bitMaskManipulation

deepwaterooo

2021 年 11 月 7 日

目录

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 , 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：0xFFFFFFFB。

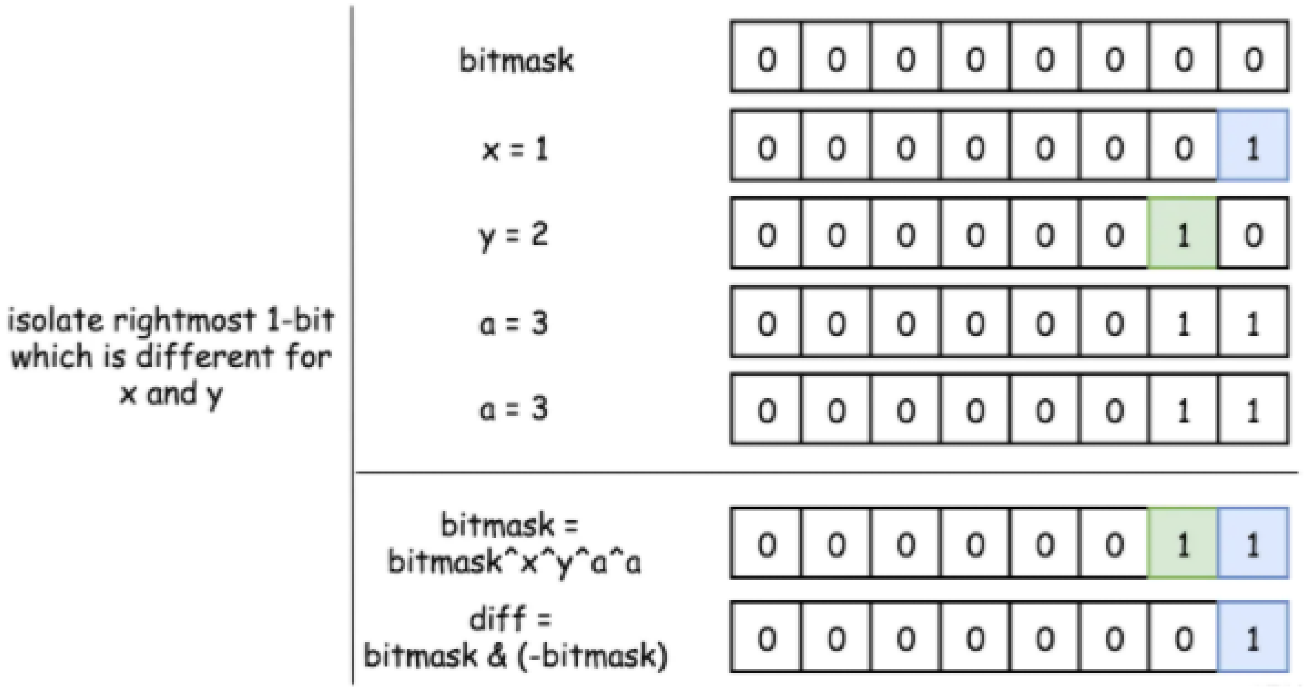
1.2 数组中不重复的两个元素

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)。



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 ˆ 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 ˆ 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有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 1111 1011

为何-5 是这样：首先把上面的 bits -1

1111 1111 1111 1111 1111 1111 1111 1010

然后再 flip 一下

0000 0000 0000 0000 0000 0000 0000 0101 = 5. 所以负数都需要先 flip 一下，然后 +1 便成了上面那样。

带负数的两数之和，有点麻烦就是有那么多 1，所以利用了自己的想象力来帮助自己理解：（不知道对不对）

举个例子：

a = -5, b = 15

把每一个 sum (a) 和 carry (b) 打出来是这样的：

```
11111111111111111111111111111011
1111
11111111111111111111111111110100
10110
11111111111111111111111111100010
101000
11111111111111111111111111001010
1000000
11111111111111111111111110001010
10000000
11111111111111111111111100001010
100000000
11111111111111111111111000001010
1000000000
11111111111111111111110000001010
10000000000
11111111111111111111100000001010
100000000000
11111111111111111111000000001010
1000000000000
11111111111111111110000000001010
10000000000000
11111111111111111100000000001010
100000000000000
11111111111111111000000000001010
1000000000000000
11111111111111110000000000001010
10000000000000000
11111111111111100000000000001010
100000000000000000
11111111111111000000000000001010
1000000000000000000
11111111111110000000000000001010
10000000000000000000
11111111111100000000000000001010
100000000000000000000
11111111111000000000000000001010
1000000000000000000000
11111111110000000000000000001010
10000000000000000000000
11111111100000000000000000001010
100000000000000000000000
11111111000000000000000000001010
1000000000000000000000000
11111110000000000000000000001010
10000000000000000000000000
11111100000000000000000000001010
100000000000000000000000000
11111000000000000000000000001010
1000000000000000000000000000
11110000000000000000000000001010
10000000000000000000000000000
11100000000000000000000000001010
100000000000000000000000000000
11000000000000000000000000001010
1000000000000000000000000000000
10000000000000000000000000001010
10000000000000000000000000000000
1010
0
10
```

我们可以看到最后是 10，在我理解，有负数的情况下，我们需要把负数的那些 1 都过滤一下，所以循环的次数会多很多，相对于正数来说。

通过上面规律，每次 a 都是减少它的 1 的数量，每次 b 都是增多它的 0 的数量，直到 a 的 1 过滤完，b 的 0 达到极限，便结束了，得到正确答案。

利用想象力的时候到了，这就相当于 a= -5 b= 15 在一个横坐标上，每一次 a 向右走一格，b 像左走一格，或者说是负数向右走，正数向左走，直到一个最小的负数走到 0，那么另外一个数就是答案。

```
public int getSum(int a, int b) {
    if (b == 0) return a;
    int sum = a ^ b;
    int carry = (a & b) << 1;
    return getSum(sum, carry);
}
public int getSum(int a, int b) { // (-1, 1) 过不了
    if (b == 0) return a;
    int carryOver = 0;
    while (b != 0) { // 这里是 b != 0, b > 0 对负数不成立
        carryOver = (a & b);
        a = a ^ b;
        b = (carryOver << 1);
    }
    return a;
}
```

1.4 201. Bitwise AND of Numbers Range

Given two integers left and right that represent the range [left, right], return the bitwise AND of all numbers in this range, inclusive.

```
public int rangeBitwiseAnd(int left, int right) {
    if (left == 0) return 0;
    if (left == 1 && right == Integer.MAX_VALUE) return 0;
    if (Integer.toBinaryString(left).length() != Integer.toBinaryString(right).length()) return 0;
    if (left == right) return left;
    int ans = left;
    for (int i = left+1; i <= right; i++) {
        ans &= i;
        if (ans == 0 || i == Integer.MAX_VALUE) return ans;
    }
    return ans;
}
```

1.5 1835. Find XOR Sum of All Pairs Bitwise AND - Hard

The XOR sum of a list is the bitwise XOR of all its elements. If the list only contains one element, then its XOR sum will be equal to this element.

For example, the XOR sum of [1,2,3,4] is equal to 1 XOR 2 XOR 3 XOR 4 = 4, and the XOR sum of [1] is equal to 3. You are given two 0-indexed arrays arr1 and arr2 that consist only of non-negative integers.

Consider the list containing the result of arr1[i] AND arr2[j] (bitwise AND) for every (i, j) pair where 0 <= i < arr1.length and 0 <= j < arr2.length.

Return the XOR sum of the aforementioned list.

```
// 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;
}
```

1.6 982. 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.

[1] DEFINITION NOT FOUND.

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

0 <= i < nums.length 0 <= j < nums.length 0 <= k < nums.length nums[i] & nums[j] & nums[k] == 0, where & represents the bitwise-AND operator.

```java
//  ‘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;
}
```

1.7 187. 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：

```java
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;
}
```

1.8 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;
}
```

1.9 782. Transform to Chessboard- Hard

You are given an n x 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，而这两种情况上各位上是相反的，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 - 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 - 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, rowDif = 0, colDif = 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;
}
```

1.10 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 <= i < j < nums.length and low <= (nums[i] XOR nums[j]) <= high.

- 1. 暴力算法

直接暴力计算，利用 num^{num2}=i 等效于 num^{i}=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 [low, 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, high] 和 [0, low - 1] 范围内数字 num2 的个数并相减，就是符合异或值为 [low, 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
    }
```

```java
}
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?!!!
}
```

1.11 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。

```java
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 ans;
}
```

1.12 957. 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.

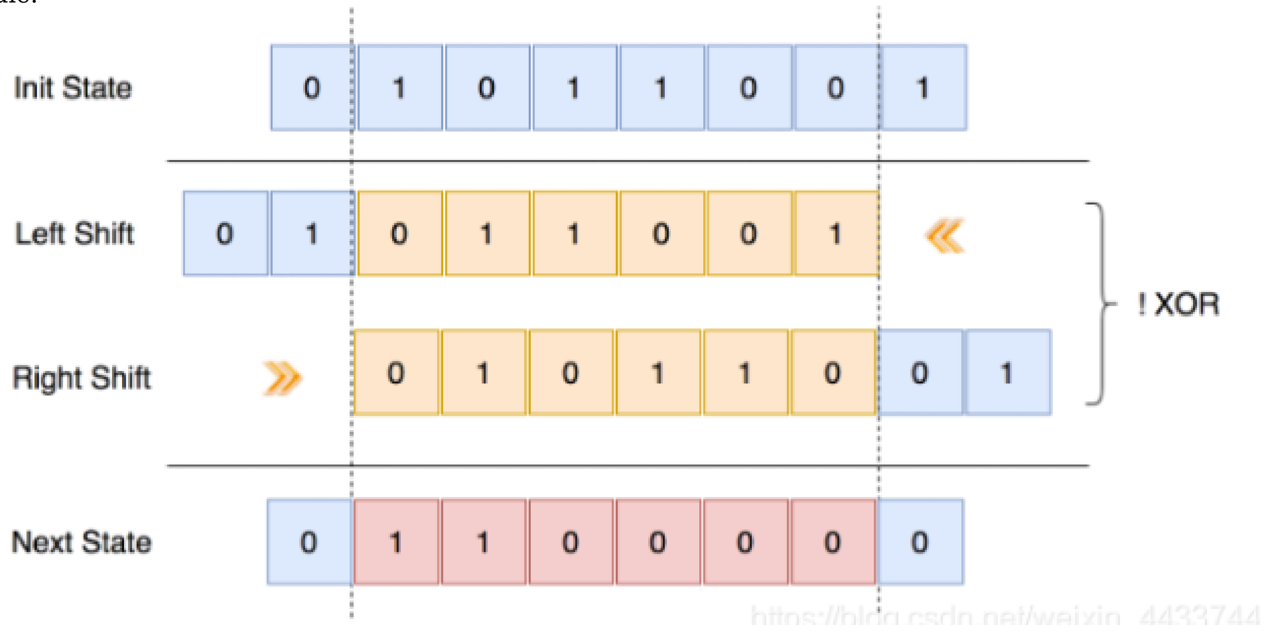
This is the method called fast-forward.

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:



```java
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

2.1 总结一下

对于一个含有 N 个元素的集合，其总共包含个子集，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：bit(i，mask) 表示取掩码的第 i 位 count(mask) 表示掩码中非零位的个数 first(mask) 表示掩码中最低非零位的数目 set(i，mask) 表示设置掩码中的第 i 位 check(i，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 « 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 « 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 & ~(1 « 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 « i)

2.2 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.

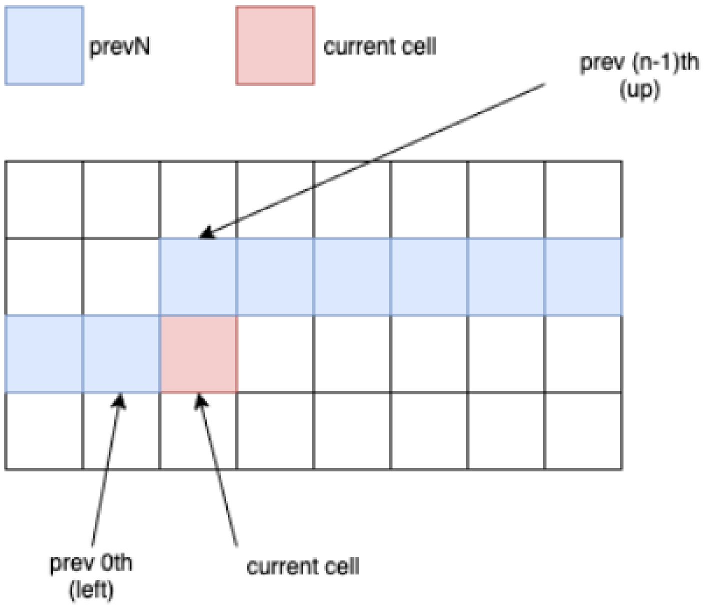
The happiness of each person is calculated as follows:

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.

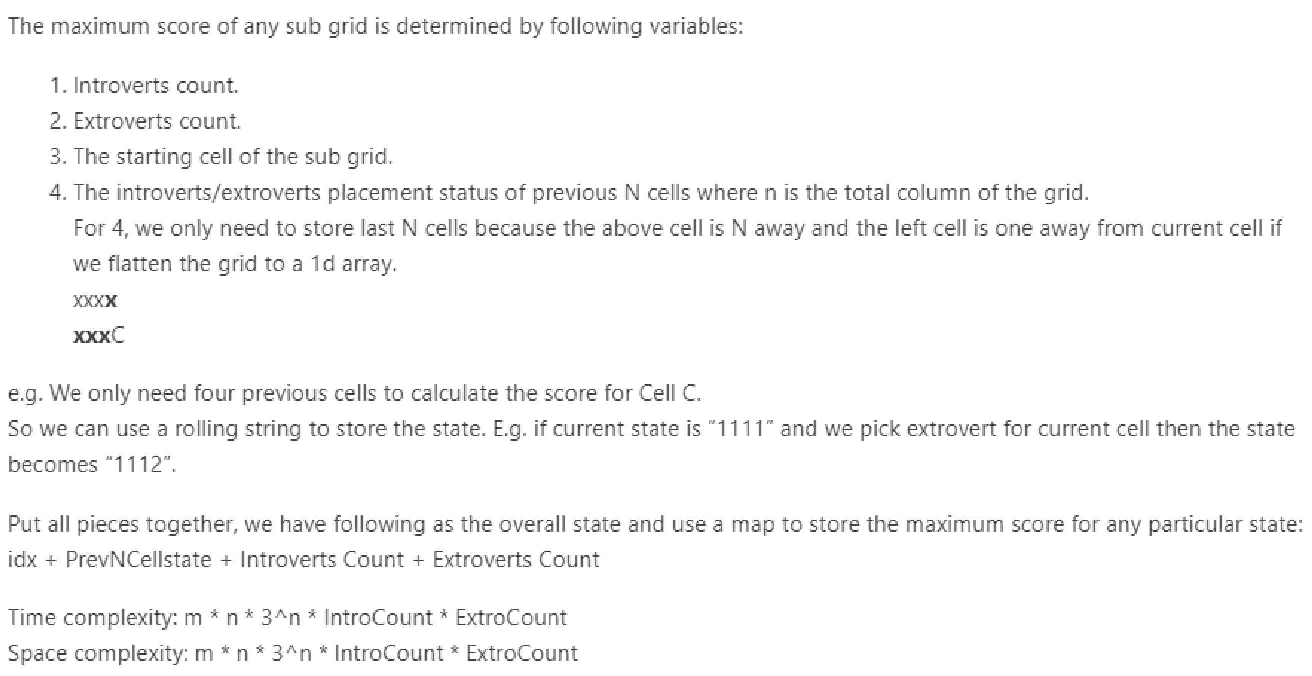
The grid happiness is the sum of each person’s happiness. Return the maximum possible grid happiness.

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 posibilities: 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.**



```java
public int getMaxGridHappiness(int m, int n, int introvertsCount, int extrovertsCount) {
    return helper(m, n, 0, 0, introvertsCount, extrovertsCount, 0, new Integer[m][n][introvertsCount+1][extrovertsCount+1][243]);
}
private int get(int preN, int i) {       // Ternary get ith bit value (0, 1 or 2)
    preN /= ((int) Math.pow(3, i));
    return preN % 3;
}
private int set(int curRow, int value) { // Ternary set new-coming bit to value
    return (curRow * 3 + value) % 243;
}                                        // Ternary bit meaning -> empty: 0, intro: 1, extro: 2
private int helper(int m, int n, int x, int y, int inCnt, int exCnt, int preN, Integer [][][][][] dp) {
    if (y == n) { // advance pointer
        y = 0;
        x++;
    }
    if (inCnt == 0 && exCnt == 0) return 0;
    if (x == m) return 0;
    if (dp[x][y][inCnt][exCnt][preN] != null) return dp[x][y][inCnt][exCnt][preN];
    int res = helper(m, n, x, y+1, inCnt, exCnt, set(preN, 0), dp); // leave the cell empty
    int up = get(preN, n-1); // get up bit -> which is at (n - 1)th
    int left = get(preN, 0); // get left bit -> which is at (0)th
    if (inCnt > 0) {
        int cur = preN;
        cur = set(cur, 1); // set new-coming bit to 1
        int addOn = 120;
        if (x-1 >= 0 && up != 0) { // up: 1 or 2
            addOn -= 30;
            if (up == 1) addOn -= 30;
            else addOn += 20;
        }
        if (y-1 >= 0 && left != 0) {
            addOn -= 30;
            if (left == 1) addOn -= 30;
            else addOn += 20;
        }
        res = Math.max(res, helper(m, n, x, y+1, inCnt-1, exCnt, cur, dp) + addOn);
    }
    if (exCnt > 0) {
        int cur = preN;
        cur = set(cur, 2); // set new-coming bit to 1
        int addOn = 40 ;
        if (x-1 >= 0 && up != 0) { // up: 1 or 2
            addOn += 20;
            if (up == 1) addOn -= 30;
            else addOn += 20;
        }
        if (y-1 >= 0 && left != 0) {
            addOn += 20;
            if (left == 1) addOn -= 30;
            else addOn += 20;
        }
        res = Math.max(res, helper(m, n, x, y+1, inCnt  , exCnt-1, cur, dp) + addOn);
    }
    return dp[x][y][inCnt][exCnt][preN] = res;
}
```



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.
   XXX**X**
   xxxC

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 * IntroCount * ExtroCount
Space complexity: m * n * 3^n * IntroCount * 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

2.3 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 prerequirements
    for (int i = 0; i < relations.length; i++) {
        int u = relations[i][0] - 1;
        int v = relations[i][1] - 1;
        pre[v] |= (1 << u);
    }
    int range = (1 << n);
    int [] cnt = new int [range]; // 所有的状态只数一遍位数
    for (int i = 0; i < range; i++)
        cnt[i] = Integer.bitCount(i);
    int [] dp = new int [range];  // dp[state] = minimum semesters to complete all the courses of 'state'.
    Arrays.fill(dp, n);
    dp[0] = 0;
    for (int i = 0; i < range; i++) {
        int available = 0;
        for (int j = 0; j < n; j++)
            if ((i & pre[j]) == pre[j]) // 可是包含了忆经选过的课程
                available |= (1 << j);  // Can study course j next, since all required courses have been studied.
        available &= ~i; // Don't want to study those already studied courses.
        int nextCourses = available;
        while (nextCourses > 0) {
            if (cnt[nextCourses] <= k)
                dp[i | nextCourses] = Math.min(dp[i | nextCourses], dp[i] + 1);
// 遍历现在可选课程的所有子集: Enumerate all subsets. E.g, available = 101, next: 100 -> 001 -> 000
            nextCourses = (nextCourses -1) & available;
        }
    }
     return dp[range-1];
}
```

- 另一种用以 ++ 写的递归的

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。

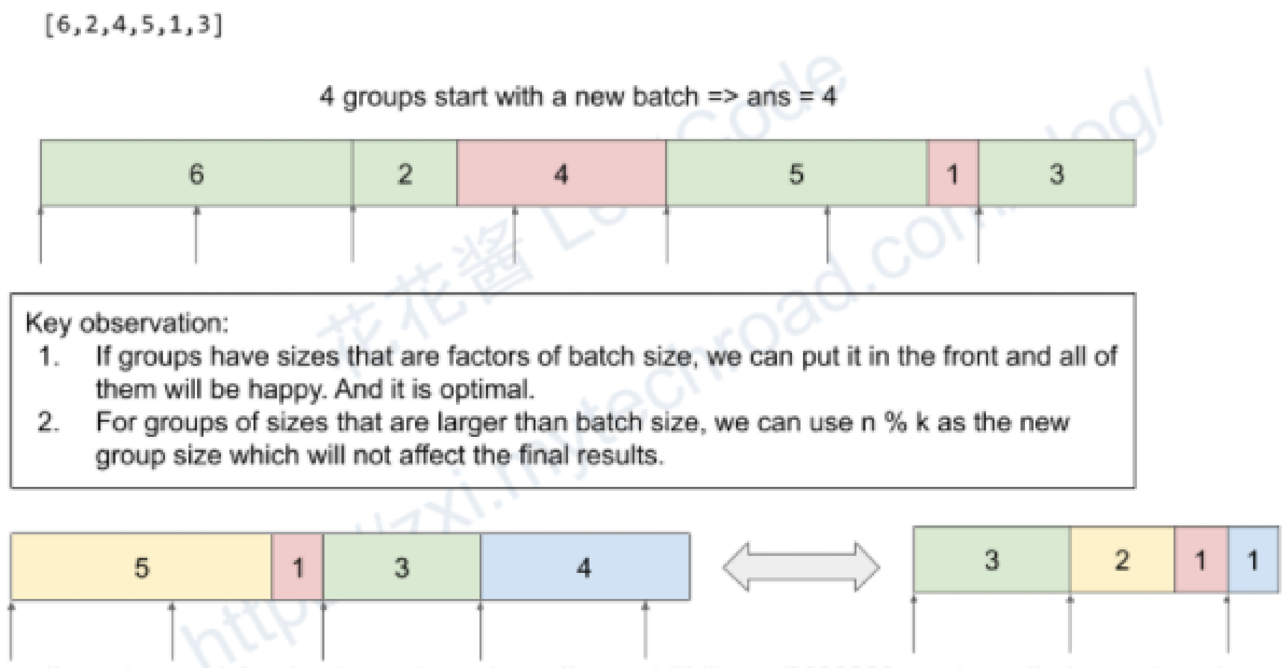
```cpp
void solve(int i, int s, int k, int n, int s0, int s1, vector<int> &f) {
    if (k == 0 || i == n) {
        f[s0 | s] = min(f[s0 | s], f[s0] + 1);
        return;
    }
    solve(i + 1, s, k, n, s0, s1, f);
    if ((s1 >> i) & 1)
        solve(i + 1, s | 1 << i, k - 1, n, s0, s1, f);
}
int minNumberOfSemesters(int n, vector<vector<int>>& dependencies, int k) {
    vector<int> pre(n, 0);
    for (const auto &v : dependencies)
        pre[v[1] - 1] |= 1 << (v[0] - 1);
    vector<int> f(1 << n, INT_MAX);
    f[0] = 0;
    for (int s0 = 0; s0 < (1 << n); s0++) {
        if (f[s0] == INT_MAX)
            continue;
        int s1 = 0;
        for (int i = 0; i < n; i++)
            if (!((s0 >> i) & 1) && ((pre[i] & s0) == pre[i]))
                s1 |= 1 << i;
        solve(0, 0, k, n, s0, s1, f);
    }
    return f[(1 << n) - 1];
}
```

2.4 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.



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.

```java
private int dfsBackTracking(int [] cnt, int batchSize, int lastGroup, int leftOverGroups) { // DFS+ 记忆化搜索求最多开心组数
    if (leftOverGroups == 0) return 0;
    String key = Arrays.toString(cnt); // 剩余组情况生成 String 作为哈希表的键, this is so called HASHING
    if (dp.containsKey(key)) return dp.get(key);
    int res = 0;
    for (int i = 1; i < batchSize; i++) {
        if (cnt[i] == 0) continue;
        --cnt[i];
        res = Math.max(res, dfsBackTracking(cnt, batchSize, lastGroup+i, leftOverGroups-1) + (lastGroup % batchSize == 0 ? 1 : 0));
        ++cnt[i];
    }
    dp.put(key, res);
    return res;
}
Map<String, Integer> dp;
int n;
public int maxHappyGroups(int batchSize, int[] groups) { // group size 30 is too large for backtracking WITHOUT modifications
    n = groups.length;
    int [] cnt = new int [batchSize];
    for (int v : groups)
        cnt[v % batchSize]++;
    dp = new HashMap<>();
    return dfsBackTracking(cnt, batchSize, 0, n-cnt[0]) + cnt[0];
}
```

- 一点儿稍微的优化，以减少回溯 backtracking 的耗时

```java
private int dfsBackTracking(int [] cnt, int batchSize, int lastGroup, int leftOverGroups) { // DFS+ 记忆化搜索求最多开心组数
    if (leftOverGroups == 0) return 0;
    String key = Arrays.toString(cnt); // 剩余组情况生成 String 作为哈希表的键, this is so called HASHING
    if (dp.containsKey(key)) return dp.get(key);
    int res = 0;
    for (int i = 1; i < batchSize; i++) {
        if (cnt[i] == 0) continue;
        --cnt[i];
        res = Math.max(res, dfsBackTracking(cnt, batchSize, lastGroup+i, leftOverGroups-1) + (lastGroup % batchSize == 0 ? 1 : 0));
        ++cnt[i];
    }
    dp.put(key, res);
    return res;
}
Map<String, Integer> dp;
int n;
public int maxHappyGroups(int batchSize, int[] groups) { // group size 30 is too large for backtracking WITHOUT modifications
    n = groups.length;
    int [] cnt = new int [batchSize];
    for (int v : groups)
        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 res = cnt[0], min = 0;
    for (int i = 1; i < batchSize/2; i++) {
        min = Math.min(cnt[i], cnt[batchSize-i]);
        cnt[i] -= min;
        cnt[batchSize-i] -= min;
        res += min;
    }
    dp = new HashMap<>();
    return res + dfsBackTracking(cnt, batchSize, 0, n-cnt[0]);
}
```

- 压缩方法里的参数

```java
private int dfsBackTracking(int [] cnt, int batchSize, int s) { // DFS+ 记忆化搜索
    String key = Arrays.toString(cnt);
    if (dp.containsKey(key)) return dp.get(key);
    int res = 0;
    for (int i = 1; i < batchSize; i++) {
        if (cnt[i] == 0) continue;
        --cnt[i];
        res = Math.max(res, dfsBackTracking(cnt, batchSize, (s+i) % batchSize) + (s == 0 ? 1 : 0));
        ++cnt[i];
    }
    dp.put(key, res);
    return res;
}
Map<String, Integer> dp;
int n;
public int maxHappyGroups(int batchSize, int[] groups) {
    n = groups.length;
    int [] cnt = new int [batchSize];
    for (int v : groups)
        cnt[v % batchSize]++;
    int res = cnt[0], min = 0;
    for (int i = 1; i < batchSize/2; i++) {
        min = Math.min(cnt[i], cnt[batchSize-i]);
        cnt[i] -= min;
        cnt[batchSize-i] -= min;
        res += min;
    }
    dp = new HashMap<>();
    return res + dfsBackTracking(cnt, batchSize, 0);
}
```

- 另一种 DP 超时的做法，掌握这个方法

```java
// Time complexity: O(n*2n) TLE TLE TLE
// Space complexity: O(2n)  TLE TLE TLE
public int maxHappyGroups(int batchSize, int[] groups) {
    int n = groups.length;
    int [] dp = new int [1 << n];
    int s = 0;
    for (int mask = 0; mask < 1 << n; mask++) {
        s = 0;
        for (int i = 0; i < n; i++)
            if ((mask & (1 << i)) >= 1)
                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];
}
```

2.5 691. 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。

```java
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

```java
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 000...000 这一状态开始，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-luc

```cpp
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。

```cpp
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];
}
```

2.6 1723. Find Minimum Time to Finish All Jobs

You are given an integer array jobs, where jobs[i] is the amount of time it takes to complete the ith job.

There are k workers that you can assign jobs to. Each job should be assigned to exactly one worker. The working time of a worker is the sum of the time it takes to complete all jobs assigned to them. Your goal is to devise an optimal assignment such that the maximum working time of any worker is minimized.

Return the minimum possible maximum working time of any assignment.

```
private void dfs(int [] arr, int k, int idx) {
    if (Arrays.stream(dp).max().getAsInt() >= min) return;
    if (idx < 0) {
        min = Math.min(min, Arrays.stream(dp).max().getAsInt());
        return;
    }
    for (int i = 0; i < k; i++) {
        if (i > 0 && dp[i] == dp[i-1]) continue;
        dp[i] += arr[idx];
        dfs(arr, k, idx-1);
        dp[i] -= arr[idx];
    }
}
int sum, min, n;
int [] dp;
public int minimumTimeRequired(int[] jobs, int k) {
    n = jobs.length;
    sum = Arrays.stream(jobs).sum();
    min = sum;
    dp = new int [k];
    Arrays.sort(jobs);
    dfs(jobs, k, n-1);
    return min;
}
```

2.7 1986. Minimum Number of Work Sessions to Finish the Tasks

There are n tasks assigned to you. The task times are represented as an integer array tasks of length n, where the ith task takes tasks[i] hours to finish. A work session is when you work for at most sessionTime consecutive hours and then take a break. You should finish the given tasks in a way that satisfies the following conditions: If you start a task in a work session, you must complete it in the same work session. You can start a new task immediately after finishing the previous one. You may complete the tasks in any order. Given tasks and sessionTime, return the minimum number of work sessions needed to finish all the tasks following the conditions above.

The tests are generated such that sessionTime is greater than or equal to the maximum element in tasks[i].

```
private void dfs(int [] arr, int t, int i, int cnt) { // cnt: sessionCnt
    if (cnt > res) return;
    if (i < 0) {
        res = Math.min(res, cnt);
        return;
    }
    for (int j = 0; j < cnt; j++)
        if (sessions[j] + arr[i] <= t) { // 把当前 task 放入旧的 sessions 里
            sessions[j] += arr[i];
            dfs(arr, t, i-1, cnt);
            sessions[j] -= arr[i];
        }
    sessions[cnt] += arr[i]; // 把当前 task 放入新的 sessions 里
    dfs(arr, t, i-1, cnt + 1);
    sessions[cnt] -= arr[i];
}
int [] sessions;
int n, res;
```

```java
public int minSessions(int[] tasks, int sessionTime) {
    n = tasks.length;
    res = n;
    sessions = new int [n];
    Arrays.sort(tasks);
    dfs(tasks, sessionTime, n-1, 0);
    return res;
}
```

- 另一种写法

```java
private int [] getMin(int [] a, int [] b) { // 这个题最近需要再写一遍
    if (a[0] > b[0]) return b;
    if (a[0] < b[0]) return a;
    if (a[1] > b[1]) return b;
    return a;
}
// dp[mask] = {a, b} where
// a - minimum number of session
// b - minimum time of last session
// The idea is to go through all tasks who belong to mask and optimally choose the last task 't' that was added to last session.
public int minSessions(int[] tasks, int sessionTime) {
    int n = tasks.length;
    int [][] dp = new int [1 << n][2];  // 在 [1, 1 << n) 范围内枚举每一个 mask 计算其包含的时间的总和
    dp[0][0] = 1;
    dp[0][1] = 0;
    for (int i = 1; i < 1 << n; i++) {
        dp[i][0] = Integer.MAX_VALUE;
        dp[i][1] = 0;
        int sum = 0;
        for (int t = 0; t < n; t++) {
            if ((i & (1 << t)) == 0) continue;
            int [] pre = dp[(1 << t) ^ i];
            if (pre[1] + tasks[t] <= sessionTime)
                dp[i] = getMin(dp[i], new int [] {pre[0], pre[1] + tasks[t]});
            else dp[i] = getMin(dp[i], new int []{pre[0]+1, tasks[t]});
        }
    }
    return dp[(1 << n) -1][0];
}
```

2.8 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 ith customer ordered. Determine if it is possible to distribute nums such that:

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

```java
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

```java
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;
}
```

2.9 956. Tallest Billboard: 这个题是最高挑战

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.

```java
// // https://blog.csdn.net/luke2834/article/details/89457888 // 这个题目要多写几遍
public int tallestBillboard(int[] rods) { // 写得好神奇呀
    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, 0,  1, -1, -1, -1, -1, -1, -1,
i: 2
-1, -1, -1,  3, 2,  3,  0,  3,  2,  3, -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

```java
// 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:
```

```java
//     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
```

```java
// 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之差
    //         新建一个 HashMap temp 用于存放第 m 个数对之前子序列累加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 对应的最大累加和
    //         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
    //         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
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);
}
```

2.10 1681. Minimum Incompatibility

You are given an integer array nums and an integer k. You are asked to distribute this array into k subsets of equal size such that there are no two equal elements in the same subset.

A subset's incompatibility is the difference between the maximum and minimum elements in that array.

Return the minimum possible sum of incompatibilities of the k subsets after distributing the array optimally, or return -1 if it is not possible.

A subset is a group integers that appear in the array with no particular order.

- Java O(k^n) solution with early termination (9ms 98%)

This problem is asking us to do the reversal of "merging k sorted lists into one sorted list".

In other words, considering we are "distributing a sorted list to k sorted lists".

The time complexity is O(k^n) since each number can have k choices.

```
private void backTracking(int [] arr, int k, int idx, int total) {
    if (total >= min) return; // early termination
    if (idx == n) {
        min = total; // With early termination, Math.min() is no longer needed.
        return;
    }
    for (int i = 0; i < dp.size(); i++) {
        LinkedList<Integer> bucket = dp.get(i);
        int dist = 0;
        if (bucket.size() < n/k && bucket.peekLast() < arr[idx]) {
            dist = arr[idx] - bucket.peekLast(); // ......
            bucket.addLast(arr[idx]);
            backTracking(arr, k, idx+1, total + dist);
            bucket.removeLast();
        }
    }
    if (dp.size() < k) { // 记住这个分组，总是有可以多分出一个组的情况需要考虑到
        LinkedList<Integer> bucket = new LinkedList<>();
        bucket.add(arr[idx]);
        dp.addLast(bucket);
        backTracking(arr, k, idx+1, total);
        dp.removeLast();
    }
}
int min = Integer.MAX_VALUE;
LinkedList<LinkedList<Integer>> dp;
int n;
public int minimumIncompatibility(int[] arr, int k) {
    n = arr.length;
    dp = new LinkedList<>();
    Arrays.sort(arr);
    backTracking(arr, k, 0, 0);
    return min == Integer.MAX_VALUE ? -1 : min;
}
```

- Optimized version (9ms): Replacing LinkedList with int[], where int[]{length, tail element} represents a sorted list/bucket since we only need to remember the length and the tail element of each sorted list.

```
int ans = Integer.MAX_VALUE;
void helper(int[] nums, int s, int[][] buckets, int idx, int size, int total) {
    if(total >= ans) return; //early termination
    if (s == nums.length) {
        ans = total; // With early termination, Math.min() is no longer needed.
    } else {
        // distribute current number to an existing bucket
        for (int i=0; i<idx; i++) {
            if (buckets[i][0] < size && buckets[i][1] < nums[s]) {
                int distance = nums[s] - buckets[i][1];
                int last = buckets[i][1];
                buckets[i][0]++;
                buckets[i][1] = nums[s];
                helper(nums, s+1, buckets, idx, size, total+distance);
                buckets[i][0]--;
                buckets[i][1] = last;
            }
        }
        // distribute current number to an empty bucket
        if (buckets.length > idx) {
            buckets[idx][0] = 1;
            buckets[idx][1] = nums[s];
            helper(nums, s+1, buckets, idx+1, size, total);
            buckets[idx][0] = 0;
        }
    }
}
public int minimumIncompatibility(int[] nums, int k) {
    Arrays.sort(nums);
    helper(nums, 0, new int[k][2], 0, nums.length/k, 0);
    return ans == Integer.MAX_VALUE?-1: ans;
}
```

2.11 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*3, 6 = 2*3, and 3 = 3 respectively. [1, 4] and [1] are not good subsets with products 4 = 2*2 and 4 = 2*2 respectively. Return the number of different good subsets in nums modulo 109 + 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.

2.11.1 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 factorialize 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*2^{18})
- If 1 exists in the input, the final answer will be result * (1 « $number_{of one}$) % 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) {
```

[1]DEFINITION NOT FOUND.

```
        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 pow(2, 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*1024 + N) = linear time

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

```java
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-1iin-n
- 暂略

2.12 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.

```java
public int maxStudents(char[][] seats) {
    int m = seats.length;
    int n = seats[0].length;
    int range = 1 << n, mask = 0;
    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++) {
        mask = 0;
        for (int j = 0; j < n; j++)
            mask = mask * 2 + (seats[i][j] == '.' ? 1 : 0);
        rowMax[i+1] = mask;
    }
    int [][] dp = new int [m+1][range];
    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 curMask = 0; curMask <= rowMax[i]; curMask++)
            if ((curMask & rowMax[i]) == curMask && (curMask & (curMask >> 1)) == 0) // 现行所有的有效掩码：既不会坐墙上，也左右无人
                for (int preMask = 0; preMask < range; preMask++)
                    if (dp[i-1][preMask] != -1 && (curMask & (preMask >> 1)) == 0 && ((curMask >> 1) & preMask) == 0)
                        dp[i][curMask] = Math.max(dp[i][curMask], dp[i-1][preMask] + Integer.bitCount(curMask));

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

- 自己写的，bug 还有找出来。。。。

```
private boolean isValid(char [][] arr, int v, int idx) { // 自己写的，不知道自己写的错哪里了，改天把它找出来
    for (int i = 0; i < n; i++)
        if (((v >> i) & 1) == 1 && arr[idx][i] != '.') return false;
    return true;
}
int m, n;
public int maxStudents(char[][] seats) {
    m = seats.length;
    n = seats[0].length;
    int range = 1 << n;
    int [][] dp = new int [m+1][range];
    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 < range; k++) { // cur mask == k
            if (!isValid(seats, k, i-1)) continue;
            for (int j = 0; j < range; j++) { // pre mask == j
                if (dp[i-1][j] == -1) continue;
                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 = 1; i < range; i++)
        max = Math.max(max, dp[m][i]);
    return max;
}
```

2.13 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.

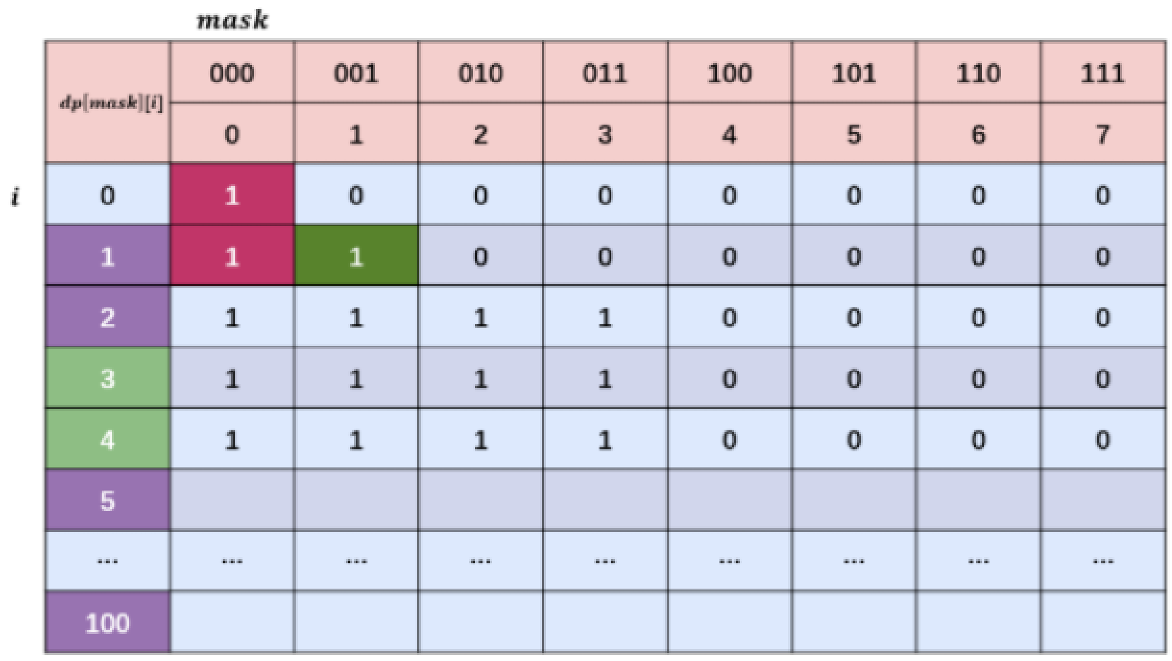
Since the answer may be too large, return it modulo $10^9 + 7$.

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 ^ 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 ^ (1 << 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：

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

这里需要对 `mask ^ (1 << j)` ** **进行说明，** `1 << j` ** 就表示左移 `j` 位，含义是第 j 个人戴帽子的的二进制表示，比如 `dp[3][2]` 中， `capList[2]={1}` ， `j = 1` ， ** `1 << j = 010` ** ；而 `mask ^ (1 << 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 ^ 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：



	mask							
dp[mask][i]	000	001	010	011	100	101	110	111
	0	1	2	3	4	5	6	7
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								

```java
public int numberWays(List<List<Integer>> hats) {
    int n = hats.size(), range = 1 << n, mod = (int)1e9 + 7;
    List<Integer> ids = new ArrayList<>();
    Map<Integer, List<Integer>> hm = new HashMap<>(); // hats map
    for (int i = 0; i < n; i++)
        for (int j = 0; j < hats.get(i).size(); j++) {
            hm.computeIfAbsent(hats.get(i).get(j), k -> new ArrayList<>());
            hm.get(hats.get(i).get(j)).add(i);
            if (!ids.contains(hats.get(i).get(j))) ids.add(hats.get(i).get(j));
        }
   int [][] dp = new int [range][ids.size()+1];
   dp[0][0] = 1; // 回到 dp[0][0] 状态时为 1 个方案数!
    for (int i = 1; i <= ids.size(); i++)
        for (int mask = 0; mask < range; mask++) {
            dp[mask][i] = dp[mask][i-1];     // 1. 第 i 个帽子不分配的情况
            int size = hm.get(ids.get(i-1)).size();
            for (int j = 0; j < size; j++)   // 2. 第 i 个帽子分给第 j 个人的情况
                if ((mask & (1 << hm.get(ids.get(i-1)).get(j))) != 0) {
                    dp[mask][i] += dp[mask ^ (1 << hm.get(ids.get(i-1)).get(j))][i-1]; // 第 i 个帽子可以是由右边异或的状态转移过来的
                    dp[mask][i] %= mod;
                }
        }
    return dp[range-1][ids.size()];
}
```

2.14 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.

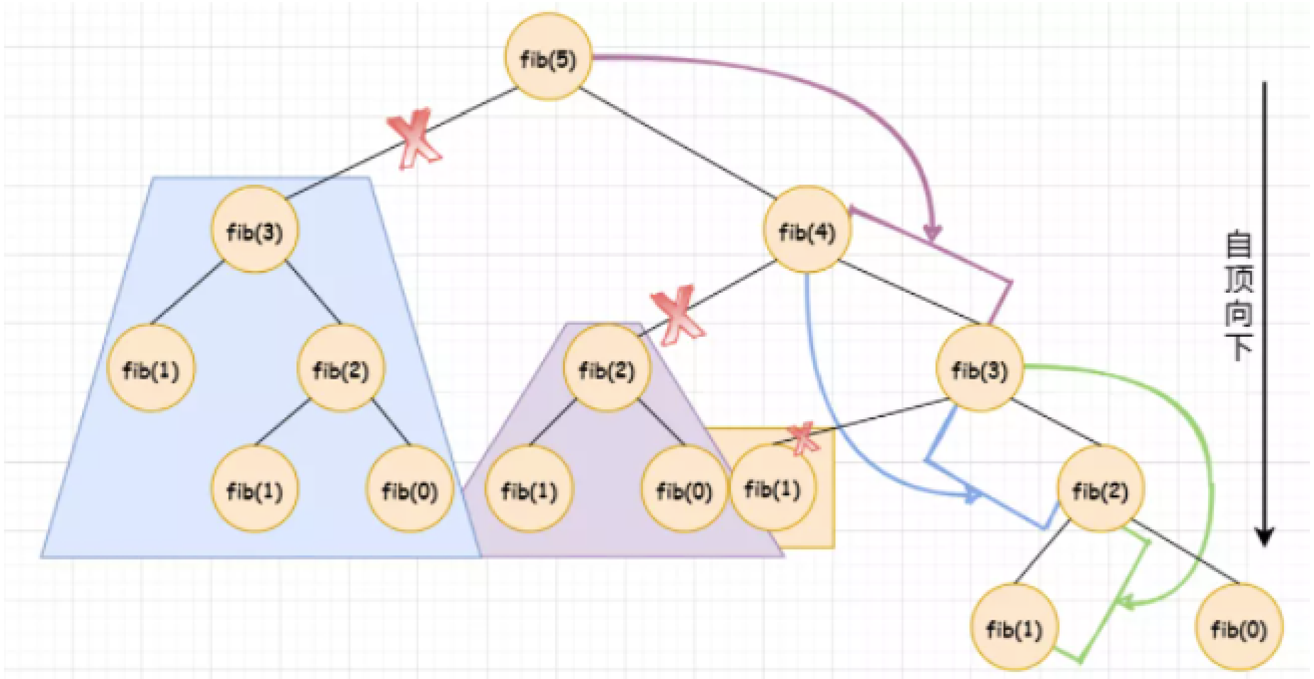
```
// Staightforward top-down DP for the first group. At the same time, we track which elements from the second group were connected in mask.
// After finishing with the first group, we detect elements in group 2 that are still disconnected,
// and connect them with the "cheapest" node in the first 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 << 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];
}
```

2.14.1 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

2.15 1755. Closest Subsequence Sum - Hard 分成两半: 由 O(2^N) 降为 O(NlogN)

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 abs(sum - goal).

Return the minimum possible value of abs(sum - 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-Hor
    s-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

2.16 2035. Partition Array Into Two Arrays to Minimize Sum Difference - Hard 上一题：分成两半的套娃题

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;
}
```

- 做题的时候有想起 tallest billboard，但因为有负数，就去想别的了，却忘记了上面那个一分为二的经典题型