complexity: O(2n) TLE TLE TLE
public int maxHappyGroups(int batchSize, int[] groups) { // tle
    int n = groups.length;
    int[] dp = new int[1 << n];
    for (int mask = 0; mask < 1 << n; mask++) {
        int s = 0;
        for (int i = 0; i < n; i++)
            if ((mask & (1 << i)) > 0)
                s = (s + groups[i]) % batchSize;
        for (int i = 0; i < n; i++)
            if ((mask & (1 << i)) == 0)
                dp[mask | (1 << i)] = Math.max(dp[mask | (1 << i)], dp[mask] + (s == 0 ? 1 : 0));
    }
    return dp[(1 << n) - 1];
}

```

2.0.9 691. Stickers to Spell Word - Hard

We are given n different types of stickers. Each sticker has a lowercase English word on it.

You would like to spell out the given string `target` by cutting individual letters from your collection of stickers and rearranging them. You can use each sticker more than once if you want, and you have infinite quantities of each sticker.

Return the minimum number of stickers that you need to spell out `target`. If the task is impossible, return `-1`.

Note: In all test cases, all words were chosen randomly from the 1000 most common US English words, and `target` was chosen as a concatenation of two random words.

- **【位图法】** 因为待匹配串 `target` 的数量最多是 15 个，因此其子集的数量最多有 2^{15} 个，而 `int` 类型占用四个字节，能够容纳标识所有 `target` 的子集。所以我们可以将 `target` 的子集映射到 `int` 的整型数中。
- **【int 与 target 子集之间的映射关系】** 将 `int` 类型分解为二进制的形式后，有些位置为 0，有些位置为 1。表明在 `target` 中哪些位置的字符是否保留（1 表示保留）。
- **【动态规划】** `dp` 中存储的是得到子集 `i`，需要的最少的单词的数量。

```

public int minStickers(String[] stickers, String target) {
    int m = target.length(), n = 1 << m;
    int[] dp = new int[1 << m];
    Arrays.fill(dp, Integer.MAX_VALUE);
    dp[0] = 0;
    int cur = 0;
    for (int i = 0; i < n; i++) {
        if (dp[i] == Integer.MAX_VALUE) continue;
        for (String s : stickers) {
            cur = i; // 关键代码（下面：在 i 上面加入一个单词后的效果）
            for (char c : s.toCharArray()) // for each char in the sticker, try apply it on a missing char in the subset of target
                for (int j = 0; j < m; j++)
                    if (target.charAt(j) == c && ((cur >> j) & 1) == 0) {
                        cur |= 1 << j; // 在 cur 中相应位置，加入 c，形成新的集合。
                        break;
                    }
            dp[cur] = Math.min(dp[cur], dp[i] + 1); // 判断是否需要替换原来 cur 中的值。
        }
    }
    return dp[n-1] == Integer.MAX_VALUE ? -1 : dp[n-1];
}

```

- 另一种解法

```

private int helper(String s) {
    if (dp.containsKey(s)) return dp.get(s);
    int ans = Integer.MAX_VALUE;
    int[] tar = new int[26];
    for (char c : s.toCharArray())
        tar[c-'a']++;
    for (int i = 0; i < m; i++) {

```

```

    if (map[i][s.charAt(0) - 'a'] == 0) continue;
    StringBuilder sb = new StringBuilder();
    for (int j = 0; j < 26; j++) {
        if (tar[j] > 0)
            for (int k = 0; k < Math.max(0, tar[j] - map[i][j]); k++)
                sb.append((char)('a' + j));
    }
    int tmp = helper(sb.toString());
    if (tmp != -1) ans = Math.min(ans, 1 + tmp);
}
dp.put(s, ans == Integer.MAX_VALUE ? -1 : ans);
return dp.get(s);
}
}
Map<String, Integer> dp;
int [][] map;
int m;
public int minStickers(String[] stickers, String target) {
    m = stickers.length;
    map = new int[m][26];
    dp = new HashMap<>();
    for (int i = 0; i < m; i++)
        for (char c : stickers[i].toCharArray())
            map[i][c - 'a']++;
    dp.put("", 0);
    return helper(target);
}
}

```

- 上面的这个，因为使用了图，以及必要的优化，性能还比较好

什么叫状态压缩？其实就是用二进制数来表示动态规划状态转移过程中的状态。

什么时候应该状态压缩？状态压缩的题目，一般都会有非常明显的标志：如果你看到有一个参数的数值小于 20，同时这道题目中有涉及到是否选取、是否使用这样的二元状态，那么这道题目很可能就是一道状态压缩的题目。

本题中的标志就是 `target` 的长度不超过 15。于是，我们可以用一个二进制数表示 `target` 的每一位是否已经获取到。

后得到的状态对应的二进制数一定大于它的父状态。所以我们可以很自然地从小状态开始，一直遍历到 111...111（目标状态）。对于每一个状态，我们遍历所有的 `stickers`，看它能够更新出怎样的状态。

为了减少计算量，预处理得到了每一个 `sticker` 包含的每一种小写字母的个数。

这里讲的 `++` 的状态优化，可以参考一下

<https://leetcode-cn.com/problems/stickers-to-spell-word/solution/zhuang-tai-ya-suo-dpji-you-hua-by-luci>

```

int INF = std::numeric_limits<int>::max();
int minStickers(vector<string>& stickers, string target) {
    vector<int> dp(1 << 15, INF);
    int n = stickers.size(), m = target.size();
    vector<vector<int>> cnt(n, vector<int>(26));
    for (int i = 0; i < n; ++i)
        for (char c : stickers[i])
            cnt[i][c - 'a']++;

    dp[0] = 0;
    for (int i = 0; i < (1 << m); ++i) {
        if (dp[i] == INF) continue;
        for (int k = 0; k < n; ++k) {
            int nxt = i;
            vector<int> left(cnt[k]);
            for (int j = 0; j < m; ++j) {
                if (nxt & (1 << j)) continue;
                if (left[target[j] - 'a'] > 0) {
                    nxt += (1 << j);
                    left[target[j] - 'a']--;
                }
            }
            dp[nxt] = min(dp[nxt], dp[i] + 1);
        }
    }
    return dp[(1 << m) - 1] == INF ? -1 : dp[(1 << m) - 1];
}
}

```

如何优化？

上面的代码通过了测试，但时间和空间消耗均无法让人满意。让我们思考一下问题出在哪里。

考虑有 `hello` 和 `world`，目标状态是 `helloworld`。我们从 `0000000000` 开始时，既考虑了使用 `hello`，也考虑了使用 `world`。这样就更新出了 `1111100000` 和 `0000011111` 两个状态。我们会发现，它们其实是殊途同归的。第一次选 `hello`，第二次就要选 `world`；第一次选 `world`，第二次就要选 `hello`。由于我们只需要计算使用贴纸的数量，先后顺序其实并不重要，这两个状态其实是重复的。

如何消除这一重复？我们可以增加一重限制。每次从当前状态开始更新时，我们只选择包含了当前状态从左边开始第一个没有包含的字母的那些贴纸。比如说在上面的例子中，在 `0000000000` 状态下，我们将只会选择 `hello`，不会选择 `world`（没有包含 `h`）。这样就去除了顺序导致的重复状态。

为了实现这一优化，我们预处理得到了 `can` 数组，记录包含每一个字母的贴纸序号。

```

int INF = std::numeric_limits<int>::max();
int minStickers(vector<string>& stickers, string target) {
    vector<int> dp(1 << 15, INF);
    int n = stickers.size(), m = target.size();
    vector<vector<int>> cnt(n, vector<int>(26));
    vector<vector<int>> can(26);
    for (int i = 0; i < n; ++i)
        for (char c : stickers[i]) {
            int d = c - 'a';
            cnt[i][d]++;
            if (can[d].empty() || can[d].back() != i)
                can[d].emplace_back(i);
        }

    dp[0] = 0;
    for (int i = 0; i < (1 << m) - 1; ++i) {
        if (dp[i] == INF)
            continue;
        int d;
        for (int j = 0; j < m; ++j) {
            if (!(i & (1 << j))) {
                d = j;
                break;
            }
        }
        d = target[d] - 'a';
        for (int k : can[d]) {
            int nxt = i;
            vector<int> left(cnt[k]);
            for (int j = 0; j < m; ++j) {
                if (nxt & (1 << j))
                    continue;
                if (left[target[j] - 'a'] > 0) {
                    nxt += (1 << j);
                    left[target[j] - 'a']--;
                }
            }
            dp[nxt] = min(dp[nxt], dp[i] + 1);
        }
    }
    return dp[(1 << m) - 1] == INF ? -1 : dp[(1 << m) - 1];
}

```

2.0.10 1994. The Number of Good Subsets

You are given an integer array `nums`. We call a subset of `nums` good if its product can be represented as a product of one or more distinct prime numbers.

For example, if `nums = [1, 2, 3, 4]`: `[2, 3]`, `[1, 2, 3]`, and `[1, 3]` are good subsets with products $6 = 2 \cdot 3$, $6 = 2 \cdot 3$, and $3 = 3$ respectively. `[1, 4]` and `2` are not good subsets with products $4 = 2 \cdot 2$ and $4 = 2 \cdot 2$ respectively. Return the number of different good subsets in `nums` modulo $10^9 + 7$.

A subset of `nums` is any array that can be obtained by deleting some (possibly none or all) elements from `nums`. Two subsets are different if and only if the chosen indices to delete are different.

2.0.11 1494. Parallel Courses II - Hard

You are given an integer `n`, which indicates that there are `n` courses labeled from 1 to `n`. You are also given an array `relations` where `relations[i] = [prevCoursei, nextCoursei]`, representing a prerequisite relationship between course `prevCoursei` and course `nextCoursei`: course `prevCoursei` has to be taken before course `nextCoursei`. Also, you are given the integer `k`.

In one semester, you can take at most `k` courses as long as you have taken all the prerequisites in the previous semester for the courses you are taking.

Return the minimum number of semesters needed to take all courses. The testcases will be generated such that it is possible to take every course.

```

public int minNumberOfSemesters(int n, int[][] relations, int k) {
    int [] pre = new int [n]; // bitmask representing prerequisites
    int r = 1 << n;
    for (int [] v : relations) {
        int x = v[0]-1, y = v[1] - 1;
        pre[y] |= 1 << x;
    }
    int [] cnt = new int [r];
    for (int i = 0; i < r; i++) cnt[i] = Integer.bitCount(i);
    int [] dp = new int [r]; // dp[state] = minimum semesters to complete all the courses of 'state'.
    Arrays.fill(dp, n);
    dp[0] = 0;
}

```

²DEFINITION NOT FOUND.

```

for (int i = 0; i < r; i++) { // 遍历所有可能的状态，对每个状态进行最大可能地优化
    int cur = 0;
    for (int j = 0; j < n; j++) // 遍历课程，筛出当前子集 mask 下，可选课程的集合
        // if ((i & pre[j]) == pre[j]) // 这门课可选，但是可能已经选过 Can study course j next, since all required courses studied.
        if (((i >> j) & 1) == 0 && (i & pre[j]) == pre[j]) // 这门课可选，并且还没有选修过
            cur |= 1 << j;
    // cur &= ~i; // 这个最容易忽视：选过的课程是不用再选的 // Don't want to study those already studied courses.
    int next = cur; // 所有可选课程的集合
    while (next > 0) {
        if (cnt[next] <= k) // 寻找和遍历符合条件的子集
            dp[i | next] = Math.min(dp[i | next], dp[i] + 1);
        next = (next - 1) & cur; // 高效速度快：遍历现在可选课程的所有子集：Enumerate all subsets. E.g, available = 101, next: 100 -> 001 -> 000
    }
}
return dp[r-1];
}

```

- 另一种慢很多的：通过递归来优化选课时间的

算法

(状态压缩动态规划) $O(n \cdot 2^n + 3^n)$

1. 统计出每个课程的直接依赖课程的二进制掩码，记为 `pre`。
2. 设状态 $f(S)$ 表示完成掩码 S 的课程所需要的最少学期数。
3. 初始时， $f(0) = 0$ ，其余为正无穷。
4. 转移时，对于某个已经修完的课程掩码 S_0 ，求出 S_1 ，表示当前可以选择的新课程（新课程需要满足依赖条件），从 S_1 中通过递归回溯选出不多于 k 门课程，其掩码记为 S ，转移
 $f(S_0 \text{ bit or } S) = \min(f(S_0 \text{ bit or } S), f(S_0) + 1)$ 。
5. 最终答案为 $f((1 << n) - 1)$ 。

时间复杂度

- 状态数有 $O(2^n)$ 个，采用向后转移的方式，每个状态需要 $O(n)$ 的时间计算 S_1 ，同时枚举 S_1 的合法子集。
- 容易证明子集共有 $O(3^n)$ 个。
- 故总时间复杂度为 $O(n \cdot 2^n + 3^n)$ ，由于合法子集数目非常少，采用递归回溯枚举子集不会出现不合法的子集，则复杂度的常数很小。

空间复杂度

- 需要 $O(2^n)$ 的额外空间存储 `pre` 数组，系统栈和动态规划的状态。

```

public int minNumberOfSemesters(int n, int[][] relations, int k) {
    int [] pre = new int [n];
    int r = 1 << n;
    for (int [] v : relations) pre[v[1]-1] |= 1 << (v[0] - 1);
    f = new int [1 << n];
    Arrays.fill(f, Integer.MAX_VALUE);
    f[0] = 0;
    for (int i = 0; i < (1 << n); i++) {
        if (f[i] == Integer.MAX_VALUE) continue;
        int cur = 0;
        for (int j = 0; j < n; j++) {
            if (((i >> j) & 1) == 0 && ((pre[j] & i) == pre[j]))
                cur |= 1 << j;
            solve(0, 0, i, cur, k, n); // 看得很昏乎：
        }
    }
    return f[r-1];
}

int [] f; // dp
private void solve(int i, int s, int state, int cur, int k, int n) {
    if (k == 0 || i == n) { // 看得很昏乎：int i: 通过对现可选课程 cur 一位一位遍历的方式找出最优级解，速度慢
        f[state | s] = Math.min(f[state | s], f[state] + 1);
        return ;
    }
    solve(i+1, s, state, cur, k, n);
    if (((cur >> i) & 1) == 1) // int i: 遍历已选课程 state 子集下，所有可选课程的子集合，通过遍历 i 优化每门可选课的时间来优化结果
        solve(i+1, s | 1 << i, state, cur, k-1, n);
}

```

2.0.12 分析一下

- The value range from 1 to 30. If the number can be reduced to 20, an algorithm runs $O(2^{20})$ should be sufficient. So I should factorize each number to figure out how many valid number within the range first.
- There are only 18 valid numbers (can be represented by unique prime numbers)
- Represent each number by a bit mask - each bit represent the prime number
- The next step should be that categorize the input - remove all invalid numbers and count the number of 1 as we need to handle 1 separately.
- The problem is reduced to a math problem and I simply test all the combinations - $O(18 \cdot 2^{18})$
- If 1 exists in the input, the final answer will be $\text{result} * (1 \ll \text{number_of_one}) \% \text{mod}$.
- <https://leetcode.com/problems/the-number-of-good-subsets/discuss/1444183/Java-Bit-Mask-%2B-DP-Solution>

```
static int mod = (int) 1e9 + 7;
static int [] map = new int [31];
static {
    int [] prime = new int [] {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}; // length: 10
    for (int i = 2; i <= 30; ++i) {
        if (i % 4 == 0 || i % 9 == 0 || i == 25) continue;
        int mask = 0;
        for (int j = 0; j < 10; ++j)
            if (i % prime[j] == 0)
                mask |= 1 << j;
        map[i] = mask;
    }
}

public int numberOfGoodSubsets(int[] nums) {
    int n = nums.length, one = 0;
    int[] dp = new int[1024], cnt = new int[31]; // 1024 ?
    dp[0] = 1;
    for (int i : nums) {
        if (i == 1) one++;
        else if (map[i] != 0) cnt[i]++;
    }
    for (int i = 0; i < 31; ++i) {
        if (cnt[i] == 0) continue;
        for (int j = 0; j < 1024; ++j) {
            if ((j & map[i]) != 0) continue; // 含有某个公共质因子 val 子集的统计数 * 当前 val 的重复次数
            dp[j | map[i]] = (int) ((dp[j | map[i]] + dp[j] * (long) cnt[i]) % mod);
        }
    }
    long res = 0;
    for (int i : dp) res = (res + i) % mod;
    res--; // 应该是减去一个 1 吧
    if (one != 0) res = res * pow(one) % mod;
    return (int) res;
}

private long pow(int n) { // 快速幂
    long res = 1, m = 2;
    while (n != 0) {
        if ((n & 1) == 1) res = (res * m) % mod;
        m = m * m % mod;
        n >>= 1;
    }
    return res;
}
```

- 另一种方法参考一下，没有使用到快速幂，稍慢一点儿
- For each number n from 1 to 30, you can decide select it or not.
 - 1 - select any times, full permutation $\text{pow}(2, \text{cnt})$
 - 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 - select 0 or one time
 - 6, 10, 14, 15, 21, 22, 26, 30 - select if the prime factors of n not yet selected
 - others - can not select

```
long f(int n, long mask) {
    if (n > 30) return mask == 0 ? 0L : 1L;
    long rst = f(n + 1, mask) % MOD;
    if (n == 2 || n == 3 || n == 5 || n == 7 || n == 11 || n == 13 || n == 17 || n == 19 || n == 23 || n == 29)
        rst = (rst + cnts[n] * f(n + 1, mask | (1 << n))) % MOD;
    else if (n == 6)
        if ((mask & (1 << 2)) == 0 && (mask & (1 << 3)) == 0)
            rst = (rst + cnts[n] * f(n + 1, mask | (1 << 2) | (1 << 3))) % MOD;
    else if (n == 10)
```

```

    if ((mask & (1 << 2)) == 0 && (mask & (1 << 5)) == 0)
        rst = (rst + cnts[n] * f(n + 1, mask | (1 << 2) | (1 << 5))) % MOD;
    else if (n == 14)
        if ((mask & (1 << 2)) == 0 && (mask & (1 << 7)) == 0)
            rst = (rst + cnts[n] * f(n + 1, mask | (1 << 2) | (1 << 7))) % MOD;
    else if (n == 22)
        if ((mask & (1 << 2)) == 0 && (mask & (1 << 11)) == 0)
            rst = (rst + cnts[n] * f(n + 1, mask | (1 << 2) | (1 << 11))) % MOD;
    else if (n == 26)
        if ((mask & (1 << 2)) == 0 && (mask & (1 << 13)) == 0)
            rst = (rst + cnts[n] * f(n + 1, mask | (1 << 2) | (1 << 13))) % MOD;
    else if (n == 15)
        if ((mask & (1 << 3)) == 0 && (mask & (1 << 5)) == 0)
            rst = (rst + cnts[n] * f(n + 1, mask | (1 << 3) | (1 << 5))) % MOD;
    else if (n == 21)
        if ((mask & (1 << 3)) == 0 && (mask & (1 << 7)) == 0)
            rst = (rst + cnts[n] * f(n + 1, mask | (1 << 3) | (1 << 7))) % MOD;
    else if (n == 30)
        if ((mask & (1 << 2)) == 0 && (mask & (1 << 3)) == 0 && (mask & (1 << 5)) == 0)
            rst = (rst + cnts[n] * f(n + 1, mask | (1 << 2) | (1 << 3) | (1 << 5))) % MOD;
    return rst;
}
int MOD = 1_000_000_007;
long[] cnts = new long[31];
public int numberOfGoodSubsets(int[] nums) {
    for (int n : nums) cnts[n]++;
    long rst = f(1, 0L);
    for (int i = 0; i < cnts[1]; i++) // 没有快速幂, 稍慢
        rst = rst * 2 % MOD;
    return (int) rst;
}

```

- // Speed up using frequency array. $O(30 \cdot 1024 + N)$ = linear time

- 看不懂: <https://leetcode.com/problems/the-number-of-good-subsets/discuss/1444661/Java-DP-%2B-Bitmask-or>

```

private static final long MOD=(long) (1e9+7);
private static long add(long a, long b){ a+=b; return a<MOD?a:a-MOD;}
private static long mul(long a, long b){ a*=b; return a<MOD?a:a%MOD;}
private static long pow(long a, long b) {
    //a %= MOD;
    //b%=(MOD-1);//if MOD is prime
    long res = 1;
    while (b > 0) {
        if ((b & 1) == 1)
            res = mul(res, a);
        a = mul(a, a);
        b >>= 1;
    }
    return add(res, 0);
}
public int numberOfGoodSubsets(int[] nums) {
    int N = nums.length, i;
    int[] primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
    int[] mask = new int[31];
    int[] freq = new int[31];
    for (int x : nums) freq[x]++;
    for (i = 1; i <= 30; i++)
        for (int j = 0; j < primes.length; j++)
            if (i % primes[j] == 0) {
                if ((i / primes[j]) % primes[j] == 0) {
                    mask[i] = 0;
                    break;
                }
                mask[i] |= (int) pow(2, j);
            }
    long[] dp = new long[1024];
    dp[0] = 1;
    for (i = 1; i <= 30; i++) {
        if (mask[i] == 0) continue;
        for (int j = 0; j < 1024; j++)
            if ((mask[i] & j) == 0 && dp[j] > 0)
                dp[(mask[i] | j)] = add(dp[(mask[i] | j)], mul(dp[j], freq[i]));
    }
    long ans = 0;
    for (i = 1; i < 1024; i++) ans = add(ans, dp[i]);
    ans = mul(ans, pow(2, freq[1]));
    ans = add(ans, 0);
    return (int) ans;
}

```

- Java 预处理 + 暴搜

- <https://leetcode-cn.com/problems/the-number-of-good-subsets/solution/java-yu-chu-li-bao-sou-by-liin-n>

- 暂略

2.0.13 1349. Maximum Students Taking Exam - Hard

Given a $m * n$ matrix seats that represent seats distributions in a classroom. If a seat is broken, it is denoted by '#' character otherwise it is denoted by a '.' character.

Students can see the answers of those sitting next to the left, right, upper left and upper right, but he cannot see the answers of the student sitting directly in front or behind him. Return the maximum number of students that can take the exam together without any cheating being possible..

Students must be placed in seats in good condition.

```
public int maxStudents(char[][] s) {
    int m = s.length, n = s[0].length, r = 1 << n;
    int[] rowMax = new int[m+1]; // 相比于我的第一行, 他是先生成了每一行的 mask base
    for (int i = 0; i < m; i++) {
        int cur = 0;
        for (int j = 0; j < n; j++)
            cur = cur * 2 + (s[i][j] == '.' ? 1 : 0); // bug: () important
        // cur = (cur << 1) | (s[i][j] == '.' ? 1 : 0); // 上下两行: 效果一样
        rowMax[i+1] = cur;
    }
    int[][] dp = new int[m+1][r];
    for (int i = 0; i <= m; i++)
        Arrays.fill(dp[i], -1);
    dp[0][0] = 0;
    // 如果想要满足限制条件 2, 则需要第 i 排可能的 bitmask 与第 i - 1 排可能的 bitmask 进行检测
    // // upper left and upper right are valid or not
    // (mask >> 1) & prev_mask
    // mask & (prev_mask >> 1)
    // dp[r - 1][prev_mask] is valid
    // 基于以上的分析, 动规方程可以归纳为以下
    // dp[r][mask] = max(dp[r][mask], dp[r - 1][prev_mask] + bit_count(mask))
    for (int i = 1; i <= m; i++)
        for (int j = 0; j <= rowMax[i]; j++) // 遍历 curMask (的所有子集): 要保证它是 rowMax[i] 的有效子集
            if ((j & rowMax[i]) == j && (j & (j >> 1)) == 0) // 是每行可坐最多人数与布位的有效子集 (& 后 = 本身), 并保证 set 位的左右无人
                for (int k = 0; k <= rowMax[i-1]; k++) // 遍历 preMask (的所有子集) k 是 (i-1 行的) 有效子集, 并且保证了当前位的左上方、和右上方无人
                    if (dp[i-1][k] != -1 && (j & (k >> 1)) == 0 && ((j >> 1) & k) == 0)
                        dp[i][j] = Math.max(dp[i][j], dp[i-1][k] + Integer.bitCount(j));

    int max = 0;
    for (int i = 0; i < r; i++)
        max = Math.max(max, dp[m][i]);
    return max;
}
```

- 自己写的, bug 终于是被自己找出来了。。。。。。

```
public int maxStudents(char[][] seats) {
    m = seats.length;
    n = seats[0].length;
    int r = 1 << n;
    int[][] dp = new int[m+1][r];
    for (int i = 0; i <= m; i++)
        Arrays.fill(dp[i], -1);
    dp[0][0] = 0;
    for (int i = 1; i <= m; i++)
        for (int k = 0; k < r; k++) {
            if (!isValid(seats, k, i-1)) continue;
            if ((k & (k >> 1)) != 0) continue; // BUG: 要保证每个 set 过的位左右无人, 这一行不可少
            for (int j = 0; j < r; j++) {
                if (dp[i-1][j] == -1) continue; // 前一排坐位的遍历, j 可能不是有效子集, 比如坐墙上或左右有人
                if (((k >> 1) & j) == 0 && (k & (j >> 1)) == 0)
                    dp[i][k] = Math.max(dp[i][k], dp[i-1][j] + Integer.bitCount(k));
            }
        }
    int max = 0;
    for (int i = 0; i < r; i++)
        max = Math.max(max, dp[m][i]);
    return max;
}

int m, n;
private boolean isValid(char[][] arr, int v, int idx) { // 只是检查不会坐在墙上: 方法应该是对的, 但可能会比较慢
    for (int i = 0; i < n; i++)
        if (((v >> i) & 1) == 1 && arr[idx][i] != '.') return false;
    return true;
}
```

- 有个姑娘总结的 dfs 记忆化搜索

```
public int maxStudents(char[][] s) { // todo: 再好好理解一下
    // state 代表某一行安排座椅的情况, 用二进制表示
    // int state = 1;
    // // 求出 state 的最大值, 即当前行每一列都安排座位的情况
    // for (int c=0; c<s[0].length; c++){
    //     state = ((1<<c) | state);
    // }
    // 利用行数和 state 的最大值初始化记忆数组
    // memo=new int[s.length][state];
```

```

memo=new int[s.length][1 << s[0].length];
// 递归求解, 从第 0 行第 0 列开始递归
// 即求出第 0 行第 0 列以及它后面一共最多可以安排座椅的最大数量
return help(s,0,0,0);
}
int[][] memo; // 记忆数组
// r: 当前行
// c: 当前列
// rowState: 当前行安排的座位状态
// 返回值: 当前位置及之后的位置一共可以安排座位的做最大数量
int help(char[][] s, int r, int c, int rowState){
    // 如果当前是某一行的第一列
    if (c==0&&r>0){
        // 当前为第一列时, rowState 值为上一行完整的座位安排情况
        // 查看记忆数组中是否存在上一行该座位安排情况的值
        // 如果存在, 返回记忆数组中的值
        if (memo[r-1][rowState]>0) return memo[r-1][rowState];
    }
    // 当前位置字符
    char ch = s[r][c];
    // 计算下一个递归位置
    int nextR=r, nextC=c;
    // 如果当前列加一小于总列数
    if (c+1<s[0].length){
        // 下一位置列数加一, 行数不变
        nextC=c+1;
    }else{ // 如果当前列加一超出总列数
        // 下一位置列数为 0
        nextC=0;
        // 下一位置行数加一
        nextR=r+1;
    }
    // 查看当前位置是否可以安排座位
    boolean canSit=canSit(s,r,c);
    // 如果当前位置是矩阵最后一格 (递归终止条件)
    if (nextR==s.length){
        // 如果能安排座位返回 1, 否则返回 0
        return canSit?1:0;
    }
    // 第一种选择, 为当前位置安排座位
    int count1=0;
    // 如果当前位置能安排座位
    if (canSit(s,r,c)){
        // 将当前位置设置为 S, 代表安排了座位
        s[r][c]='S';
        // 递归求解下一个位置以及之后位置一共能安排的做多座位数
        // 再加上一代表当前位置安排了一个座位
        count1=1+help(s, nextR,nextC,c==0?1:((rowState<<1)|1));
        // 递归结束后, 将当前位置还原成原始字符, 以便之后查询其他递归可能
        s[r][c]=ch;
    }
    // 第二种选择, 不为当前位置安排座位
    int count2=help(s, nextR,nextC,c==0?0:((rowState<<1)|0));
    // 两种选择的最大值为当前结果
    int res = Math.max(count1,count2);
    // 如果当前是某一行的首列
    if (c==0&&r>0){
        // 将当前结果计入记忆数组
        memo[r-1][rowState]=res;
    }
    return res;
}
// 判断当前位置是否可以安排座位
boolean canSit(char[][] s, int r, int c){
    if (s[r][c]=='#') return false;
    if (r>0&&c>0) // 左前方有座位, 当前不能安排座位
        if (s[r-1][c-1]=='S') return false;
    if (r>0&&c<s[0].length-1) // 右前方有座位, 当前不能安排座位
        if (s[r-1][c+1]=='S') return false;
    // 右方有座位, 当前不能安排座位
    if (c>0)
        if (s[r][c-1]=='S') return false;
    // 因为递归顺序是从上至下, 从左至右, 因此右方和下方都是空位不需判断
    return true;
}

```

2.0.14 1434. Number of Ways to Wear Different Hats to Each Other - Hard

There are n people and 40 types of hats labeled from 1 to 40.

Given a list of list of integers hats, where hats[i] is a list of all hats preferred by the i -th person.

Return the number of ways that the n people wear different hats to each other.

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

当前 $i = 2$, $mask = 011$, 要将帽子 2 分配给人 $j = 1$, 我们就应该保证人 $j = 1$ 的前一个状态的第 j 位为 0, 也就是说, 其当前尚未分配帽子, 只有在没有分配到帽子且可以戴帽子的时候才能将帽子 i 分配给 j , 所有我们要考虑问题的子状态 ($i = i - 1 = 1$ $mask = 011 \wedge 010 = 001$), 也就是说 $dp[3][2] = dp[3][2] + dp[1][1] = 0 + 1 = 1$, 注意 $dp[][]$ 二维数组初始化均为 0, 大胆推广一下, $dp[mask][i] += dp[mask \wedge (1 \ll j)][i - 1]$, 其中 $j \in capList[i]$ 。再结合不分配帽子 i 的情况, 则状态转移方程为:

$$dp[mask][i] = \overbrace{dp[mask][i - 1]}^{\text{不分配帽子 } i \text{ 的情况}} + \underbrace{\sum_{j \in capList[i]} dp[mask \wedge (1 \ll j)][i - 1]}_{\text{分配帽子 } i \text{ 的情况}}$$

这里需要对 $mask \wedge (1 \ll j)$ 进行说明, $1 \ll j$ 就表示左移 j 位, 含义是第 j 个人戴帽子的二进制表示, 比如 $dp[3][2]$ 中, $capList[2] = \{1\}$, $j = 1$, $**1 \ll j = 010**$; 而 $mask \wedge (1 \ll j)$ 则表示 $mask$ 的第 j 位为 0 的情况, 即第 j 个人当前未戴帽子 i 的状态, 比如 $010 \wedge 011 = 001$, 我们发现异或的作用就是将第 j 位为 1 的情况变成 0, 而其他位置保持不变。

则 $i = 2$ 时的 DP Table 如下所示:

		<i>mask</i>							
		000	001	010	011	100	101	110	111
		0	1	2	3	4	5	6	7
<i>i</i>	0	1	0	0	0	0	0	0	0
	1	1	1	0	0	0	0	0	0
	2	1	1	1	1	0	0	0	0
	3	1	1	1	1	0	0	0	0
	4	1	1	1	1	0	0	0	0
	5								

	100								

```
static final int mod = (int)1e9 + 7;
public int numberWays(List<List<Integer>> hats) {
    int n = hats.size(), r = 1 << n;
    Map<Integer, List<Integer>> m = new HashMap<>(); // [k, v]: [帽子 id, List<Integer> persons]
    List<Integer> id = new ArrayList<>();
```

```

for (int i = 0; i < n; i++) {
    List<Integer> l = hats.get(i);
    for (Integer v : l) {
        m.computeIfAbsent(v, z -> new ArrayList<>()).add(i);
        if (!id.contains(v)) id.add(v);
    }
}
int [][] dp = new int [r][id.size()+1];
dp[0][0] = 1; // 初始化: dp[0][0] 状态时为 1 个方案数!
for (int j = 1; j <= id.size(); j++) // 遍历帽子 id
    for (int i = 0; i < r; i++) { // 遍历人戴帽子的子集状态 mask
        dp[i][j] = dp[i][j-1]; // 这顶帽子是可以——不分配出去——的 // 1. 第 i 个帽子不分配的情况
        List<Integer> candi = m.get(id.get(j-1)); // 分配出去: 遍历子集状态 mask 里的每个戴帽子的人, 假如第 j 顶帽子是分给这个人的方案数 // 2. 第 i 个帽子分
        for (Integer p : candi)
            if (((i >> p) & 1) == 1)
                dp[i][j] = (dp[i][j] + dp[i ^ (1 << p)][j-1]) % mod;
    }
return (int)dp[r-1][id.size()];
}

```

2.0.15 1595. Minimum Cost to Connect Two Groups of Points - Hard 自顶向下 (dfs + 记忆数组); 自底向上: DP table

You are given two groups of points where the first group has size1 points, the second group has size2 points, and size1 >= size2.

The cost of the connection between any two points are given in an size1 x size2 matrix where cost[i][j] is the cost of connecting point i of the first group and point j of the second group. The groups are connected if each point in both groups is connected to one or more points in the opposite group. In other words, each point in the first group must be connected to at least one point in the second group, and each point in the second group must be connected to at least one point in the first group.

Return the minimum cost it takes to connect the two groups.

```

// Straightforward top-down DP for the first group. At the same time, we track which elements from the second group were connected in mask.
// After finishing with the first group, we detect elements in group 2 that are still disconnected,
// and connect them with the "cheapest" node in the first group.
private int dfs(List<List<Integer>> arr, int i, int mask, int [] minCost) { // 自顶向下, 需要记忆
    if (dp[i][mask] != null) return dp[i][mask];
    // if (i == m && Integer.bitCount(mask) == n) return 0; // 这行可要可不要
    if (i == m) {
        int res = 0;
        for (int j = 0; j < n; j++)
            if ((mask & (1 << j)) == 0) res += minCost[j];
        return dp[i][mask] = res;
    }
    int res = Integer.MAX_VALUE;
    for (int j = 0; j < n; j++) // 只有暴力查找尝试了所有可能性, 才是全局最优解
        res = Math.min(res, dfs(arr, i+1, mask | (1 << j), minCost) + arr.get(i).get(j));
    return dp[i][mask] = res;
}
Integer [][] dp; // (number of points assigned in first group, bitmask of points assigned in second group).
int m, n;
public int connectTwoGroups(List<List<Integer>> cost) {
    m = cost.size();
    n = cost.get(0).size();
    dp = new Integer [m+1][1 << n]; // 右边点组过程中共有 1 << n 种状态, 但是如何知道记住右边的点分别是与左边哪个点连接起来的呢?
    int [] minCost = new int [n]; // 对右边的每个点, 它们分别与左边点连通, 各点所需的最小花费
    Arrays.fill(minCost, Integer.MAX_VALUE);
    for (int j = 0; j < n; j++)
        for (int i = 0; i < m; i++)
            minCost[j] = Math.min(minCost[j], cost.get(i).get(j));
    return dfs(cost, 0, 0, minCost);
}

```

- 动态规划, 用二进制压缩状态, 注意分析几种情况, 就能推出来正确的状态转移方程。

```

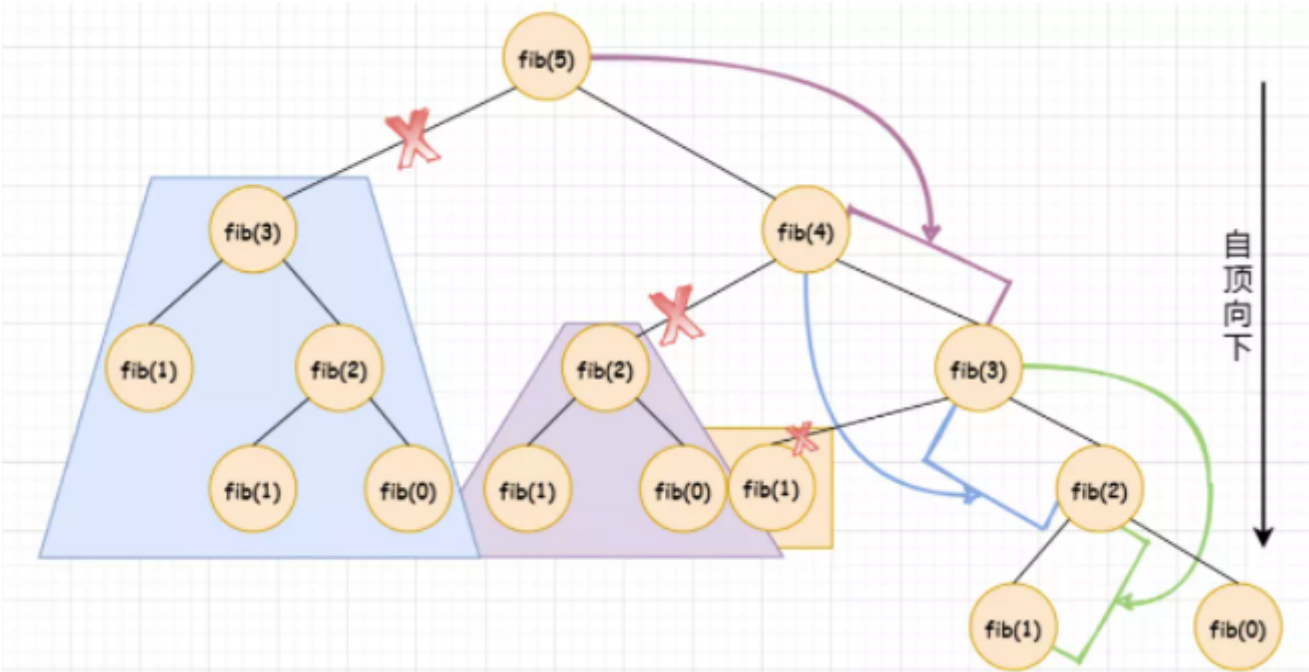
public int connectTwoGroups(List<List<Integer>> cost) {
    int m = cost.size();
    int n = cost.get(0).size();
    int [][] dp = new int [m+1][1 << n]; // 右边点组过程中共有 1 << n 种状态, 但是如何知道记住右边的点分别是与左边哪个点连接起来的呢?
    for (int i = 0; i < m; i++)
        Arrays.fill(dp[i], Integer.MAX_VALUE/2);
    for (int i = 0; i < m; i++) { // 暴力求解所有值取最小
        for (int j = 0; j < 1 << n; j++) {
            for (int k = 0; k < n; k++) {
                if (i > 0 && dp[i-1][j^(1 << k)] != Integer.MAX_VALUE/2)
                    dp[i][j] = Math.min(dp[i][j], cost.get(i).get(k) + dp[i-1][j ^ (1 << k)]);
                if (i > 0 && dp[i-1][j] != Integer.MAX_VALUE/2)
                    dp[i][j] = Math.min(dp[i][j], cost.get(i).get(k) + dp[i-1][j]);
                if (i == 0 && (j ^ (1 << k)) == 0) dp[i][j] = cost.get(i).get(k);
                else if (dp[i][j^(1 << k)] != Integer.MAX_VALUE/2)
                    dp[i][j] = Math.min(dp[i][j], cost.get(i).get(k) + dp[i][j ^ (1 << k)]);
            }
        }
    }
}

```

```
    }  
    }  
    return dp[m-1][(1 << n)-1];  
}
```

2.0.16 自顶向下与自底向上

以前写 dfs 的时候总是会忘记记忆数组，现在明白为什么需要记忆以避免重复操作



在两种方法的比较里，自顶向下最下层的操作就转化为自底向上最底层的相关处理，注意两种方法里的不同与相互转换

Chapter 3

排列与组合

3.0.1 1611. Minimum One Bit Operations to Make Integers Zero - Hard

Given an integer n , you must transform it into 0 using the following operations any number of times:

Change the rightmost (0th) bit in the binary representation of n . Change the i th bit in the binary representation of n if the $(i-1)$ th bit is set to 1 and the $(i-2)$ th through 0th bits are set to 0. Return the minimum number of operations to transform n into 0.

1. 解题思路与分析

给定一个正整数 n ，将其视为二进制数。允许进行两种操作：

- 1、将其最低位取反；
- 2、如果左起第 i 位是1，并且第 $i-1$ 位一直到第0位都是0，那就可以将第 $i+1$ 位取反。

问要将其变为0至少需要多少步。

首先，对于任意 n ，它一定要先变成形如1100...0的数，才能继续变为0。这是因为要将 n 变为0，必须其中某一步是将最高位变为0，而最高位变为0必须要得到形如1100...0的数才能做。我们来证明两个命题：

- 1、对于 $n = 2^k$ 来说，至少需要 $2^{k+1} - 1$ 步。

数学归纳法：当 $k = 0$ 时显然。假设对于 $k - 1$ 的情况上述命题也正确，对于 k 的情况，由于10...0先要变成110...0，对于任意的能把10...0变成110...0的最短的操作序列，都不会改动最高位的1（如果改变了的话，说明中途已经到了110...0了，与最短矛盾），所以把10...0变成110...0的最短的操作序列其实就是把0变为10...0的操作序列（即只考虑除了最高位1的剩余数字）。而题目的两个操作实际上是在做异或运算，这个运算是可逆的，所以把0变为10...0的最短操作序列的逆序就是把10...0变为0的最短操作序列，从而根据归纳假设，其有 $2^k - 1$ 步，接着110...0变成10...0需要1步，然后再根据归纳假设，其经过 $2^k - 1$ 步变为0，所以总共步数就是 $2^k - 1 + 2^k = 2^{k+1} - 1$ 步。由数学归纳法，命题对于任意 k 都对。

- 2、将 $n \rightarrow x$ 的最少步数等于 $n \wedge x \rightarrow x \wedge x = 0$ 的最少步数，这一点由异或运算满足结合律显然成立（左边同时做异或即可）。

我们回到原问题，设 $f(n)$ 是答案。对于 n 而言， $n \rightarrow 110...0$ 需要的步数就等于 $n \wedge 110...0 \rightarrow 110...0 \wedge 110...0 = 0$ 的步数，我们取110...0是能将 n 的最高位1通过异或消掉的数，令 k 是小于等于 n 的最大的2的幂（例如 $n = 10$ 的时候 $k = 2^3 = 8$ ， $n = 17$ 的时候 $k = 2^4 = 16$ ），那么显然 $k + k/2$ 就是形如110...0并且能将 n 的最高位1消掉的数。所以有

$$f(n) = f(n \wedge 110...0) + 1 + k - 1 = f(n \wedge 110...0) + k$$

只需要直接DFS就行了。因为每次参数的最高位都会被消掉，所以位数一定会越变越少，时间复杂度就是 $O(\log n)$ 。代码如下：

```
public int minimumOneBitOperations(int n) { // O(logN)
    if (n == 0) return 0;
    int k = 1;
    while (k << 1 <= n)
        k <<= 1;
    return minimumOneBitOperations(k ^ (k >> 1) ^ n) + k;
}
public int minimumOneBitOperations(int n) {
    int ans = 0;
    while (n > 0) {
        ans ^= n;
        n /= 2;
    }
    return ans;
}
```

3.0.2 1643. Kth Smallest Instructions - Hard

Bob is standing at cell (0, 0), and he wants to reach destination: (row, column). He can only travel right and down. You are going to help Bob by providing instructions for him to reach destination.

The instructions are represented as a string, where each character is either:

'H', meaning move horizontally (go right), or 'V', meaning move vertically (go down). Multiple instructions will lead Bob to destination. For example, if destination is (2, 3), both "HHHV" and "HVHVH" are valid instructions.

However, Bob is very picky. Bob has a lucky number k , and he wants the k th lexicographically smallest instructions that will lead him to destination. k is 1-indexed.

Given an integer array `destination` and an integer k , return the k th lexicographically smallest instructions that will take Bob to destination.

1. 解题思路与分析

方法一：优先确定高位 + 组合计数

思路与算法

当字符串中每一种字符的数量固定时（例如对于本题，我们需要在字符串中放入 h 个 H 和 v 个 V），如果要求出字典序第 k 小的字符串，可以考虑从高位向低位依次确定每一个位置的字符。

如果我们在最高位放置了 H，那么剩余的 $(h-1, v)$ 就是一个规模减少的相同问题；同理如果我们在高位放置了 V，那么剩余的 $(h, v-1)$ 也是一个规模减少的相同问题。

我们考虑最高位是放 H 还是 V。由于后者的字典序较大，因此如果最高位放 V，那么所有最高位为 H 的字符串的字典序都比它小，这样的字符串共有

$$o = \binom{h+v-1}{h-1}$$

个。也就是确定了最高位为 H，剩余 $h+v-1$ 个位置中选择 $h-1$ 个放入 H，其余位置自动放入 V 的方案数。因此：

- 如果 k 大于这个组合数 o ，那么最高位一定是 V。我们将 v 减少 1，并且需要将 k 减少 o ，这是因为剩余部分应当是包含 $(h, v-1)$ 的字典序第 $k-o$ 小的字符串；
- 如果 k 小于 o ，那么最高位是 H。我们将 h 减少 1，但我们不需要改变 k 的值，这是因为剩余部分就是包含 $(h-1, v)$ 的字典序第 k 小的字符串。

这样一来，我们就可以从高位开始，依次确定每一个位置的字符了。需要注意的是，当 $h=0$ 时，我们只能放 V，无需进行判断。

代码

对于 Python 语言，可以使用 `math.comb()` 方便地求出组合数。但对于 C++ 而言，由于本题会导致乘法溢出，因此可以考虑使用组合数的递推式

$$c[n][k] = c[n-1][k-1] + c[n-1][k]$$

预处理处所有可能需要用到的组合数。

本题中，可能需要计算的最大组合数为 $\binom{29}{14}$ ，在 C++ 语言中，直接通过先乘法后除法的方法计算该组合数，在乘法过程中就会超出 64 位无符号整数的上限。

```
public String kthSmallestPath(int[] a, int k) {
    int m = a[0], n = a[1], mn = m + n; // m rows, n cols
    int[][] c = new int[mn][mn]; // calculate combinations
    c[0][0] = 1;
    for (int i = 1; i < mn; i++) {
        c[i][0] = 1; // 从这些数量中选 0 个的可能性：1-->就是什么也不选
```

```

    for (int j = 1; j <= i && j < n; j++) // 对于第 j 个数, 有两种选择:
        c[i][j] = c[i-1][j] + c[i-1][j-1]; // 可以不选 j, 则所有选的 j 个数由前 i-1 个数选出; 或选 j, 则从前 i-1 个数中选择出 j-1 个数
}
String ans = "";
for (int i = 0; i < mn; i++) { // loop 出 m+n 步中, 根据 k 值的大小, 每一步的选择
    if (n > 0) { // 如果当前选择的是 "V": 那么所有最高位为 H 的字符串的字典序都比它小, 这样的字符串共有 cnt 种
        int cnt = c[m+n-1][n-1]; // 会有多少种选择
        if (k > cnt) { // k 比这个值大, 说明必须选 V
            ans += "V"; // 我们将 vv 减少 11, 并且需要将 kk 减少 oo, 这是因为剩余部分应当是包含 (h, v-1)(h, v1) 的字典序第 k-oko 小的字符串;
            --m; // BUG: 这里可能没有理解对, 想要减少 'V', 减少的当然是行数呀
            k -= cnt;
        } else {
            ans += "H";
            --n; // BUG: 减少 H, 减少的是列数
        }
    } else {
        ans += "V";
        --m;
    }
}
return ans;
}

```

3.0.3 1467. Probability of a Two Boxes Having The Same Number of Distinct Balls - Hard

Given $2n$ balls of k distinct colors. You will be given an integer array `balls` of size k where `balls[i]` is the number of balls of color i .

All the balls will be shuffled uniformly at random, then we will distribute the first n balls to the first box and the remaining n balls to the other box (Please read the explanation of the second example carefully).

Please note that the two boxes are considered different. For example, if we have two balls of colors a and b , and two boxes $[]$ and $()$, then the distribution $[a]$ (b) is considered different than the distribution $[b]$ (a) (Please read the explanation of the first example carefully).

We want to calculate the probability that the two boxes have the same number of distinct balls.

Example: $[2, 1, 1]$

for first ball, we can put both ball in first bin and 0 ball in second bin OR we can put 1 ball in first bin, second ball in 2nd bin OR we can put both in second bin

for second ball, we can put ball in first bin and 0 ball in second bin, similarly, we can put 1 ball in second bin.

same thing with the third ball

Try all possible permutations recursively. And, in the end check, if there are equal number of distinct balls in both bins or not.

1. 解题思路与分析

思路比较朴素, 就是先把这 $2n$ 个数字分成两块, 对应两个括号。先不考虑括号里这些数字的排列情况。对于每一种颜色的球, 可以放 $0, 1, 2, \dots$ `balls[i]` 个到第一个块。

依次类推到其他颜色的球。我们可以遍历所有的这些可能的分块方法。`balls[i] ≤ 6`, `balls.length ≤ 8`, 所以一共最多有 7^8 种方法可能不算太多。

对于上述每一种分块, 都会有 $P(\text{ballsA}) * P(\text{ballsB})$ 种可能。 $P(\text{balls})$ 就是我们下面要计算的, 给定每种球的数量, 有多少种不同的排列。

$P(\text{balls}) = \text{sum}(\text{balls})! / (\text{balls}^1! * \text{balls}^2! * \text{balls}^2! \dots \text{balls}[n-1]!);$

可以从小到大依次计算

把 $\text{balls}^4 + \text{balls}^2 + \dots \text{balls}[i-1] = \text{sum}$

$P[0 : i] = p[0 : i-1] * (\text{sum} + 1) * (\text{sum} + 2) * \dots (\text{sum} + \text{balls}[i]) / \text{factorial}[\text{balls}[i]]$

修改了下题解, 不需要用 `BigInteger` 了, java 的 `Double` 就足够大。

```

private static final double [] fact = {1, 1, 2, 6, 24, 120, 720};
private double difCnt = 0;
int n;
public double getProbability(int[] balls) { // 居然是个回壘 + 裁枝
    n = balls.length;
    double totCnt = getPermutation(balls);
    dfs(0, balls, new int [n]);
    return difCnt / totCnt;
}
private double getPermutation(int [] a) {
    double [] ans = new double [n];
    ans[0] = 1;
    int sum = a[0];
}

```

¹DEFINITION NOT FOUND.

²DEFINITION NOT FOUND.

```

    for (int i = 1; i < n; i++) {
        ans[i] = ans[i-1];
        for (int j = sum+1; j <= sum + a[i]; j++)
            ans[i] *= j;
        ans[i] /= fact[a[i]];
        sum += a[i];
    }
    return ans[n-1];
}

private void dfs(int idx, int [] a, int [] b) {
    if (idx == n) {
        int ca = 0, cb = 0, sa = 0, sb = 0;
        for (int i = 0; i < n; i++) {
            sa += a[i];
            sb += b[i];
            if (a[i] > 0) ca++;
            if (b[i] > 0) cb++;
        }
        if (ca == cb && sa == sb)
            difCnt += getPermutation(a) * getPermutation(b);
        return ;
    }
    for (int i = 0; i <= a[idx]; i++) {
        a[idx] -= i;
        b[idx] += i;
        dfs(idx+1, a, b);
        b[idx] -= i;
        a[idx] += i;
    }
}

```

- 以前参考过的题解

```

public double getProbability(int[] balls) {
    int sum = Arrays.stream(balls).sum();
    double all = allCases(balls, 0, 0, 0, 0, 0, sum);
    double valid = casesWithEqualDistinctBalls(balls, 0, 0, 0, 0, 0, sum);
    return (1.0 * valid / all);
}

// disF = distinct balls in first bin
// disS = distinct balls in second bin
// f = number of balls in first bin
// s = number of balls in second bin
public double allCases(int [] arr, int pos, int f, int s, int disF, int disS, int sum) {
    if (pos == arr.length) {
        // for all cases, we just need to check if both bins have same number of balls or not
        if (f == s) return fact(sum / 2) * fact(sum / 2); // numerator of our permutations
        return 0;
    }
    // we put all balls in second bin
    double ans = 1.0 * allCases(arr, pos+1, f, s+arr[pos], disF, disS+1, sum) / fact(arr[pos]);
    // we put all balls in first bin
    ans += 1.0 * allCases(arr, pos+1, f+arr[pos], s, disF+1, disS, sum) / fact(arr[pos]);
    for (int i = 1; i < arr[pos]; i++) // 把每一种颜色的球放到两个里面盒子里都有
        ans += 1.0 * allCases(arr, pos+1, f+i, s+arr[pos]-i, disF+1, disS+1, sum) / (fact(i) * fact(arr[pos]-i));
    return ans;
}

public double casesWithEqualDistinctBalls(int [] arr, int pos, int f, int s, int disF, int disS, int sum) {
    if (pos == arr.length) {
        if (f == s && disF == disS) return fact(sum / 2) * fact(sum / 2);
        return 0;
    }
    double ans = 1.0 * casesWithEqualDistinctBalls(arr, pos+1, f, s+arr[pos], disF, disS+1, sum) / fact(arr[pos]);
    ans += 1.0 * casesWithEqualDistinctBalls(arr, pos+1, f+arr[pos], s, disF+1, disS, sum) / fact(arr[pos]);
    for (int i = 1; i < arr[pos]; i++)
        ans += 1.0 * casesWithEqualDistinctBalls(arr, pos+1, f+i, s+arr[pos]-i, disF+1, disS+1, sum) / (fact(i) * fact(arr[pos]-i));
    return ans;
}

private double fact(double n) {
    double res = 1;
    for (int i = 2; i <= n; i++)
        res = res * i;
    return res;
}

// Complexity: There can be total of (8 * 6) balls with 8 distinct. Complexity of recursion is f * s * disF * disS = 0(48 * 48 * 8 * 8)

```

3.0.4 1569. Number of Ways to Reorder Array to Get Same BST - Hard

Given an array `nums` that represents a permutation of integers from 1 to `n`. We are going to construct a binary search tree (BST) by inserting the elements of `nums` in order into an initially empty BST. Find the number of different ways to reorder `nums` so that the constructed BST is identical to that formed from the original array `nums`.

For example, given `nums` = [2,1,3], we will have 2 as the root, 1 as a left child, and 3 as a right child. The array [2,3,1] also yields the same BST but [3,2,1] yields a different BST.

Return the number of ways to reorder nums such that the BST formed is identical to the original BST formed from nums.

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

1. 解题思路与分析: 官方题解

- <https://leetcode-cn.com/problems/number-of-ways-to-reorder-array-to-get-same-bst/solution/jiang-z>

根节点是数组第一个数

然后分为左右两个子树，左右子树之间的顺序不乱就可以

假设左子树 L 长度 n_L ，右子树 R 长度 n_R ，存在方案数为 $C_{n_L+n_R}^{n_L} * f(L) * f(R)$

```
public int numOfWays(int[] a) {
    int n = a.length;
    f = new int[n+1][n+1];
    f[1][0] = 1; // C_1^0 = 1
    f[1][1] = 1; // C_1^1 = 1
    for (int i = 2; i <= n; i++) // DP 求解组合数
        for (int j = 0; j <= i; j++)
            if (j == 0 || j == i) f[i][j] = 1; // C_n^0 = C_n^n = 1
            else f[i][j] = (f[i-1][j-1] + f[i-1][j]) % mod; // 选与不选第 j 个数
    return (int)((dfs(Arrays.stream(a).boxed().collect(Collectors.toList())) - 1) % mod);
}
int mod = (int)1e9 + 7;
int[][] f;
private long dfs(List<Integer> a) {
    if (a.size() <= 1) return 1;
    int root = a.get(0), n = a.size();
    List<Integer> l = new ArrayList<>();
    List<Integer> r = new ArrayList<>();
    for (int v : a)
        if (v < root) l.add(v);
        else if (v > root) r.add(v);
    long cntLeft = dfs(l), cntRight = dfs(r);
    return ((f[n-1][l.size()] * cntLeft % mod) * cntRight) % mod;
}
```

2. 解题思路与分析: 先根据数组 nums 把整棵二叉查找树 TT 建立出来

```
static final int mod = (int)1e9 + 7;
long[][] c;
public int numOfWays(int[] a) {
    int n = a.length;
    if (n == 1) return 0;
    c = new long[n][n];
    c[0][0] = 1;
    for (int i = 1; i < n; ++i) {
        c[i][0] = 1;
        for (int j = 1; j < n; ++j)
            c[i][j] = (c[i-1][j-1] + c[i-1][j]) % mod;
    }
    TreeNode root = new TreeNode(a[0]);
    for (int i = 1; i < n; ++i) {
        int val = a[i];
        insert(root, val);
    }
    dfs(root);
    return (root.ans - 1 + mod) % mod;
}
public void insert(TreeNode root, int v) {
    TreeNode r = root;
    while (true) {
        ++r.size;
        if (v < r.val) {
            if (r.left == null) {
                r.left = new TreeNode(v);
                return;
            }
            r = r.left;
        } else {
            if (r.right == null) {
                r.right = new TreeNode(v);
                return;
            }
            r = r.right;
        }
    }
}
public void dfs(TreeNode r) {
    if (r == null) return;
    dfs(r.left);
    dfs(r.right);
    int lsize = r.left != null ? r.left.size : 0;
    int rsize = r.right != null ? r.right.size : 0;
```

```

    int lans = r.left != null ? r.left.ans : 1;
    int rans = r.right != null ? r.right.ans : 1;
    r.ans = (int) (c[lsize + rsize][lsize] % mod * lans % mod * rans % mod);
}
class TreeNode {
    TreeNode left;
    TreeNode right;
    int val;
    int size;
    int ans;
    TreeNode(int v) {
        this.val = v;
        this.size = 1;
        this.ans = 0;
    }
}

```

- 复杂度分析

时间复杂度: $O(n^2)$

时间复杂度由以下三部分组成:

预处理组合数的时间复杂度为 $O(n^2)$

建立二叉查找树的平均时间复杂度为 $O(n \log n)$ 。但在最坏情况下,当数组 `nums` 中的数单调递增或递减时,二叉查找树退化成链式结构,建立的时间复杂度为 $O(n^2)$

动态规划的时间复杂度为 $O(n)O(n)$, 即为对二叉查找树进行遍历需要的时间。

空间复杂度: $O(n^2)$

3. 解题思路与分析: 并查集 + 乘法逆元优化

```

static final int mod = (int)1e9 + 7;
long [] fac;
long [] inv;
long [] facInv;
public int numOfWays(int[] a) { // 这个方法还要再消化一下
    int n = a.length;
    if (n == 1) return 0;
    fac = new long[n];
    inv = new long[n];
    facInv = new long[n];
    fac[0] = inv[0] = facInv[0] = 1;
    fac[1] = inv[1] = facInv[1] = 1;
    for (int i = 2; i < n; ++i) {
        fac[i] = fac[i - 1] * i % mod;
        inv[i] = (mod - mod / i) * inv[mod % i] % mod;
        facInv[i] = facInv[i - 1] * inv[i] % mod;
    }
    Map<Integer, TreeNode> found = new HashMap<Integer, TreeNode>();
    UnionFind uf = new UnionFind(n);
    for (int i = n - 1; i >= 0; --i) {
        int val = a[i] - 1;
        TreeNode node = new TreeNode();
        if (val > 0 && found.containsKey(val - 1)) {
            int lchild = uf.getroot(val - 1);
            node.left = found.get(lchild);
            node.size += node.left.size;
            uf.findAndUnite(val, lchild);
        }
        if (val < n - 1 && found.containsKey(val + 1)) {
            int rchild = uf.getroot(val + 1);
            node.right = found.get(rchild);
            node.size += node.right.size;
            uf.findAndUnite(val, rchild);
        }
        int lsize = node.left != null ? node.left.size : 0;
        int rsize = node.right != null ? node.right.size : 0;
        int lans = node.left != null ? node.left.ans : 1;
        int rans = node.right != null ? node.right.ans : 1;
        node.ans = (int) (fac[lsize + rsize] * facInv[lsize] % mod * facInv[rsize] % mod * lans % mod * rans % mod);
        found.put(val, node);
    }
    return (found.get(a[0] - 1).ans - 1 + mod) % mod;
}
class UnionFind {
    public int[] parent;
    public int[] size;
    public int[] root;
    public int n;
    public UnionFind(int n) {
        this.n = n;
        parent = new int[n];
        size = new int[n];
    }
}

```

```

    root = new int[n];
    Arrays.fill(size, 1);
    for (int i = 0; i < n; i++) {
        parent[i] = i;
        root[i] = i;
    }
}

public int findset(int x) {
    return parent[x] == x ? x : (parent[x] = findset(parent[x]));
}

public int getroot(int x) {
    return root[findset(x)];
}

public void unite(int x, int y) {
    root[y] = root[x];
    if (size[x] < size[y]) {
        int temp = x;
        x = y;
        y = temp;
    }
    parent[y] = x;
    size[x] += size[y];
}

public boolean findAndUnite(int x, int y) {
    int i = findset(x);
    int j = findset(y);
    if (i != j) {
        unite(i, j);
        return true;
    }
    return false;
}
}

class TreeNode {
    TreeNode left;
    TreeNode right;
    int size;
    int ans;
    TreeNode() {
        size = 1;
        ans = 0;
    }
}
}

```

- 复杂度分析

时间复杂度：O(n(n))。

空间复杂度：O(n)

3.0.5 903. Valid Permutations for DI Sequence - Hard

You are given a string *s* of length *n* where *s*[*i*] is either:

‘D’ means decreasing, or ‘I’ means increasing. A permutation perm of *n* + 1 integers of all the integers in the range [0, *n*] is called a valid permutation if for all valid *i*:

If *s*[*i*] = ‘D’, then perm[*i*] > perm[*i* + 1], and If *s*[*i*] = ‘I’, then perm[*i*] < perm[*i* + 1]. Return the number of valid permutations perm. Since the answer may be large, return it modulo 10⁹ + 7.

当是降序时，下一个数字不小于当前最后一个数字，反之是升序时，下一个数字小于当前最后一个数字，所以可以写出状态转移方程如下所示：

```

if (S[i-1] == 'D')    dp[i][j] += dp[i-1][k]    ( j <= k <= i )
else                 dp[i][j] += dp[i-1][k]    ( 0 <= k < j )

```

(动态规划) $O(n^3)$

题解1:动态规划。这道题首先给出了最后答案很大需要取模，所以这道题很明显是一道计数DP类的问题，接下来我们需要寻找状态表示和状态转移。很朴素的想法就是 $dp[i]$ 代表了 $[0 : i]$ 这 $i + 1$ 个元素满足前 i 个序列的方案数，但是这样，我们无法直接进行状态转移，因为下一个元素放置什么除了取决于上下降关系外，还取决于当前最后一个数字是多少。基于此，我们可以提出这样的状态表示：

$dp[i][j]$ 代表将 $[0 : i]$ 这 $i + 1$ 个元素满足前 i 个DI序列并且最后一个元素是 j 的方案数，这里很明显 $j \leq i$ 。

接下来我们考虑状态转移：

考虑 $dp[2][1]$ 代表的是前三个元素并且末尾是 1 的方案数，我们从中任取一种方案， $2, 0, 1$ ，接下来我们需要放置元素 3 了。

假设此时 $s[2] = 0$ 。这是我们显然不能直接把 3 放在后面。考虑当前最后一个元素为 1 ，那么我们可以把序列中所有大于等于 1 的元素都加上 1 得到序列 $3, 0, 2$ （这个时候是不会改变当前序列的大小关系的），这个时候，我们再把 1 添加到序列末尾得到 $3, 0, 2, 1$ 。同样的我们可以把序列中所有大于等于 0 的元素都加上 1 得到序列 $3, 1, 2$ ，这时候我们再把 0 添加到末尾，得到 $3, 1, 2, 0$ 。进一步的我们可以得到，对 $dp[i - 1][k]$ 中的所有方案，我们都可以将该序列中所有大于等于 $j (0 \leq j \leq k)$ 的元素加上 1 ，并在末尾添加一个 j 得到 $dp[i][j]$ 中的一种方案。也就是 $dp[i][j]$ 可以由所有的 $j \leq k < i$ 对应的 $dp[i - 1][k]$ 生成。

假设此时 $s[2] = 1$ 。这时候我们可以直接把 3 放在后面。考虑当前最后一个元素为 1 ，我们同样的可以把序列中所有大等于 2 的元素都加上 1 得到 $3, 0, 1$ ，再把 2 放在序列末尾得到 $3, 0, 1, 2$ 。进一步的，我们可以得到对于 $dp[i - 1][k]$ 中的所有方案，我们都可以把该序列中所有大于等于 $j (k < j \leq i)$ 的数字都加上 1 ，并在末尾添加一个 j 得到 $dp[i][j]$ 中的一种方案。也就是 $dp[i][j]$ 可以由所有的 $0 \leq k < j$ 对应的 $dp[i - 1][k]$ 生成。

基于此，我们的状态转移可以写成：

当 $s[i - 1] = 'I'$ 时， $dp(i, j) = \sum_{k=0}^{j-1} dp(i - 1, k)$

当 $s[i - 1] = 'D'$ 时， $dp(i, j) = \sum_{k=j}^{i-1} dp(i - 1, k)$

状态初始化： $dp[0][0] = 1$ ，刚开始只能把 0 放在起始位置上。

时间复杂度分析： $O(n^3)$

```
public int numPermsDISequence(String s) {
    int n = s.length(), mod = (int)1e9 + 7, res = 0;
    int [][] dp = new int [n+1][n+1];
    dp[0][0] = 1;
    for (int i = 1; i <= n; i++)
        for (int j = 0; j <= i; j++) // 考虑当前最后一个元素为 j
            if (s.charAt(i-1) == 'D')
                // 可以把序列中所有大于等于 j 的元素都加上 1 得到新序列（这个时候是不会改变当前序列的大小关系的），再把 j 添加到序列末尾得到
                for (int k = j; k <= i; k++)
                    dp[i][j] = (dp[i][j] + dp[i-1][k]) % mod;
            else // 考虑当前最后一个元素为 j
                // 把该序列中所有大于等于 j (k < j <= i) 的数字都加上 1，并在末尾添加一个 j 得到 dp[i][j] 中的一种方案
                for (int k = 0; k < j; k++) // 把序列中所有大等于 j 的元素都加上 1，再把 j 放在序列末尾得到
                    dp[i][j] = (dp[i][j] + dp[i-1][k]) % mod;
    for (int i = 0; i <= n; i++)
        res = (res + dp[n][i]) % mod;
    return (int)res;
}
```

1. 解题思路与分析: (优化版本动态规划) $O(n^2)$

题解 2: 动态规划优化。延续上述的思路，但是我们发现我们的重复计算太多了。

当 $s[i - 1] == 'D'$ 的时候：

```
dp[i][j] = dp[i - 1][j] + dp[i - 1][j + 1] + dp[i - 1][j + 2] + ... + dp[i][i - 1]
dp[i][j + 1] = dp[i - 1][j + 1] + dp[i - 1][j + 2] + ... + dp[i][i - 1]
dp[i][j] = dp[i][j + 1] + dp[i - 1][j]
```

基于此，我们需要从后往前计算，而且从 $i - 1$ 开始计算就可以了，一方面是当 $s[i - 1] = 'D'$ 的时候， i 不可能出现在最后一个位置上，同时也避免了 $j + 1$ 越界。

当 $s[i - 1] = 'I'$ 的时候

```
dp[i][j - 1] = dp[i - 1][0] + dp[i - 1][1] + ... + dp[i - 1][j - 2]
dp[i][j] = dp[i][j - 1] + dp[i - 1][j - 1]
```

基于此，我们需要从前往后计算，而且从 1 开始计算就可以了，一方面是当 $s[i - 1] = 'I'$ 的时候， 0 不可能出现在最后一个位置上，同时也避免了 $j - 1$ 越界。

```
dp[i][j] = dp[i - 1][0] + dp[i - 1][1] + ... + dp[i - 1][j - 1]
```

时间复杂度: $O(n^2)$

```
static final int mod = (int)1e9 + 7;
public int numPermsDISequence(String t) {
    int n = t.length(), ans = 0;
    char [] s = t.toCharArray();
    int [][] dp = new int [n + 1][n + 1];
    dp[0][0] = 1;
    for (int i = 1; i <= n; i++)
        if (s[i - 1] == 'D')
            for (int j = i - 1; j >= 0; j--)
                dp[i][j] = (dp[i][j + 1] + dp[i - 1][j]) % mod;
            else for (int j = 1; j <= i; j++)
                dp[i][j] = (dp[i][j - 1] + dp[i - 1][j - 1]) % mod;
    for (int i = 0; i <= n; i++)
        ans = (ans + dp[n][i]) % mod;
    return ans;
}
```

(分治/区间动态规划) $O(n^3)$

题解3:分治算法 (区间动态规划)。

状态表示: `dp[i][j]` 表示填充区间 `p[i:j]` 共 `j - i + 1` 个数的方案数。不考虑具体数字, 只考虑数字之间的大小关系。

状态初始化: `dp[i][i] = 1`, 区间内只有一个数字, 数字大小关系是唯一确定的。最终返回 `dp[0][n]` 即可。

状态转移: 我们考虑枚举当前区间最大的数可以放在哪些位置。我们知道控制区间 `dp[i][j]` 的是序列 `s[i:j - 1]`。

如果 `s[i] = 'D'`, 那么最大的元素可以放在 `p[i]` 上。 `dp[i][j] += dp[i + 1][j]`

如果 `s[j - 1] = 'I'`, 那么最大的元素可以放在 `p[j]` 上。 `dp[i][j] += dp[i][j - 1]`

每一个区间其实类似于一个摆动序列, 最大元素只可能放在峰值的位置上。所以我们枚举 `[i + 1:j - 1]` 中每一个位置 `k`。如果 `s[k - 1] = 'I' and s[k] = 'D'`, 那么当前位置就是一个峰值。那么现在数组被划分成了两个部分 `s[i:k - 1]` 和 `s[k + 1,j]`。区间长度为 `j - i + 1`, 除去最大的那个元素, 那么还剩 `j - i` 个元素, 我们需要从这 `j - i` 个元素中挑选出 `k - i` 个元素放在左半区间。这样我们既可以求解两个子区间了。

```
dp[i][j] += dp[i][k - 1] * dp[k + 1][j] * C(j - i, k - i) if s[k - 1] = 'I' and s[k] = 'D' for
```

。

这里我们介绍一下组合数的动态规划求解:

状态表示: `C[i][j]` 代表从 `i` 个数选择 `j` 个数的所有方案数。

状态初始化: `C[i][0] = 1` 代表, 从 `i` 个数选择0个数只有一种情况什么都不选。

状态转移: `C[i][j] = C[i - 1][j - 1] + C[i - 1][j]`。考虑从 `i` 个数中选择 `j` 个数的方案数, 我们考虑最后一个数字选不选, 如果最后一个数字选的话, 那么相当于还需要从前 `i - 1` 个数字中选择 `j - 1` 个数字, 如果最后一个数字不选的话, 相当于还需要从前 `i - 1` 个数字中选择 `j` 个数字。

时间复杂度分析: $O(n^3)$

```
static final int mod = (int)1e9 + 7;
public int numPermsDISequence(String t) {
    int n = t.length(), ans = 0;
    char[] s = t.toCharArray();
    long[][] dp = new long[n + 1][n + 1];
    long[][] c = new long[n + 2][n + 2];
    for (int i = 0; i <= n; i++)
        dp[i][i] = 1;
    for (int i = 0; i <= n + 1; i++) {
        c[i][0] = 1;
        for (int j = 1; j <= i; j++)
            c[i][j] = (int)((c[i - 1][j] + c[i - 1][j - 1]) % mod);
    }
    for (int len = 2; len <= n + 1; len++)
        for (int i = 0; i + len - 1 <= n; i++) {
            int j = i + len - 1;
            if (s[i] == 'D') // 如果 s[i] = 'D', 那么最大的元素可以放在 p[i] 上。dp[i][j] += dp[i + 1][j]
                dp[i][j] = (dp[i][j] + dp[i + 1][j]) % mod;
            if (s[j - 1] == 'I') // 如果 s[j - 1] = 'I', 那么最大的元素可以放在 p[j] 上。dp[i][j] += dp[i][j - 1]
                dp[i][j] = (dp[i][j] + dp[i][j - 1]) % mod;
            for (int k = i + 1; k <= j - 1; k++)
                if (s[k - 1] == 'I' && s[k] == 'D')
                    dp[i][j] = (dp[i][j] + (1L * c[len - 1][k - i] * dp[i][k - 1] % mod * dp[k + 1][j] % mod) % mod);
        }
    return (int)dp[0][n];
}
```

3.0.6 233. Number of Digit One - Hard

Given an integer n , count the total number of digit 1 appearing in all non-negative integers less than or equal to n .

1. 解题思路与分析: 递归

```
[1--9] --> 1
[10--19] --> 11
[20--29] --> 1
[30--39] --> 1
....
[90--99] --> 1
[100--199] --> 100 + count(99)
[200--299] --> count(99)
....
[900--999] --> count(99)
[1000--1999] --> 1000 + count(999) // 首先千位数上一直都是 1, 所以这里有 1000 个 1, 然后在个位, 十位和百位上的 1 就是 count(999)
[2000--2999] --> count(999)
[3000--3999] --> count(999)
...
[9000--9999] --> count(999)
```

上面的数看上去已经有点规律了, 那么我们如果要求解符合题目要求的 n 总共多少个 1,

a. 假设 $n=2345$,

那么 $f(2345) = 1000 + f(345) + f(999) * 2$

相当于 $[0-999]$ 之间的 1 会需要计数两次, 然后呢, 加上千位数的 1 会计数 1000 次。最后, 还有 345 需要计数

b. 假设 $n=3568$,

那么 $f(3568) = 1000 + f(568) + f(999) * 3$

同理, $[1000-1999]$ 在千位数上的 1 有 1000 个, 然后, $[0-999]$ 需要计数三次, 就是千位数分别数 0, 1, 2 的时候总共有三次,

最后, 千位数是 3 的时候, 568 的时候需要计数。

c. 假设 $n=1729$,

那么 $f(1729) = 729 + 1 + f(999) + f(729)$

考虑千位数的时候, 千位数数 1 的情况和前面两个情况不太一样, 这时候, 千位数是 1 的数是 $[1000-1729]$, 所以总共有 $729 + 1$ 个,

然后, 千位数数 0 的时候, $[0-999]$ 就是 $f(999)$, 千位数是 1 的时候, 还有 $f(729)$ 需要计数

```
public int countDigitOne(int n) {
    if (n <= 0) return 0;
    if (n < 10) return 1;
    int base = (int) Math.pow(10, String.valueOf(n).length() - 1);
    int fst = n / base, residual = n % base;
    if (fst == 1)
        return residual + 1 + countDigitOne(residual) + countDigitOne(base - 1);
    else
        return base + countDigitOne(residual) + fst * countDigitOne(base - 1);
}
```

2. 解题思路与分析

以算百位上 1 为例子:

假设百位上是 0, 1, 和 ≥ 2 三种情况:

case 1: $n=3141092$, $a=31410$, $b=92$. 计算百位上 1 的个数应该为 $3141 * 100$ 次。

case 2: $n=3141192$, $a=31411$, $b=92$. 计算百位上 1 的个数应该为 $3141 * 100 + (92 + 1)$ 次。

case 3: $n=3141592$, $a=31415$, $b=92$. 计算百位上 1 的个数应该为 $(3141 + 1) * 100$ 次。

所以可以将每一位归纳成这样一个公式:

$(a + 8) / 10 * m + (a \% 10 == 1) * (b + 1)$

需要注意的坑, 虽然最终结果不会超过 `int` 范围, 但是因为中间计算涉及乘法, 所以会出现溢出, 需要用 `long long` 存储中间变量。

```
public int countDigitOne(int n) { //
    long digit = 1L; // 值为 1, 10, 100, ..., 表示当前处理的十进制数位
    int ans = 0;
    while (n >= digit) { // 枚举每一位上 1 的个数
        int cnt1 = (int) (n / (digit * 10) * digit); // 由更高数位的值产生的当前数位 1 的出现次数
        int cnt2 = Math.min(Math.max((int) (n % (digit * 10) - digit + 1), 0), (int) digit); // 由更低数位的值产生的当前数位 1 的出现次数
        ans += cnt1 + cnt2;
        digit *= 10L; // 十进制数位左移 1 位
    }
    return ans;
}
```

3. 解题思路与分析: 数每一位上 1 的个数

The idea is to calculate occurrence of 1 on every digit. There are 3 scenarios, for example

if $n = xyzdabc$

and we are considering the occurrence of one on thousand, it should be:

```
(1) xyz * 1000          if d == 0
(2) xyz * 1000 + abc + 1 if d == 1
(3) xyz * 1000 + 1000   if d > 1
```

iterate through all digits and sum them all will give the final answer

```
public int countDigitOne(int n) {
    if (n <= 0) return 0;
    int q = n, x = 1, ans = 0;
    do {
        int digit = q % 10;
        q /= 10;
        ans += q * x;
        if (digit == 1) ans += n % x + 1;
        if (digit > 1) ans += x;
        x *= 10;
    } while (q > 0); // q > 0
    return ans;
}

public int countDigitOne(int n) {
    int count = 0;
    //依次考虑个位、十位、百位... 是 1
    //k = 1000, 对应于上边举的例子
    for (int k = 1; k <= n; k *= 10) {
        // xyzdabc
        int abc = n % k;
        int xyzd = n / k;
        int d = xyzd % 10;
        int xyz = xyzd / 10;
        count += xyz * k;
        if (d > 1) {
            count += k;
        }
        if (d == 1) {
            count += abc + 1;
        }
        //如果不加这句话, 虽然 k 一直乘以 10, 但由于溢出的问题
        //k 本来要大于 n 的时候, 却小于了 n 会再次进入循环
        //此时代表最高位是 1 的情况也考虑完成了
        if (xyz == 0) {
            break;
        }
    }
    return count;
}

public int countDigitOne(int n) {
    int count = 0;
    for (long k = 1; k <= n; k *= 10) {
        long r = n / k, m = n % k;
        // sum up the count of ones on every place k
        count += (r + 8) / 10 * k + (r % 10 == 1 ? m + 1 : 0);
    }
    return count;
}
```

4. 解题思路与分析

复杂度分析

时间复杂度: $O(\log n)O(\log n)$ 。nn 包含的数位个数与 nn 呈对数关系。

空间复杂度: $O(1)O(1)$ 。

```
public int countDigitOne(int n) {
    // mulk 表示 10^k
    // 在下面的代码中, 可以发现 k 并没有被直接使用到 (都是使用 10^k)
    // 但为了让代码看起来更加直观, 这里保留了 k
    long mulk = 1;
    int ans = 0;
    for (int k = 0; n >= mulk; ++k) {
        ans += (n / (mulk * 10)) * mulk + Math.min(Math.max(n % (mulk * 10) - mulk + 1, 0), mulk);
        mulk *= 10;
    }
    return ans;
}
```

Chapter 4

binary Search

4.1 LeetCode Binary Search Summary 二分搜索法小结

- <https://segmentfault.com/a/1190000016825704>
- <https://www.cnblogs.com/grandyang/p/6854825.html>

4.1.1 标准二分查找

```
public int search(int[] nums, int target) {  
    int left = 0;  
    int right = nums.length - 1;  
    while (left <= right) {  
        int mid = left + ((right - left) >> 1);  
        if (nums[mid] == target) return mid;  
        else if (nums[mid] > target)  
            right = mid - 1;  
        else  
            left = mid + 1;  
    }  
    return -1;  
}
```

循环终止的条件包括：

- 找到了目标值
- $left > right$ （这种情况发生于当 $left, mid, right$ 指向同一个数时，这个数还不是目标值，则整个查找结束。）

$left + ((right - left) \gg 1)$ 对于目标区域长度为奇数而言，是处于正中间的，对于长度为偶数而言，是中间偏左的。因此左右边界相遇时，只会是以下两种情况：

- $left/mid, right$ ($left, mid$ 指向同一个数， $right$ 指向它的下一个数)
- $left/mid/right$ ($left, mid, right$ 指向同一个数)

即因为 mid 对于长度为偶数的区间总是偏左的，所以当区间长度小于等于 2 时， mid 总是和 $left$ 在同一侧。

4.1.2 二分查找左边界

利用二分法寻找左边界是二分查找的一个变体，应用它的题目常常有以下几种特性之一：

- 数组有序，但包含重复元素
- 数组部分有序，且不包含重复元素
- 数组部分有序，且包含重复元素

1. 左边界查找类型 1

类型 1 包括了上面说的第一种，第二种情况。

既然要寻找左边界，搜索范围就需要从右边开始，不断往左边收缩，也就是说即使我们找到了 $nums[mid] == target$ ，这个 mid 的位置也不一定就是最左侧的那个边界，我们还是要向左侧查找，所以我们在 $nums[mid]$ 偏大或者 $nums[mid]$ 就等于目标值的时候，继续收缩右边界，算法模板如下：

```

public int search(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target)
            left = mid + 1;
        else
            right = mid;
    }
    return nums[left] == target ? left : -1;
}

```

返回值: `nums[left] == target ? left : -1`

与标准的二分查找不同:

首先, 这里的右边界的更新是 `right = mid`, 因为我们需要在找到目标值后, 继续向左寻找左边界。

其次, 这里的循环条件是 `left < right`。

因为在最后 `left` 与 `right` 相邻的时候, `mid` 和 `left` 处于相同的位置 (前面说过, `mid` 偏左), 则下一步, 无论怎样, `left`, `mid`, `right` 都将指向同一个位置, 如果此时循环的条件是 `left <= right`, 则需要再进入一遍循环, 此时, 如果 `nums[mid] < target` 还好说, 循环正常终止; 否则, 我们会令 `right = mid`, 这样并没有改变 `left, mid, right` 的位置, 将进入死循环。

事实上, 我们只需要遍历到 `left` 和 `right` 相邻的情况就行了, 因为这一轮循环后, 无论怎样, `left, mid, right` 都会指向同一个位置, 而如果这个位置的值等于目标值, 则它就一定是最左侧的目标值; 如果不等于目标值, 则说明没有找到目标值, 这也就是为什么返回值是 `nums[left] == target ? left : -1`。

```

public int searchInsert(int[] nums, int target) {
    int len = nums.length;
    if (nums[len - 1] < target) return len;
    int left = 0;
    int right = len - 1;
    while (left <= right) {
        int mid = (left + right) / 2;
        // 等于的情况最简单, 我们应该放在第 1 个分支进行判断
        if (nums[mid] == target) {
            return mid;
        } else if (nums[mid] < target) {
            // 题目要求我们返回大于或者等于目标值的第 1 个数的索引
            // 此时 mid 一定不是所求的左边界,
            // 此时左边界更新为 mid + 1
            left = mid + 1;
        } else {
            // 既然不会等于, 此时 nums[mid] > target
            // mid 也一定不是所求的右边界
            // 此时右边界更新为 mid - 1
            right = mid - 1;
        }
    }
    // 注意: 一定得返回左边界 left,
    // 理由是对 [1,3,5,6], target = 2, 返回大于等于 target 的第 1 个数的索引, 此时应该返回 1
    // 在上面的 while (left <= right) 退出循环以后, right < left, right = 0, left = 1
    // 根据题意应该返回 left,
    // 如果题目要求你返回小于等于 target 的所有数里最大的那个索引值, 应该返回 right
    return left;
}

```

2. 左边界查找类型 2

左边界查找的第二种类型用于数组部分有序且包含重复元素的情况, 这种条件下在我们向左收缩的时候, 不能简单的令 `right = mid`, 因为有重复元素的存在, 这会导致我们有可能遗漏掉一部分区域, 此时向左收缩只能采用比较保守的方式, 代码模板如下:

```

public int search(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target)
            left = mid + 1;
        else if (nums[mid] > target)
            right = mid;
        else
            right--;
    }
    return nums[left] == target ? left : -1;
}

```

它与类型 1 的唯一区别就在于对右侧值的收缩更加保守。这种收缩方式可以有效地防止我们一下子跳过了目标边界从而导致了搜索区域的遗漏。

4.1.3 二分查找右边界

```
public int search(int[] nums, int target) {
    int left = 0;
    int right = nums.length - 1;
    while (left < right) {
        int mid = left + ((right - left) >> 1) + 1;
        if (nums[mid] > target)
            right = mid - 1;
        else
            left = mid;
    }
    return nums[right] == target ? right : -1;
}
```

- 循环条件: $\text{left} < \text{right}$
- 中间位置计算: $\text{mid} = \text{left} + ((\text{right} - \text{left}) \gg 1) + 1$
- 左边界更新: $\text{left} = \text{mid}$
- 右边界更新: $\text{right} = \text{mid} - 1$
- 返回值: $\text{nums}[\text{right}] == \text{target} ? \text{right} : -1$

这里大部分和寻找左边界是对称着来写的，唯独有一点需要尤其注意——中间位置的计算变了，我们在末尾多加了 1。这样，无论对于奇数还是偶数，这个中间的位置都是偏右的。

对于这个操作的理解，从对称的角度看，寻找左边界的时候，中间位置是偏左的，那寻找右边界的时候，中间位置就应该偏右，但是这显然不是根本原因。根本原因是，在最后 left 和 right 相邻时，如果 mid 偏左，则 left , mid 指向同一个位置， right 指向它们的下一个位置，在 $\text{nums}[\text{left}]$ 已经等于目标值的情况下，这三个位置的值都不会更新，从而进入了死循环。所以我们应该让 mid 偏右，这样 left 就能向右移动。这也就是为什么我们之前一直强调查找条件，判断条件和左右边界的更新方式三者之间需要配合使用。

右边界的查找一般来说不会单独使用，如有需要，一般是需要同时查找左右边界。

4.1.4 二分查找左右边界

前面我们介绍了左边界和右边界的查找，那么查找左右边界就容易很多了——只要分别查找左边界和右边界就行了。

4.1.5 二分查找极值

二分查找还有一种有趣的变体是二分查找极值点，之前我们使用 $\text{nums}[\text{mid}]$ 去比较的时候，常常是和给定的目标值 target 比，或者和左右边界比较，在二分查找极值点的应用中，我们是和相邻元素去比，以完成某种单调性的检测。关于这一点，我们直接来看一个例子就明白了。

Find Peak Element

A peak element is an element that is greater than its neighbors.

Given an input array nums , where $\text{nums}[i] \neq \text{nums}[i+1]$, find a peak element and return its index.

The array may contain multiple peaks, in that case return the index to any one of the peaks is fine.

You may imagine that $\text{nums}[-1] = \text{nums}[n] = -$.

这一题的有趣之处在于他要求求一个局部极大值点，并且整个数组不包含重复元素。所以整个数组甚至可以是无序的——你可能很难想象我们可以在一个无序的数组中直接使用二分查找，但是没错！我们确实可以这么干！谁要人家只要一个局部极大值即可呢。

```
public int findPeakElement(int[] nums) {
    int left = 0;
    int right = nums.length - 1;
    while (left < right) {
        int mid = left + ((right - left) >> 1);
        if (nums[mid] < nums[mid + 1])
            left = mid + 1;
        else
            right = mid;
    }
    return left;
}

public int binarySearch2(int[] nums, int target) {
    // left 和 right 都在数组下标范围内
    // [left, right]
    int left = 0;
    int right = nums.length - 1;
    // while 循环跳出的条件是 left > right
    // 所以如果没找到 target 的话，也不需要特判了
    while (left <= right) {
```

```

    int mid = left + (right - left) / 2;
    if (nums[mid] == target) return mid;
    else if (nums[mid] < target)
        left = mid + 1;
    else
        right = mid - 1;
}
// 如果没找到就只能返回-1
return -1;
}
// 模板二，适合判断当前 index 和 index + 1 之间的关系。
// right 指针一开始的定义是在数组下标范围外的，[left, right)，所以在需要移动 right 指针的时候不能写成 right = mid。这样会遗漏掉一些下标的判断。
public int binarySearch3(int[] nums, int target) {
    // right 不在下标范围内
    // [left, right)
    int left = 0;
    int right = nums.length;
    // while 循环跳出的条件是 left == right
    // 这个模板比较适合判断当前 index 和 index + 1 之间的关系
    // left < right, example, left = 0, right = 1
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) return mid;
        else if (nums[mid] < target)
            left = mid + 1;
        else
            // 因为搜索范围是左闭右开所以这里不能-1
            right = mid;
    }
    // 最后的特判
    if (left != nums.length && nums[left] == target)
        return left;
    return -1;
}
// while 条件不满足的时候，left + 1 == right，两下标应该指向某个下标 i 和 i + 1。这样如果有什么特殊的值需要判断，应该不是 left 就是 right 了。
public int binarySearch1(int[] nums, int target) {
    // left 和 right 都在数组下标范围内
    // [left, right)
    int left = 0;
    int right = nums.length - 1;
    // 举例，start = 0, end = 3
    // 中间隔了起码有 start + 1 和 start + 2 两个下标
    // 这样跳出 while 循环的时候，start + 1 == end
    // 才有了最后的两个判断
    while (left + 1 < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) return mid;
        else if (nums[mid] < target)
            left = mid;
        else
            right = mid;
    }
    // 特判
    if (nums[left] == target) return left;
    if (nums[right] == target) return right;
    // 如果没找到就只能返回-1
    return -1;
}

```

Template #1:	Template #2:	Template #3:
<pre>// Pre-processing ... left = 0; right = length - 1; while (left <= right) { mid = left + (right - left) / 2; if (nums[mid] == target) { return mid; } else if (nums[mid] < target) { left = mid + 1; } else right = mid - 1; } ... // right + 1 == left // No more candidate</pre>	<pre>// Pre-processing ... left = 0; right = length; while (left < right) { mid = left + (right - left) / 2; if (nums[mid] < target) { left = mid + 1; } else { right = mid; } } ... // left == right // 1 more candidate // Post-Processing</pre>	<pre>// Pre-processing ... left = 0; right = length - 1; while (left + 1 < right) { mid = left + (right - left) / 2; if (num[mid] < target) { left = mid; } else { right = mid; } } ... // left + 1 == right // 2 more candidates // Post-Processing</pre>

4.1.6 第一类：需查找和目标值完全相等的数

这是最简单的一类，也是我们最开始学二分查找法需要解决的问题，比如我们有数组 [2, 4, 5, 6, 9]，target = 6，那么我们可以写出二分查找法的代码如下：

```
int find(vector<int>& nums, int target) {
    int left = 0, right = nums.size();
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) return mid;
        else if (nums[mid] < target) left = mid + 1;
        else right = mid;
    }
    return -1;
}
```

会返回 3，也就是 target 的在数组中的位置。注意二分查找法的写法并不唯一，主要可以变动地方有四处：

- 第一处是 right 的初始化，可以写成 nums.size() 或者 nums.size() - 1。
- 第二处是 left 和 right 的关系，可以写成 left < right 或者 left <= right。
- 第三处是更新 right 的赋值，可以写成 right = mid 或者 right = mid - 1。
- 第四处是最后返回值，可以返回 left, right, 或 right - 1。

- 但是这些不同的写法并不能随机的组合，像博主的那种写法，
 - 若 right 初始化为 nums.size()，那么就必须用 left < right，而最后的 right 的赋值必须用 right = mid。
 - 但是如果我们 right 初始化为 nums.size() - 1，那么就必须用 left <= right，并且 right 的赋值要写成 right = mid - 1，不然就会出错。

所以博主的建议是选择一套自己喜欢的写法，并且记住，实在不行就带简单的例子来一步一步执行，确定正确的写法也行。

第一类应用实例：

Intersection of Two Arrays

4.1.7 第二类：查找第一个不小于目标值的数，可变形为查找最后一个小于目标值的数

这是比较常见的一类，因为我们要查找的目标值不一定会在数组中出现，也有可能是跟目标值相等的数在数组中并不唯一，而是有多个，那么这种情况下 nums[mid] == target 这条判断语句就没有必要存在。比如在数组 [2, 4, 5, 6, 9] 中查找数字 3，就会返回数字 4 的位置；在数组 [0, 1, 1, 1, 1] 中查找数字 1，就会返回第一个数字 1 的位置。我们可以使用如下代码：

```
int find(vector<int>& nums, int target) {
    int left = 0, right = nums.size();
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] < target) left = mid + 1;
        else right = mid;
    }
    return right;
}
```

最后我们需要返回的位置就是 `right` 指针指向的地方。在 C++ 的 STL 中有专门的查找第一个不小于目标值的数的函数 `lower_bound`，在博主的解法中也会时不时的用到这个函数。但是如果面试的时候人家不让使用内置函数，那么我们只能老老实实写上这段二分查找的函数。

这一类可以轻松的变形为查找最后一个小于目标值的数，怎么变呢。我们已经找到了第一个不小于目标值的数，那么再往前退一位，返回 `right - 1`，就是最后一个小于目标值的数。

第二类应用实例：

Heaters, Arranging Coins, Valid Perfect Square, Max Sum of Rectangle No Larger Than K, Russian Doll Envelopes

第二类变形应用：Valid Triangle Number

4.1.8 第三类：查找第一个大于目标值的数，可变形为查找最后一个不大于目标值的数

这一类也比较常见，尤其是查找第一个大于目标值的数，在 C++ 的 STL 也有专门的函数 `upper_bound`，这里跟上面的那种情况的写法上很相似，只需要添加一个等号，将之前的 `nums[mid] < target` 变成 `nums[mid] <= target`，就这一个小小的变化，其实直接就改变了搜索的方向，使得在数组中有很多跟目标值相同的数字存在的情况下，返回最后一个相同的数字的下一个位置。比如在数组 `[2, 4, 5, 6, 9]` 中查找数字 3，还是返回数字 4 的位置，这跟上面那查找方式返回的结果相同，因为数字 4 在此数组中既是第一个不小于目标值 3 的数，也是第一个大于目标值 3 的数，所以 *make sense*；在数组 `[0, 1, 1, 1, 1]` 中查找数字 1，就会返回坐标 5，通过对比返回的坐标和数组的长度，我们就知道是否存在这样一个大于目标值的数。参见下面的代码：

```
int find(vector<int>& nums, int target) {
    int left = 0, right = nums.size();
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] <= target) left = mid + 1;
        else right = mid;
    }
    return right;
}
```

这一类可以轻松的变形为查找最后一个不大于目标值的数，怎么变呢。我们已经找到了第一个大于目标值的数，那么再往前退一位，返回 `right - 1`，就是最后一个不大于目标值的数。比如在数组 `[0, 1, 1, 1, 1]` 中查找数字 1，就会返回最后一个数字 1 的位置 4，这在有些情况下是需要这么做的。

第三类应用实例：

Kth Smallest Element in a Sorted Matrix

第三类变形应用示例：

Sqrt(x)

4.1.9 第四类：用子函数当作判断关系（通常由 `mid` 计算得出）

这是最令博主头疼的一类，而且通常情况下都很难。因为这里在二分查找法重要的比较大小的地方使用到了子函数，并不是之前三类中简单的数字大小的比较，比如 *Split Array Largest Sum* 那道题中的解法一，就是根据是否能分割数组来确定下一步搜索的范围。类似的还有 *Guess Number Higher or Lower* 这道题，是根据给定函数 `guess` 的返回值情况来确定搜索的范围。对于这类题目，博主也很无奈，遇到了只能自求多福了。

第四类应用实例：

Split Array Largest Sum, Guess Number Higher or Lower, Find K Closest Elements, Find K-th Smallest Pair Distance, Kth Smallest Number in Multiplication Table, Maximum Average Subarray II, Minimize Max Distance to Gas Station, Swim in Rising Water, Koko Eating Bananas, Nth Magical Number

4.1.10 第五类：其他（通常 `target` 值不固定）

有些题目不属于上述的四类，但是还是需要用到二分搜索法，比如这道 *Find Peak Element*，求的是数组的局部峰值。由于是求的峰值，需要跟相邻的数字比较，那么 `target` 就不是一个固定的值，而且这道题的一定要注意的是 `right` 的初始化，一定要是 `nums.size() - 1`，这是由于算出了 `mid` 后，`nums[mid]` 要和 `nums[mid+1]` 比较，如果 `right` 初始化为 `nums.size()` 的话，`mid+1` 可能会越界，从而不能找到正确的值，同时 `while` 循环的终止条件必须是 `left < right`，不能有等号。

类似的还有一道 *H-Index II*，这道题的 `target` 也不是一个固定值，而是 `len-mid`，这就很有意思了，跟上面的 `nums[mid+1]` 有异曲同工之妙，`target` 值都随着 `mid` 值的变化而变化，这里的 `right` 的初始化，一定要是 `nums.size() - 1`，而 `while` 循环的终止条件必须是 `left <= right`，这里又必须要有等号，是不是很头大 -.-!!!

其实仔细分析的话，可以发现其实这跟第四类还是比较相似，相似点是都很难 --!!!，第四类中虽然是用子函数来判断关系，但大部分时候 `mid` 也会作为一个参数带入子函数进行计算，这样实际上最终算出的值还是受 `mid` 的影响，但是 `right` 却可以初始化为数组长度，循环条件也可以不带等号，大家可以对比区别一下 ~

第五类应用实例：

Find Peak Element

H-Index II

4.2 793. Preimage Size of Factorial Zeroes Function

Let $f(x)$ be the number of zeroes at the end of $x!$. Recall that $x! = 1 * 2 * 3 * \dots * x$ and by convention, $0! = 1$.

For example, $f(3) = 0$ because $3! = 6$ has no zeroes at the end, while $f(11) = 2$ because $11! = 39916800$ has two zeroes at the end. Given an integer k , return the number of non-negative integers x have the property that $f(x) = k$.

```
private long numberOfTrailingZeros(long v) {
    long cnt = 0;
    for (; v > 0; v /= 5)
        cnt += v / 5;
    return cnt;
}
public int preimageSizeFZF(int k) {
    long left = 0, right = 5L * (k + 1);
    while (left < right) {
        long mid = left + (right - left) / 2;
        long cnt = numberOfTrailingZeros(mid);
        if (cnt == k) return 5;
        if (cnt < k) left = mid + 1;
        else right = mid;
    }
    return 0;
}
```

- 下面这种解法是把子函数融到了 `while` 循环内，使得看起来更加简洁一些，解题思路跟上面的解法一模一样，参见代码如下：

```
public int preimageSizeFZF(int k) {
    long left = 0, right = 5L * (k + 1);
    while (left < right) {
        long mid = left + (right - left) / 2, cnt = 0;
        for (long i = 5; mid / i > 0; i *= 5)
            cnt += mid / i;
        if (cnt == k) return 5;
        if (cnt < k) left = mid + 1;
        else right = mid;
    }
    return 0;
}
```

下面这种解法也挺巧妙的，也是根据观察规律推出来的，我们首先来看 x 为 1 到 25 的情况：

```
x:    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
f(x): 0 0 0 0 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 4 4 4 4 4 6
g(x): 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 2
```

这里， $f(x)$ 是表示 $x!$ 末尾零的个数，而 $g(x) = f(x) - f(x-1)$ ，其实还可以通过观察发现， $f(x) = \text{sum}(g(x))$ 。

再仔细观察上面的数字，发现 $g(x)$ 有正值的时候都是当 x 是 5 的倍数的时候，那么来专门看一下 x 是 5 的倍数时的情况吧：

```
x:    5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100 105 110 115 120 125
g(x): 1 1 1 1 2 1 1 1 1 2 1 1 1 1 2 1 1 1 1 2 1 1 1 1 3
```

仔细观察上面的红色数字， $g(x)=1$ 时，是 5 的倍数， $g(x)=2$ 时，都是 25 的倍数， $g(x)=3$ 时，是 125 的倍数，那么就有：

```
g(x) = 0    if x % 5 != 0,
g(x) >= 1   if x % 5 == 0,
g(x) >= 2   if x % 25 == 0.
```

如果继续将上面的数字写下去，就可以发现规律， $g(x)$ 按照 1 1 1 1 x 的规律重复五次，第五次的时候 x 自增 1。再继续观察：

当 $x=25$ 时， $g(x)=2$ ，此时 $K=5$ 被跳过了。

当 $x=50$ 时， $g(x)=2$ ，此时 $K=11$ 被跳过了。

当 $x=75$ 时， $g(x)=2$ ，此时 $K=17$ 被跳过了。

当 $x=100$ 时， $g(x)=2$ ，此时 $K=23$ 被跳过了。

当 $x=125$ 时， $g(x)=3$ ，此时 $K=29, 30$ 被跳过了。

进一步，可以发现如下规律：

5(=1*5), 11(=6*1+5), 17(=6*2+5), 23(=6*3+5), 29(=6*4+5), 30(=6*5), 36(=31+5), 42(=31+6+5), 48(=31+6*2+5)

这些使得 x 不存在的 K ，出现都是有规律的，它们减去一个特定的基数 $base$ 后，都是余 5，而余 1, 2, 3, 4 的，都是返回 5。那么这个基数 $base$ ，实际是 1, 6, 31, 156, ...，是由 $base = base * 5 + 1$ ，不断构成的，通过这种不断对基数取余的操作，可以最终将 K 降为小于等于 5 的数，就可以直接返回结果了，参见代码如下：

```
public int preimageSizeFZF(int k) {
    if (k < 5) return 5;
    int base = 1;
    while (base * 5 + 1 <= k)
        base = base * 5 + 1;
    if (k / base == 5) return 0;
    return preimageSizeFZF(k % base);
}
```

4.3 2040. Kth Smallest Product of Two Sorted Arrays - Hard

Given two sorted 0-indexed integer arrays `nums1` and `nums2` as well as an integer `k`, return the `k`th (1-based) smallest product of `nums1[i] * nums2[j]` where $0 \leq i < \text{nums1.length}$ and $0 \leq j < \text{nums2.length}$.

Thinking Process

- I can put the index pair for the two arrays in a priority queue and compute the answer gradually. However, the K can be up to 10^9 . This will lead to TLE.
- The element can be negative. Maybe I need to know the number of negative elements and handle 4 different combinations: (negative array 1, negative array 2), (negative array 1, positive array 2), (positive array 1, negative array 2), (positive array 1, positive array 2). At least, I can know the number of products of each combination and locate k -th product among them.
- Even though I know which combination the k -th product belongs to, it doesn't guarantee I can use the priority queue approach. I need another hint.
- Continue with above, I think I need some way to eliminate some number of products step by step to reach the goal.
- Since the array is sorted, if I turn my attention on `nums1[i] * nums2[j]`, I can know there are $j + 1$ products which are less than or equal to `nums1[i] * nums2[j]` that are generated by `nums1[i]`. Then I realize that I should try the binary search.

Algorithm

- Binary search the answer
- For each `nums1[i]`, count the number of products that are less than or equal to the current guess

```
private static long INF = (long)1e10;
public long kthSmallestProduct(int[] a, int[] b, long k) {
    int m = a.length, n = b.length;
    long lo = -INF - 1, hi = INF + 1;
    while (lo < hi) {
        long mid = lo + (hi - lo) / 2, cnt = 0;
        for (int i : a) { // 对于数组 a 中的每一个数与 b 中元素的乘积，数 <= mid 的个数，二分搜索
            if (i >= 0) {
                int l = 0, r = n - 1, p = 0;
                while (l <= r) {
                    int c = l + (r - l) / 2;
                    long mul = i * (long)b[c];
                    if (mul <= mid) {
                        p = c + 1;
                        l = c + 1;
                    } else r = c - 1;
                }
                cnt += p;
            } else { // i < 0
                int l = 0, r = n - 1, p = 0;
                while (l <= r) {
                    int c = l + (r - l) / 2;
                    long mul = i * (long)b[c];
                    if (mul <= mid) {
                        p = n - c; // i < 0, 数右边 <= mid 的个数
                        r = c - 1;
                    } else l = c + 1;
                }
                cnt += p;
            }
        }
        if (cnt >= k) hi = mid;
        else lo = mid + 1;
    }
    return lo;
}
```

- 另一种相当于换汤不换药的写法

```
private static long INF = (long)1e10;
public long kthSmallestProduct(int[] a, int[] b, long k) {
    long lo = -INF, hi = INF;
    while (lo < hi) {
        long mid = lo + hi + 1 >> 1;
        if (f(a, b, mid) < k) lo = mid;
        else hi = mid - 1;
    }
    return lo;
}
private long f(int [] a, int [] b, long mid) {
    long cnt = 0;
    for (int v : a) {
        int l = 0, r = b.length;
        if (v < 0) {
            while (l < r) {
                int m = l + r >> 1;
                if ((long)v * b[m] >= mid) l = m + 1;
                else r = m;
            }
            cnt += b.length - l;
        } else { // v >= 0
            while (l < r) {
                int m = l + r >> 1;
                if ((long)v * b[m] < mid) l = m + 1;
                else r = m;
            }
            cnt += l;
        }
    }
    return cnt;
}
```