LeetCode Online Coding Interview Questions – 错题集

deepwaterooo

2021 年 9 月 30 日

目录

1 一周计划

- 今天电脑坏了，因为没能更新好 zsh 有个后台程序无限执行，弄了大半天电脑运行环境，现在的配置很好用很喜欢，除了小问题，基本满意
- 计划接下来几周，每天刷 6-10 个题，边刷边记错题进小本本，到这个周末刷到 850 左右；
- union find 的几个题，现在应该可以试着扫完
- 最讨厌扫描线，几个双数怎么也数不清楚。。。。。这个周扫几个出去
- bit manipulation bitmasks, dyanmic programming 的 mask 相关的，循序渐近地试着把它们写完
- Deque 双端队列 O（n）解法的概念在建立，需要练习和熟悉
- 如果说以前是迷迷糊糊刷题求 AC, 现在基本的概念在建立，希望从以前代码和题目的解法效率中等偏下走向代码优化中等偏上，寻求高效、最优解法的提升
- 其它的，等这个周 60 个题左右之后，再总结计划下一周
- linked list, trees 比较简单，基本写完；
- bfs, dfs 在续写了 dynamic programming 之后，终于觉得它们俩个也比较简单，试着扫完；

2 Bit Manipulations

2.1 基本概念：原码、反码、与补码（对负数尤其重要）

2.1.1 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。

2.1.2 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。

2.1.3 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。

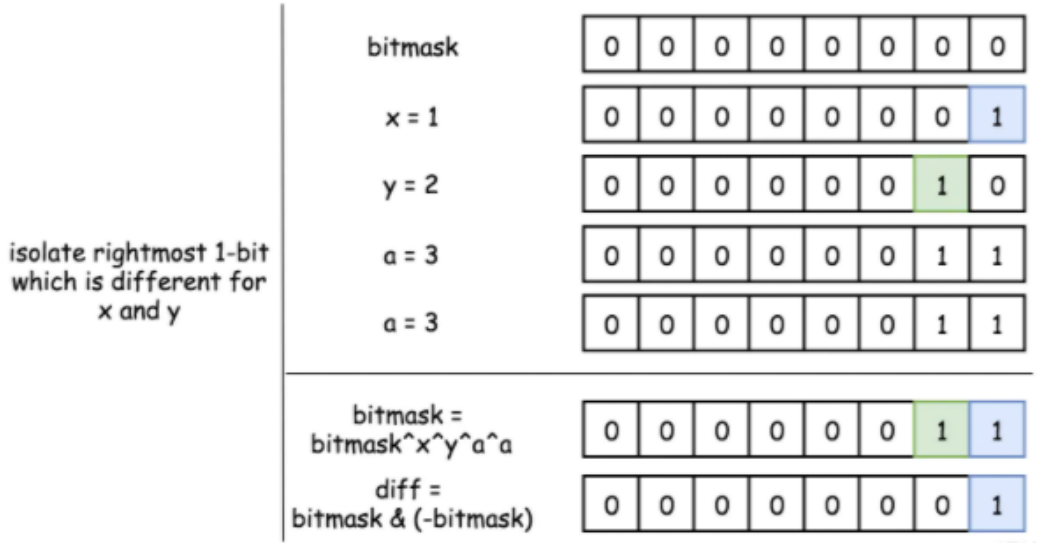
2.2 数组中不重复的两个元素

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)。



2.3 371. 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
```

```
10000000000000000000000000000000
11000000000000000000000000001010
10000000000000000000000000000000
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。

```java
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;
}
```

2.4 201. 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.

```java
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;
}
```

2.5 1835. Find XOR Sum of All Pairs Bitwise AND

Hard

234

26

Add to List

Share 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;
}
```

2.6 982. 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.

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.

```
//   '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] DEFINITION NOT FOUND.

2.7 187. 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：

```
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;
}
```

2.8 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。

```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;
}
```

2.9 957. 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：

```java
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)) { // 应该是找到某个重复的规律，今天不想再看这题了
        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.

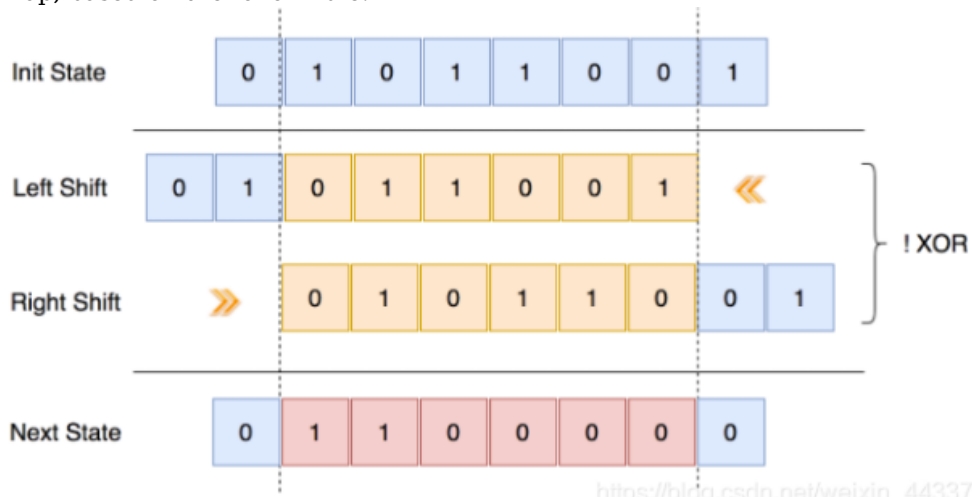
This is the method called fast-forward.

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:




```
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;
}
```

3 Bit Masks

3.1 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)

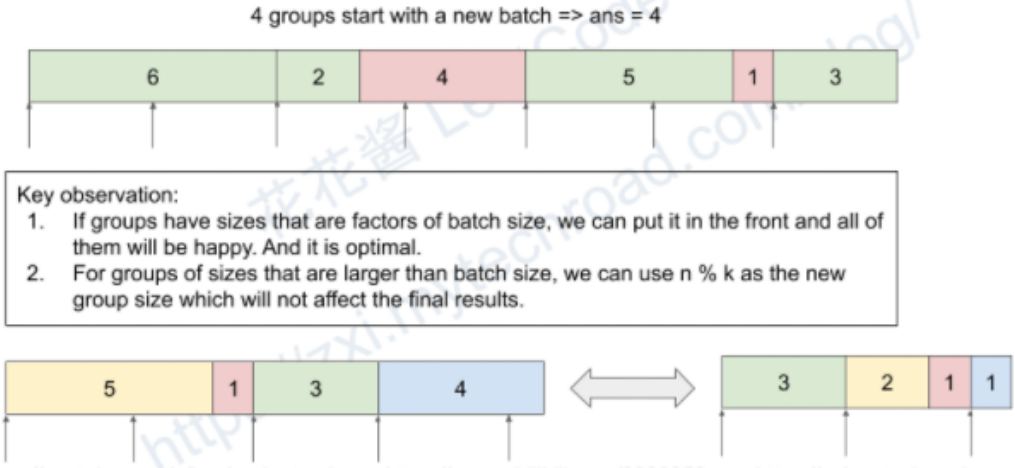
3.2 1815. Maximum Number of Groups Getting Fresh Donuts - Hard

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]




```
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 ?
        ++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 modificatio
    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

```
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 ?
        ++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 modificatio
    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

```
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

```
// 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];
}
```

3.3 691. 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
                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 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

```
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];
}
```

3.4 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;
}
```

3.5 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;
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

```
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 ses
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];
}
```

3.6 956. 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.

```
// // 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

```
// 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
```

```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
    //          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
    //          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
    //          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
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);
}
```

3.7 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.

```java
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;
}
```

3.8 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 [2] 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] DEFINITION NOT FOUND.

3.8.1 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-2B-DP-Solution-(15ms)

```java
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 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-Linear-Time-or-9-ms

```
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-
- 暂略

3.9 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[][] 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
                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;
}
```

3.10 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：

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

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：

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

```
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
                    dp[mask][i] %= mod;
                }
        }
    return dp[range-1][ids.size()];
}
```

3.11 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 con
// 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);
}
```

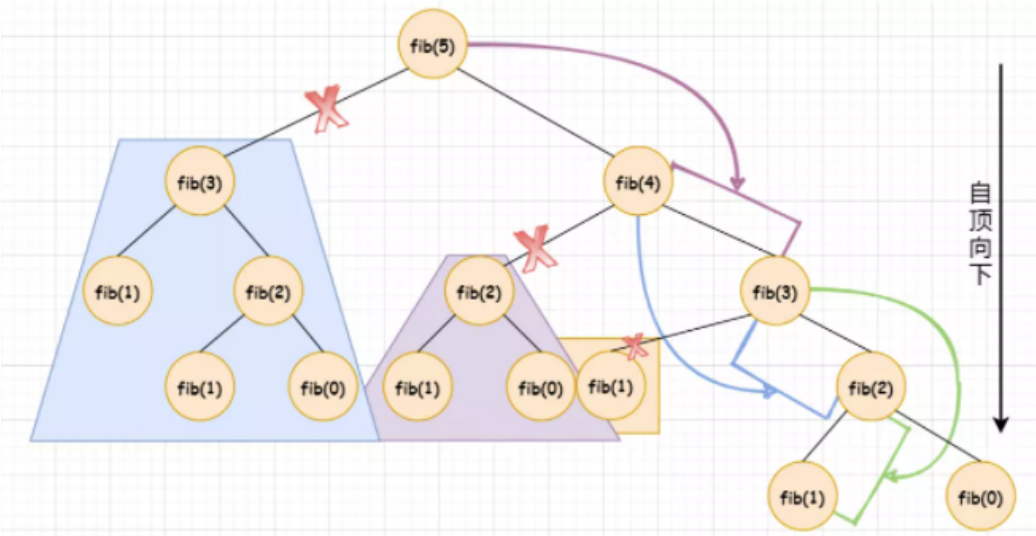
- 动态规划，用二进制压缩状态，注意分析几种情况，就能推出来正确的状态转移方程。

```
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];
}
```

3.11.1 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

4 Dynamic Programming, 动态规划

- 根据 CLRS，动态规划分为两种：
- top-down with memoization (递归记忆化搜索)

等价于带缓存的，搜索树上的 DFS 比较贴近于新问题正常的思考习惯

- bottom-up (自底向上循环迭代)

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

4.1 1125. Smallest Sufficient Team - Hard 这个题要再写几遍：不是我的编码风格

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

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

For example, team = [0, 1, 3] represents the people with skills people[3], people[4], and people[1]. Return any sufficient team of the smallest possible size, represented by the index of each person. You may return the answer in any order.

It is guaranteed an answer exists.

```
private int teamSize;
private Set<Integer> resTeam;
public int[] smallestSufficientTeam(String[] req_skills, List<List<String>> people) {
    int n = people.size();
    teamSize = n;
    resTeam = new HashSet<Integer>();
    HashMap<String, Set<Integer>> map = new HashMap<>(); // map input to <Skill, Set<People>>;
    for(int i = 0; i < n; i++)
        for(String skill: people.get(i)) {
            map.putIfAbsent(skill, new HashSet<Integer>());
            map.get(skill).add(i);
        }
    HashSet<Integer> team = new HashSet<>(); // search res;
    dfs(map, req_skills, team, 0);
    int[] res = new int[resTeam.size()]; // collect res;
    int index = 0;
    for (int person: resTeam)
        res[index++] = person;
    return res;
}
private void dfs(HashMap<String, Set<Integer>> map, String[] req_skills, HashSet<Integer> team, int index) {
    if(team.size() > teamSize)  return;
    if(index == req_skills.length) {
        teamSize = team.size();
        resTeam = new HashSet<Integer>(team);
        return;
    }
    for(int people: map.get(req_skills[index])) {
        boolean isNewPerson = !team.contains(people);
        team.add(people);
        dfs(map, req_skills, team, index + 1);
        if (isNewPerson)
            team.remove(people);
    }
}
```

4.2 1575. Count All Possible Routes - Hard

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

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

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

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

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

```
// 自顶向下 （记忆化搜索）
// 每个 dfs 搜索当前状态为城市 i，油量 f 到达终点的方案数。这样决策的时候就很直观：当前这个状态的方案数，由可去的城市的，且油量为剩余
// 初始化：每个状态都初始化为-1。
```

[3]DEFINITION NOT FOUND.

[4]DEFINITION NOT FOUND.

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

4.3 514. Freedom Trail - Hard

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

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

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

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

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

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

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

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

4.3.2 解题思路分析动态规划

- 博主最先尝试的用贪婪算法来做，就是每一步都选最短的转法，但是 OJ 中总有些 test case 会引诱贪婪算法得出错误的结果，因为全局最优解不一定都是局部最优解，而贪婪算法一直都是在累加局部最优解，这也是为啥 DP 解法这么叼的原因。贪婪算法好想好实现，但是不一定能得到正确的结果。DP 解法难想不好写，但往往才是正确的解法，这也算一个 trade off 吧。

- 此题需要使用一个二维数组 dp，其中 dp[i][j] 表示转动从 i 位置开始的 key 串所需要的最少步数 (这里不包括 spell 的步数，因为 spell 可以在最后统一加上)，此时表盘的 12 点位置是 ring 中的第 j 个字符。不得不佩服这样的设计的确很巧妙，我们可以从 key 的末尾往前推，这样 $dp^{3,3}$就是我们所需要的结果，因为此时是从 key 的开头开始转动，而且表盘此时的 12 点位置也是 ring 的第一个字符。现在我们来看如何找出递推公式，对于 dp[i][j]，我们知道此时要将 key[i] 转动到 12 点的位置，而此时表盘的 12 点位置是 ring[j]，我们有两种旋转的方式，顺时针和逆时针，我们的目标肯定是要求最小的转动步数，而顺时针和逆时针的转动次数之和刚好为 ring 的长度 n，这样我们求出来一个方向的次数，就可以迅速得到反方向的转动次数。为了将此时表盘上 12 点位置上的 ring[j] 转动到 key[i]，我们要将表盘转动一整圈，当转到 key[i] 的位置时，我们计算出转动步数 diff，然后计算出反向转动步数，并取二者较小值为整个转动步数 step，此时我们更新 dp[i][j]，更新对比值为 step + dp[i+1][k]，这个也不难理解，因为 key 的前一个字符 key[i+1] 的转动情况 suppose 已经计算好了，那么 dp[i+1][k] 就是当时表盘 12 点位置上 ring[k] 的情况的最短步数，step 就是从 ring[k] 转到 ring[j] 的步数，也就是 key[i] 转到 ring[j] 的步数，用语言来描述就是，从 key 的 i 位置开始转动并且此时表盘 12 点位置为 ring[j] 的最小步数 (dp[i][j]) 就等价于将 ring[k] 转动到 12 点位置的步数 (step) 加上从 key 的 i+1 位置开始转动并且 ring[k] 已经在表盘 12 点位置上的最小步数 (dp[i+1][k]) 之和。
- 突然发现这不就是之前那道 Reverse Pairs 中解法一中归纳的顺序重现关系的思路吗，都做了总结，可换个马甲就又不认识了，泪目中。。。

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

4.3.3 解题思路分析: dfs + 记忆数组

- 过程就是需要一步一步求 key 里面的每个字符。如果当前位置已经是对应到这个字符，那么直接按按钮就可以
- 如果当前位置不是，那么有两种旋转方式，顺时针或者逆时针, 然后找到第一个字符就是在同一个方向上的最短距离，
- 因为在同一个方向上，即使后面有重复的字符，无论后面的字符在那里，遇到第一个符合条件的字符就按按钮一定是最优解。
- 但是在不同方向上就不一定了，有可能一个方向上当前字符距离更短，但是有可能后面的字符距离会更远，
  - 比如 ring=ABCDEFGBF , key=BG, 如果看第一个字符，那应该是顺时针，只需要转一格就到，逆时针需要转两格，
  - 但是顺时针第一步快了以后，后面到 G 会需要更长的步骤。而逆时针会比较快。

- 所以，基本的逻辑是每一步不能决定当前哪个方向是否是最优解，只有不断递归，把每步的两个方向全部尝试完到 key 结束才可以
- 当然，如果不做任何处理，这样做是要超时的（我开始就写了这样一个版本），一个直观的做法，就是在递归的基础上
  - 加一个记忆表，针对 ring 的位置 index 和 key 的 kindex 做记录，如果已经存在一个解了就可以直接返回结果
- 这个递归 +memorization 的解法，那一定存在一个 bottom up 的动态规划解法，这个后面再学习

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

4.4 847. Shortest Path Visiting All Nodes

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

```
public int shortestPathLength(int[][] graph) {
    int n = graph.length;
    int tar = 0, res = 0;
    HashSet<String> s = new HashSet<>();
    Queue<Pair<Integer, Integer>> q = new LinkedList<>();
```

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

4.5 1931. Painting a Grid With Three Different Colors

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

- lightweighted 轻巧点儿的解题方案: bitmask

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

- 比较传统一点儿的解法，思路清晰

```java
// 参考的答案里，这个最逻辑简单、通俗大众易懂，但稍显笨重，两个图，用一个链表来记忆一行的涂色方案，如果有更精巧一点儿的 bitmask，是我
// https://leetcode.com/problems/painting-a-grid-with-three-different-colors/discuss/1334366/Easy-Java-comments-28ms-O(n*P*
// 先预处理得到单行的所有有效涂色方案，
// 再进一步计算得到每种单行方案对应的有效邻行方案
// 在此基础上，结合动态规划方法，逐行求解各种涂色状态对应的方案总数，最后统计得到总方案数。
public int colorTheGrid(int m, int n) {
// 获得单行所有涂色方案
    Map<Integer, List<Integer>> line = new HashMap<>(); //  3^m ways of paying one row
    int range = (int)Math.pow(3, m); // 用 0、1、2 表示各个网格的颜色，key 为方案对应的数值，value 为方案对应的数组
    for (int i = 0; i < range; i++) {
        List<Integer> list = new ArrayList<>(); //  val val values (0, 1, 2) of every m cols into list
        int val = i;
        for (int j = 0; j < m; j++) {
            list.add(val % 3);
            val /= 3;
        }
        boolean valid = true; // 确认该数组中是否存在相邻位置颜色相同
        for (int j = 1; j < m; j++)
            if (list.get(j-1) == list.get(j)) {
                valid = false;
                break;
            }
        if (valid) line.put(i, list); // 相邻网格颜色均不同，为有效方案，加入哈希表
    }
// 预处理得到每种单行方案对应的有效邻行方案
    Map<Integer, List<Integer>> adj = new HashMap<>();
    Iterator it = line.entrySet().iterator();
    while (it.hasNext()) {     //  3^m ways of paying one row
        Map.Entry entry = (Map.Entry)it.next();
        int va = (int)entry.getKey();
        List<Integer> lva = (List<Integer>)entry.getValue();
        adj.put(va, new ArrayList<Integer>());
        Iterator itb = line.entrySet().iterator();
        while (itb.hasNext()) { //  3^m ways of paying one row
            Map.Entry enb = (Map.Entry)itb.next();
            int vb = (int)enb.getKey();
            List<Integer> lvb = (List<Integer>)enb.getValue();
            boolean valid = true;
            for (int i = 0; i < m; i++)
                if (lva.get(i) == lvb.get(i)) {
                    valid = false;
                    break;
                } // among 3^m ways of painting one row, how many is valid, and valid mask into adj.get(va);
            if (valid) adj.get(va).add(vb);
        }
    }
// 动态规划，逐行求解方案数
    int mod = (int)(1e9+7);
    long [] dp = new long [range];  // 上一行各种涂色方案对应的总方法数
    for (int i = 0; i < range; i++) // 初始化
        dp[i] = line.containsKey(i) ? 1 : 0;
    for (int i = 1; i < n; i++) {   // 从第二行开始动态规划
        long [] cur = new long [range];  // 新一行各种涂色方案对应的总方法数
        for (int j = 0; j < range; j++)
            if (adj.containsKey(j)) {    // 该方案有效
                for (int v : adj.get(j)) // 遍历有效的相邻方案
                    cur[j] = (cur[j] + dp[v]) % mod; // 总方法数累加
            }
        System.arraycopy(cur, 0, dp, 0, range);
    }
    long ans = 0;
    for (int i = 0; i < range; i++)
        ans = (ans + dp[i]) % mod;
    return (int)ans;
}
```

4.6 313. Super Ugly Number

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

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

- 下面这种解法也很巧妙

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

4.7 1786. Number of Restricted Paths From First to Last Node

There is an undirected weighted connected graph. You are given a positive integer n which denotes that the graph has n nodes labeled from 1 to n, and an array edges where each edges[i] = [ui, vi, weighti] denotes that there is an edge between nodes ui and vi with weight equal to weighti. A path from node start to node end is a sequence of nodes [z0, z1, z2, . . . , zk] such that

z0 = start and zk = end and there is an edge between zi and zi+1 where 0 <= i <= k-1. The distance of a path is the sum of the weights on the edges of the path. Let distanceToLastNode(x) denote the shortest distance of a path between node n and node x. A restricted path is a path that also satisfies that distanceToLastNode(zi) > distanceToLastNode(zi+1) where 0 <= i <= k-1. Return the number of restricted paths from node 1 to node n. Since that number may be too large, return it modulo 109 + 7.

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

4.8 913. Cat and Mouse

A game on an undirected graph is played by two players, Mouse and Cat, who alternate turns. The graph is given as follows: graph[a] is a list of all nodes b such that ab is an edge of the graph. The mouse starts at node 1 and goes first, the cat starts at node 2 and goes second, and there is a hole at node 0. During each player's turn, they must travel along one edge of the graph that meets where they are. For example, if the Mouse is at node 1, it must travel to any

node in graph[4]. Additionally, it is not allowed for the Cat to travel to the Hole (node 0.) Then, the game can end in three ways: If ever the Cat occupies the same node as the Mouse, the Cat wins. If ever the Mouse reaches the Hole, the Mouse wins. If ever a position is repeated (i.e., the players are in the same position as a previous turn, and it is the same player's turn to move), the game is a draw. Given a graph, and assuming both players play optimally, return 1 if the mouse wins the game, 2 if the cat wins the game, or 0 if the game is a draw.

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

4.9 1728. Cat and Mouse II

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

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

4.10 322. Coin Change

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

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

```
    }
    return dp[amount] == amount + 1 ? -1 : dp[amount];
}
```

4.11 518. Coin Change 2

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

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

4.12 backpack III

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

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

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

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

4.14 1049. Last Stone Weight II

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

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

4.15 1449. Form Largest Integer With Digits That Add up to Target

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

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

4.16 516. Longest Palindromic Subsequence

Given a string s, find the longest palindromic subsequence's length in s. A subsequence is a sequence that can be derived from another sequence by deleting some or no elements

without changing the order of the remaining elements.

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

4.17 1143. Longest Common Subsequence

Given two strings text1 and text2, return the length of their longest common subsequence. If there is no common subsequence, return 0. A subsequence of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters. For example, "ace" is a subsequence of "abcde". A common subsequence of two strings is a subsequence that is common to both strings.

4.18 312. Burst Balloons 区间型动态规划的典型代表

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

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

```
}
int [][] dp;
int n;
public int maxCoins(int[] nums) {
    int n = nums.length + 2;
    dp = new int [n][n];
    int [] arr = new int [n];
    System.arraycopy(nums, 0, arr, 1, n-2);
    arr[0] = arr[n-1] = 1;
    return memorizedSearch(arr, 1, n-2);
}
```

4.19 1000. Minimum Cost to Merge Stones

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

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

4.20 375. Guess Number Higher or Lower II - Medium

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

I pick a number between 1 and n.

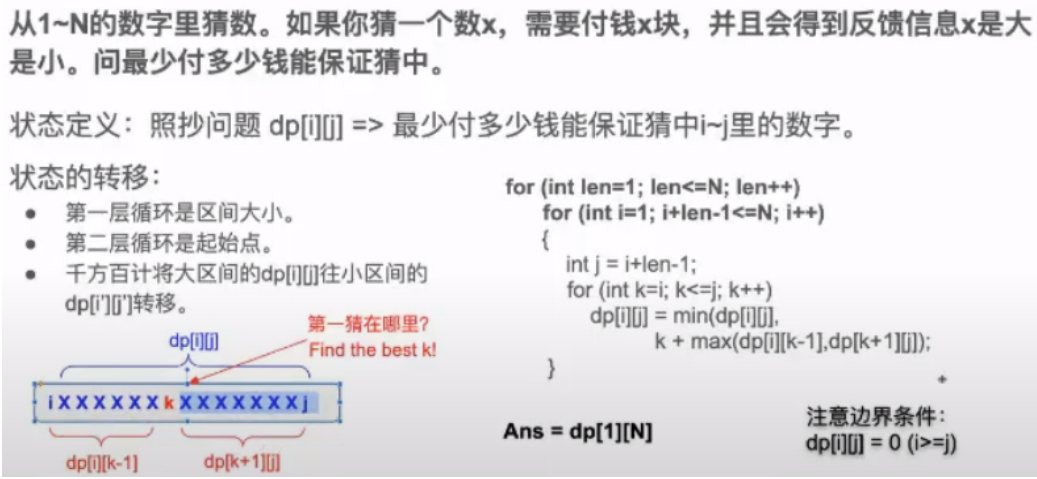
You guess a number.

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

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

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

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




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

4.21 486. Predict the Winner

You are given an integer array nums. Two players are playing a game with this array: player 1 and player 2. Player 1 and player 2 take turns, with player 1 starting first. Both players start the game with a score of 0. At each turn, the player takes one of the numbers from either end of the array (i.e., nums[3] or nums[nums.length - 1]) which reduces the size of the array by 1. The player adds the chosen number to their score. The game ends when there are no more elements in the array. Return true if Player 1 can win the game. If the scores of both players are equal, then player 1 is still the winner, and you should also return true. You may assume that both players are playing optimally.

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

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

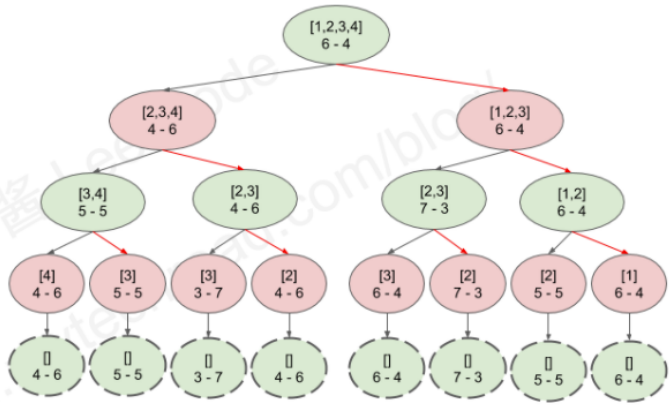
Solution: Recursion + Memorization

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

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

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

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




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

4.22 877. Stone Game

Alice and Bob play a game with piles of stones. There are an even number of piles arranged in a row, and each pile has a positive integer number of stones piles[i]. The objective of the game is to end with the most stones. The total number of stones across all the piles is odd, so there are no ties. Alice and Bob take turns, with Alice starting first. Each turn, a player takes the entire pile of stones either from the beginning or from the end of the row. This continues until there are no more piles left, at which point the person with the most stones wins. Assuming Alice and Bob play optimally, return true if Alice wins the game, or false if Bob wins.

```java
// 使用 helper 函数表示 Alex 能比 Lee 多选的分数。可能比双函数更简洁易懂了。
// 记忆化递归的缺点：1. 有可能爆栈；2. 无法降维，而 DP 是可以降维的。
// 模板：
// dfs + memoization 模板
private int dfs(int [] arr, int l, int r) {
    if (l > r) return 0;
    if (dp[l][r] > 0) return dp[l][r]; // 走了来时的路，不需要重走，直接返回
    dp[l][r] = Math.max(arr[l] - dfs(arr, l+1, r), arr[r]-dfs(arr, l, r-1));
    return dp[l][r];
}
int [][] dp;
public boolean stoneGame(int[] piles) {
    int n = piles.length;
    dp = new int[n][n];
    return dfs(piles, 0, n-1) > 0;
}
// 动态规划解法比较难想，dp 数组的第 i 个位置表示的是从第 i 个石头到第 i+l-1 个石头之间最大的比对手得分。
// 使用的是一个长度变量和起始索引，计算每个位置开始的长度 1～N 长度的区间的 dp 状态。
public boolean stoneGame(int[] piles) {
    int n = piles.length;
    int [][] dp = new int[n][n];
    for (int i = n-1; i >= 0; i--) { // 最后一列
        dp[i][i] = piles[i];         // 填右上角
        for (int j = i+1; j < n; j++)
            dp[i][j] = Math.max(piles[i] - dp[i+1][j], piles[j]-dp[i][j-1]);

    }
```

```
    return dp[0][n-1] > 0;
}
```

4.23 1140. Stone Game II

Alice and Bob continue their games with piles of stones. There are a number of piles arranged in a row, and each pile has a positive integer number of stones piles[i]. The objective of the game is to end with the most stones. Alice and Bob take turns, with Alice starting first. Initially, M = 1. On each player's turn, that player can take all the stones in the first X remaining piles, where 1 <= X <= 2M. Then, we set M = max(M, X). The game continues until all the stones have been taken. Assuming Alice and Bob play optimally, return the maximum number of stones Alice can get.

```
// dfs + memoization 模板
// 当次取的最优策略是限制下一次取的数量
private int dfs(int [] arr, int idx, int m) {
    if (dp[idx][m] > 0) return dp[idx][m];
    if (idx == n) return 0;
    if (idx >= n - 2 * m) {
        dp[idx][m] = suf[idx];
        return dp[idx][m];
    }
    int min = Integer.MAX_VALUE;
    for (int i = 1; i <= 2 * m; i++) // 选择限制对手得分最少的情况
        min = Math.min(min, dfs(arr, idx+i, Math.max(m, i)));
    dp[idx][m] = suf[idx] - min;
    return dp[idx][m];
}
int [][] dp;
int [] suf;
int n;
public int stoneGameII(int[] piles) {
    n = piles.length;
    dp = new int [n][2*n];
    suf = new int [n+1];
    for (int i = n-1; i >= 0; i--)
        suf[i] = suf[i+1] + piles[i];
    return dfs(piles, 0, 1);
}
```

4.24 1872. Stone Game VIII

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

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

```
    Arrays.fill(dp, Integer.MIN_VALUE);
    int [] pre = new int [n+1];
    for (int i = 1; i <= n; i++)
        pre[i] = pre[i-1] + stones[i-1];
    dp[n-1] = pre[n];
    for (int i = n-2; i >= 0; i--)
        dp[i] = Math.max(dp[i+1], pre[i+1]-dp[i+1]);
    return dp[1];
}
```

4.25 213. House Robber II

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed. All houses at this place are arranged in a circle. That means the first house is the neighbor of the last one. Meanwhile, adjacent houses have a security system connected, and it will automatically contact the police if two adjacent houses were broken into on the same night. Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

```
private int dfs(int [] arr, int i, int end, int [] dp) { // end: n-1
    if (i > end) return 0;
    if (i == end) {
        dp[i] = arr[i];
        return dp[i];
    }
    if (dp[i] > 0) return dp[i];
    dp[i] = Math.max(arr[i] + dfs(arr, i+2, end, dp), dfs(arr, i+1, end, dp));
    return dp[i];
}
int n;
public int rob(int[] nums) {
    n = nums.length;
    if (n == 1) return nums[0];
    if (Arrays.stream(nums).distinct().count() == 1 && nums[0] == 0) return 0;
    int [] dp = new int [n];
    int max = dfs(nums, 0, n-2, dp);
    Arrays.fill(dp, 0);
    return Math.max(max, dfs(nums, 1, n-1, dp));
}
```

4.26 740. Delete and Earn: 这也是上面一类题目，变形了居然没有认出来再写一下这个

You are given an integer array nums. You want to maximize the number of points you get by performing the following operation any number of times: Pick any nums[i] and delete it to earn nums[i] points. Afterwards, you must delete every element equal to nums[i] - 1 and every element equal to nums[i] + 1. Return the maximum number of points you can earn by applying the above operation some number of times.

- 该题可以简化到 Leetcode 第 198 题 House Robber。关键题意理解：对于 nums = [2, 2, 3, 3, 3, 4]，如果我们选了数字 3，则要删除每一个（所有）2 和 4，之后数组变为 [3, 3]，此时如果再选 3，并不会删除其他元素，剩余数组变为[1]，再选 3，数组变为空，返回结果是 3+3+3=9。
- 那么既然如此，我们可以将上面的例子转化为 House Robber 的形式，nums' = [0, 0, 22, 33, 4]，nums'[i] 代表值为 i 的数字的总数和，每次若选择 nums' 数组的一个元素，则相邻元素不能在下一次被选中，相当于被删去。转化后，用 House Robber 的思路来处理 nums' 数组即可。

```java
private int dfs(int [] arr, int v) {
    if (v <= 0) return 0;
    if (dp[v] > 0) return dp[v];
    dp[v] = Math.max(dfs(arr, v-1), v * cnt[v] + dfs(arr, v-2));
    return dp[v];
}
int [] dp;
int [] cnt;
public int deleteAndEarn(int[] nums) {
    Arrays.sort(nums);
    dp = new int [nums[nums.length-1]+1];
    cnt = new int[nums[nums.length-1]+1];
    for (int v : nums)
        cnt[v]++;
    return dfs(nums, nums[nums.length-1]);
}
```

4.27 123. Best Time to Buy and Sell Stock III

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

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

4.28 188. Best Time to Buy and Sell Stock IV

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

```java
public int maxProfit(int k, int [] prices) {
    if (prices == null || prices.length == 0) return 0;
    int n = prices.length;
    int diff = 0;
```

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

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

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

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

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

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

- 感觉自己 DP 的能力还是太弱，越是这样越需要迎难而上。
- 这个题和 714. Best Time to Buy and Sell Stock with Transaction Fee 比较像。做题方法都是使用了两个数组：
- cash 该天结束手里没有股票的情况下，已经获得的最大收益
- hold 该天结束手里有股票的情况下，已经获得的最大收益
- 状态转移方程式这样的：

  - cash[i] 代表的是手里没有股票的收益，这种可能性是今天卖了或者啥也没干。max(昨天手里有股票的收益 + 今天卖股票的收益，昨天手里没有股票的收益)，即 max(sell[i - 1], hold[i - 1] + prices[i])；
  - hold[i] 代表的是手里有股票的收益，这种可能性是今天买了股票或者啥也没干，今天买股票必须昨天休息。所以为 max(今天买股票是前天卖掉股票的收益-今天股票的价格，昨天手里有股票的收益)。即 max(hold[i - 1], sell[i - 2] - prices[i])。

- 另外需要注意的是，题目说的是昨天卖了股票的话今天不能买，对于开始的第一天，不可能有卖股票的行为，所以需要做个判断。
- 该算法的时间复杂度是 O(n)，空间复杂度是 O(n)。

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

4.31 1039. Minimum Score Triangulation of Polygon

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

4.32 673. Number of Longest Increasing Subsequence

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

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

4.33 823. Binary Trees With Factors

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

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

4.34 907. Sum of Subarray Minimums

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

```
public int sumSubarrayMins(int[] arr) {
    int n = arr.length;
    // for each A[i], find k <= i <= j, so that A[i] is the min from [k,j]
    // sum += A[i] * (i-k+1) * (j-i+1)
    // so we need to find the next min to the right and to the left
    //这个过程可以简化为使用一个栈。对于被某个数从栈中弹出的数而言，它右侧第一个比它小的数就是这个数。所以我们可以对所有被弹出的数得到
    long [] right = new long [n];  // next smaller element index to the right
    long [] left = new long[n];    // next smaller element index to the left
    Stack<Integer> s = new Stack<>();
    for (int i = 0; i < n; i++) {
        while (!s.isEmpty() && arr[i] <= arr[s.peek()]) {
            right[s.pop()] = i-1;
        }
        s.push(i);
    }
    while (!s.isEmpty()) {
        right[s.pop()] = n-1;
    }
    s.clear();
    for (int i = n-1; i >= 0; i--) {
        while (!s.isEmpty() && arr[i] < arr[s.peek()])
            left[s.pop()] = i+1;
        s.push(i);
    }
    while (!s.isEmpty())
        left[s.pop()] = 0;
    long sum = 0;
    long leftsize = 0, rightsize = 0;
    for (int i = 0; i < n; i++) {
        leftsize = i - left[i] +1;
        rightsize = right[i] - i + 1;
        sum += arr[i] * leftsize * rightsize;
        sum %= mod;
```

```
    }
    return (int)sum;
}
int mod = 1_000_000_007;
public int sumSubarrayMins(int[] arr) {
    int n = arr.length;
    long [] left = new long[n];
    long [] right = new long[n];
    long sum = 0;
    long cnt = 0;
    int j = 0;
    for (int i = 0; i < n; i++) { // 计算左边比自身大的数的个数
        cnt = 1;
        j = i-1;
        while (j >= 0 && arr[j] >= arr[i]) {
            cnt += left[j];
            j -= left[j];
        }
        left[i] = cnt;
    }
    // 就是因为计算了两个方向，所以对于数组里面有相同元素的情况下，需要特别考虑一下。
    //      不能重复计算，也不能漏掉，
    //      具体就是一个方向的时候用 <=，另外一个方向的时候用 <。这个在做的时候也 bug 了。
    for (int i = n-1; i >= 0; i--) { // 计算右边比自身大的数的个数
        cnt = 1;
        j = i+1;
        while (j < n && arr[j] > arr[i]) {
            cnt += right[j];
            j += right[j];
        }
        right [i] = cnt;
    }
    for (int i = 0; i < n; i++)
        sum += arr[i] * left[i] * right[i];
    return (int) (sum % mod);
}
```

4.35 494. Target Sum - Medium

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

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

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

- 该题是一道非常经典的题目，在面试中很可能会考到。该题有多种解法。

4.35.1 第一种解法：DFS，brute force。我们对 nums 数组中的每个数字，都尝试在其前面添加正号和负号，最后暴力求解，统计数组中各数字组合值为 target 的情况。(该理解是错误的，我们可以使用带备忘录机制的自顶向下的 DP 方法，代码见下)

```
//带备忘录机制的自顶向下 DP 解法 //map 存储重复的值，there are obvious a lot of overlap subproblems
private int helper(int[] nums, int index, int sum, int S, Map<String, Integer> map){
    String encodeString = index + "->" + sum; //经过之前不同的运算过程到达 index 的 sum 值碰巧与之前某一个运算过程的结果相同
    if (map.containsKey(encodeString)) return map.get(encodeString);
    if (index == nums.length)
        if (sum == S) return 1;
        else return 0;
    int curNum = nums[index];
    int add = helper(nums, index + 1, sum - curNum, S, map);
    int minus = helper(nums, index + 1, sum + curNum, S, map);
    map.put(encodeString, add + minus);
    return add + minus;
}
public int findTargetSumWays(int[] nums, int S) {
    if (nums == null || nums.length == 0) return 0;
```

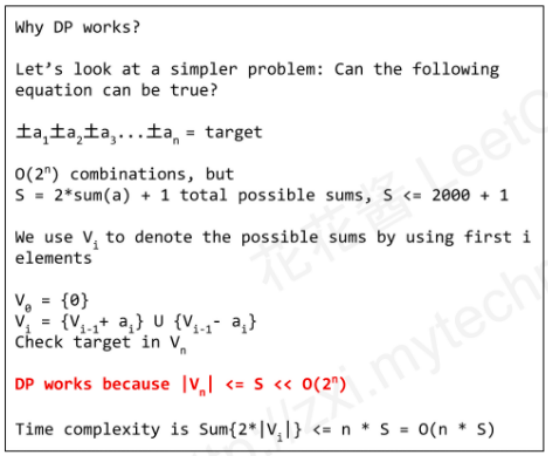
```
    return helper(nums, 0, 0, S, new HashMap<>());
}
```

4.35.2 第二种解法：**DP**。我们使用 **Vi** 来表示数组中的前 **i** 个数所能求得的和的集合。初始化时

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

Vn 就是 nums 数组所有数字的组合值之和的集合

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



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

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

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

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

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

4.36 1477. Find Two Non-overlapping Sub-arrays Each With Target Sum

Given an array of integers arr and an integer target. You have to find two non-overlapping sub-arrays of arr each with a sum equal target. There can be multiple answers so you have to find an answer where the sum of the lengths of the two sub-arrays is minimum. Return the

minimum sum of the lengths of the two required sub-arrays, or return -1 if you cannot find such two sub-arrays.

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

4.37 1771. Maximize Palindrome Length From Subsequences

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

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

4.38 907. Sum of Subarray Minimums

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

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

```
            return (int) (sum % mod);
        }
```

4.39 689. Maximum Sum of 3 Non-Overlapping Subarrays

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

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

4.40 363. Max Sum of Rectangle No Larger Than K

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

```
public int maxSumSubmatrix(int[][] mat, int k) {
    int m = mat.length;
    int n = mat[0].length;
    if (m == 1 && n == 1) return mat[0][0];
    int [][] pre = new int [m][n];
    int res = Integer.MIN_VALUE;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            int t = mat[i][j];
            if (i > 0) t += pre[i-1][j];
            if (j > 0) t += pre[i][j-1];
            if (i > 0 && j > 0) t -= pre[i-1][j-1];
            pre[i][j] = t;
            for (int r = 0; r <= i; r++) {
```

```
                for (int c = 0; c <= j; c++) {
                    int d = pre[i][j];
                    if (r > 0) d -= pre[r-1][j];
                    if (c > 0) d -= pre[i][c-1];
                    if (r > 0 && c > 0) d += pre[r-1][c-1];
                    if (d <= k) res = Math.max(res, d);
                }
            }
        }
    }
    return res;
}
// 把二维数组按行或列拆成多个一维数组，然后利用一维数组的累加和来找符合要求的数字，
// 这里用了 lower_bound 来加快的搜索速度，也可以使用二分搜索法来替代。
public int maxSumSubmatrix(int[][] mat, int target) {
    int row = mat.length;
    int col = mat[0].length;
    int res = Integer.MIN_VALUE;
    boolean key = col > row ? false : true;
    int m = Math.min(row, col);
    int n = Math.max(row, col);
    int [] pre = new int [n];
    TreeSet<Integer> ts = new TreeSet<>(); //用来保存当前高度下，长度为从 0 开始到 k 位置的矩形的结果。理解 set 的含义是解决此题的
    Integer tmp = 0;
    for (int i = 0; i < m; i++) { // 找从第 i 行开始一直到第 0 行这 i+1 行的可能组成的矩形长度
        Arrays.fill(pre, 0);
        for (int j = i; j >= 0; j--) {
            ts.clear();
            ts.add(0);
            int curSum = 0;
            for (int k = 0; k < n; k++) {
                if (key)
                    pre[k] += mat[k][j];
                else pre[k] += mat[j][k];
                curSum += pre[k];
                 // * 因为要满足  （sum-set 中的元素）<=target,
                 // * 而且 sum-set 中的元素的值要尽可能的大，
                 // * 所以也就是再求小于等于 sum-target 中满足条件的元素的最小的一个
                 // * 正好 TreeSet 中提供了这个方法 ceil()，可以很方便的找出这个元素
                tmp = ts.ceiling(curSum - target);
                if (tmp != null) res = Math.max(res, curSum - tmp);
                ts.add(curSum);
            }
        }
    }
    return res;
}
```

4.41 805. Split Array With Same Average

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

```
public boolean splitArraySameAverage(int[] nums) {
    int n = nums.length;
    int m = n / 2;
    int sum = Arrays.stream(nums).sum();
    boolean poss = false;
    for (int i = 1; i <= m; i++) {
        if (sum * i % n == 0) {
            poss = true;
            break;
        }
    }
    if (!poss) return false;
    List<Set<Integer>> ls = new ArrayList<>();
    for (int i = 0; i <= m; i++)
        ls.add(new HashSet<Integer>());
```

```
    ls.get(0).add(0);    // 这种构建子序列和的方法，要学习一下
    for (int v : nums) { // for each element in A, we try to add it to sums[i] by joining sums[i - 1]
        for (int i = m; i >= 1; i--) {
            for (int t : ls.get(i-1)) {
                ls.get(i).add(t + v);
            }
        }
    }
    // System.out.println("ls.size(): " + ls.size());
    // for (int z = 0; z < ls.size(); ++z) {
    //     for (Integer x : ls.get(z))
    //         System.out.print(x + ", ");
    //     System.out.print("\n");
    //     System.out.print("\n ");
    // }
    for (int i = 1; i <= m; i++) {
        if (sum * i % n == 0 && ls.get(i).contains(sum * i / n))
            return true;
    }
    return false;
}
```

4.42 1981. Minimize the Difference Between Target and Chosen Elements

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

```
// DP + BitSet : 这里面有个小问题需要挑出来
// 使用一个 DP 数组存下当前行和之前行每行选一个数可能构成的和，
// 在本题中，可以使用 BitSet（简介）来存储之前行可以组成的和（由于所有数的最大值为 70，而行数最大也为 70，故 BitSet 最大的位数即为 4
// 对于当前行，遍历 BitSet 已经 set 过的位（即代表之前行可能组成的和），然后加上当前数，set 新的和
// 最后遍历 BitSet，求出当前位与 target 的最小值
public int minimizeTheDifference(int[][] mat, int target) {
    int m = mat.length;
    int n = mat[0].length;
    BitSet sum = new BitSet(); // 遍历每一行，存下当前行和之前行可能组成的和
    for (int i = 0; i < n; i++) // 初始时存下第一行
        sum.set(mat[0][i]);
    for (int i = 1; i < m; i++) {
        BitSet newSum = new BitSet(); // 用来存新的和
        for (int j = 0; j < n; j++) {
            // 注意：要遍历 BitSet 中的真实位，请使用以下循环：previousSetBit() 方法 用于查找在指定的起始索引上或之前是否存在任何真
            // for (int i = bs.length(); (i = bs.previousSetBit(i-1)) >= 0; ) {
            //     // operate on index i here
            // }
            for (int k = sum.length(); (k = sum.previousSetBit(k-1)) >= 0; ) {
                newSum.set(k+mat[i][j]);
            }
        }
        sum = newSum;
    }
    int ans = 4900;
    for (int k = sum.length(); (k = sum.previousSetBit(k-1)) >= 0;) {
        int diff = Math.abs(k - target);
        ans = Math.min(ans, diff);
    }
    return ans;
}
public int minimizeTheDifference(int[][] mat, int target) {
    int m = mat.length;
    int n = mat[0].length;
    int diff = Integer.MAX_VALUE, limit = 4900;
    int [] dp = new int[limit];
    for (int i = 0; i < n; i++) // 相当于是手工实现 java BitSet
        dp[mat[0][i]] = 1;
    for (int i = 1; i < m; i++) {
        int [] tmp = new int [limit];
```

```
        for (int v = limit-1; v >= 0; v--) {
            if (dp[v] == 0) continue;
            for (int j = 0; j < n; j++) {
                if (v + mat[i][j] < limit)
                    tmp[v+mat[i][j]] = 1;
            }
        }
        System.arraycopy(tmp, 0, dp, 0, dp.length);
    }
    for (int i = 0; i < limit; i++)
        if (dp[i] > 0) diff = Math.min(diff, Math.abs(i-target));
    return diff;  // min difference
}
```

4.43 805. Split Array With Same Average

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

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

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

5 sliding window

5.1 862. Shortest Subarray with Sum at Least K - Hard

Given an integer array nums and an integer k, return the length of the shortest non-empty subarray of nums with a sum of at least k. If there is no such subarray, return -1.

A subarray is a contiguous part of an array.

```java
public int shortestSubarray(int[] nums, int k) {
    int n = nums.length;
    int [] sum = new int[n+1];
    for (int i = 1; i <= n; i++)
        sum[i] = nums[i-1] + sum[i-1];
    int res = n + 1;
    ArrayDeque<Integer> q = new ArrayDeque<>(); // decreasing sum [] deque
    for (int i = 0; i <= n; i++) {
        while (!q.isEmpty() && sum[i] - sum[q.peekFirst()] >= k)  // 左出:
            res = Math.min(res, i - q.pollFirst()); // 取值了       // 取解
        while (!q.isEmpty() && sum[q.peekLast()] >= sum[i])       // 右出
            q.pollLast();
        q.offerLast(i);                                           // 当前元素进队列
    }
    return res <= n ? res : -1;
}
```

5.2 1425. Constrained Subsequence Sum - Hard

Given an integer array nums and an integer k, return the maximum sum of a non-empty subsequence of that array such that for every two consecutive integers in the subsequence, nums[i] and nums[j], where i < j, the condition j - i <= k is satisfied.

A subsequence of an array is obtained by deleting some number of elements (can be zero) from the array, leaving the remaining elements in their original order.

5.3 239. Sliding Window Maximum - Hard

You are given an array of integers nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

```
public int[] maxSlidingWindow(int[] arr, int k) {
    int n = arr.length, startWindowIdx = 0;
    ArrayDeque<Integer> q = new ArrayDeque<>(); // 维持一个递减队列
    int [] ans = new int [n - k + 1];
    for (int i = 0; i < n; i++) {
        startWindowIdx = i-k+1;
        while (!q.isEmpty() && i - q.peekFirst() >= k) q.pollFirst();     // 左出 q: maintain k size window, 去头: 去掉 k wi
        while (!q.isEmpty() && arr[q.peekLast()] <= arr[i]) q.pollLast(); // 右出 q: 去掉递减队列尾部所有不大于当前值的元素, 就
        q.offerLast(i);  // 进 q: 进后此时 q.size() == k
        if (startWindowIdx >= 0)
            ans[startWindowIdx] = arr[q.peekFirst()]; // 使用递减队列左端最大值
    }
    return ans;
}
```

- 线段树的做法

```
// https://blog.csdn.net/Yaokai_AssultMaster/article/details/79599809
public class MaxSeg {
    List<Integer> tree = new ArrayList<>();
    int n;
    public MaxSeg (int [] arr) {
        n = arr.length;
        tree = new ArrayList<>(2*n);
        for (int i = 0; i < n; i++)
            tree.add(0);
        for (int i = 0; i < n; i++)
            tree.add(arr[i]); // same effect as below
        for (int i = n-1; i >= 0; i--) // i >= 0
            tree.set(i, Math.max(tree.get(2*i), tree.get(2*i+1)));
    }
    public void update(int idx, int v) {
        idx += n;
        tree.set(idx, v);
        while (idx > 1) {
            idx /= 2;
            tree.set(idx, Math.max(tree.get(2*idx), tree.get(2*idx+1)));
        }
    }
    public int getMax(int l, int r) {
        l += n;
        r += n;
        int max = Integer.MIN_VALUE;
        while (l < r) {
            if ((l & 1) == 1) {
                max = Math.max(max, tree.get(l));
                l++;
            }
            if ((r & 1) == 1) {
                r--;            // order matters !!!
                max = Math.max(max, tree.get(r));
            }
            l >>= 1;
            r >>= 1;
        }
        return max;
    }
}
```

```java
public int[] maxSlidingWindow(int[] arr, int k) {
    int n = arr.length;
    MaxSeg mat = new MaxSeg(arr);
    if (n == k) return new int [] {mat.getMax(0, n)};
    int [] res = new int [n-k+1];
    for (int i = 0; i+k <= n; i++)
        res[i] = mat.getMax(i, i+k);
    return res;
}
```

5.4 76. Minimum Window Substring

Given two strings s and t of lengths m and n respectively, return the minimum window substring of s such that every character in t (including duplicates) is included in the window. If there is no such substring, return the empty string "". The testcases will be generated such that the answer is unique. A substring is a contiguous sequence of characters within the string.

```java
private boolean satisfies(Map<Character, Integer> s, Map<Character, Integer> t) {
    if (s.size() < t.size()) return false;
    for (Map.Entry<Character, Integer> en : t.entrySet()) {
        if (!s.containsKey(en.getKey()) || s.containsKey(en.getKey()) && s.get(en.getKey()) < en.getValue()) return false;
    }
    return true;
}
public String minWindow(String s, String t) {
    int m = s.length();
    int n = t.length();
    if (m < n) return "";
    if (m == 1 && n == 1 && s.charAt(0) != t.charAt(0)) return "";
    if (n == 1) {
        boolean contains = false;
        for (char c : s.toCharArray()) {
            if (c == t.charAt(0)) {
                contains = true;
                break;
            }
        }
        return !contains ? "" : t;
    }
    Map<Character, Integer> mt = new HashMap<>();
    for (char c : t.toCharArray())
        mt.put(c, mt.getOrDefault(c, 0) + 1);
    Map<Character, Integer> ms = new HashMap<>();
    int l = 0, r = 0, i = 0, j = 0, pl = 0;
    String res = "", tmp = "";
    while (i < m) {
        while (i < m && !satisfies(ms, mt)) {
            ms.put(s.charAt(i), ms.getOrDefault(s.charAt(i), 0) + 1);
            ++i;
        }
        if (satisfies(ms, mt)) {
            tmp = s.substring(l, i);
            if (res.equals("") || res.length() > tmp.length()) res = tmp;
        }
        pl = l;
        while (l < i && satisfies(ms, mt)) {
            System.out.println("\nl: " + l);

            ms.put(s.charAt(l), ms.get(s.charAt(l))- 1);
            if (ms.get(s.charAt(l)) == 0) ms.remove(s.charAt(l));
            ++l;
        }
        if (satisfies(ms, mt) || pl != l) {
            tmp = s.substring(l-1, i);
            if (res.equals("") || res.length() > tmp.length()) res = tmp;
        }
        if (i == m) break;
    }
    return res;
}
```

5.5 632. Smallest Range Covering Elements from K Lists

You have k lists of sorted integers in non-decreasing order. Find the smallest range that includes at least one number from each of the k lists. We define the range [a, b] is smaller than range [c, d] if b - a < d - c or a < c if b - a == d - c.

```
// 时间复杂度：O(nk \log k)O(nklogk)，其中 nn 是所有列表的平均长度，kk 是列表数量。所有的指针移动的总次数最多是 nknk 次，每次从堆中
// 空间复杂度：O(k)O(k)，其中 kk 是列表数量。空间复杂度取决于堆的大小，堆中维护 kk 个元素。
public int[] smallestRange(List<List<Integer>> nums) {
    int n = nums.size();
    int resMin = 0, resMax = Integer.MAX_VALUE;
    int minRg = resMax - resMin;
    int max = Integer.MIN_VALUE;
    int [] next = new int [n]; // 各子链表中比当前 idx 位数值大的下一个数的下标，即 idx+1, 初始化全为 0
    Queue<Integer> q = new PriorityQueue<>(new Comparator<Integer>() {
            public int compare(Integer idx1, Integer idx2) {
                return nums.get(idx1).get(next[idx1]) - nums.get(idx2).get(next[idx2]);
            }
        });
    for (int i = 0; i < n; i++) {
        q.offer(i); // 0, 1, 2, .... n-1 真神奇呀
        max = Math.max(max, nums.get(i).get(0));
    }
    int minIdx = 0, curRg = 0;
    while (true) {
        minIdx = q.poll(); // 取出的是最小值的子链表的序号，而子链表里的当前最小值所在子链表中的位置存于 next[minIdx] 中
        curRg = max - nums.get(minIdx).get(next[minIdx]);
        if (curRg < minRg)  {
            minRg = curRg;
            resMin = nums.get(minIdx).get(next[minIdx]);
            resMax = max;
        }
        next[minIdx]++;
        if (next[minIdx] == nums.get(minIdx).size()) break;
        q.offer(minIdx); // 加回去，但是 queue 里真正比较的值已经变了，变强大了。。。 // 更新最小值的替换值
        max = Math.max(max, nums.get(minIdx).get(next[minIdx]));  // 更新最大值
    }
    return new int [] {resMin, resMax};
}
// 这里的 BB 序列是什么？我们可以用一个哈希映射来表示 BB 序列—— B[i]
// B[i] 表示 ii 在哪些列表当中出现过，
// 这里哈希映射的键是一个整数，表示列表中的某个数值，
// 哈希映射的值是一个数组，这个数组里的元素代表当前的键出现在哪些列表里。
// 如果列表集合为：
// 0: [-1, 2, 3]
// 1: [1]
// 2: [1, 2]
// 3: [1, 1, 3]
// 那么可以得到这样一个哈希映射
// -1: [0]
// 1: [1, 2, 3, 3]
// 2: [0, 2]
// 3: [0, 3]
public int[] smallestRange(List<List<Integer>> nums) {
    int n = nums.size();
    Map<Integer, List<Integer>> indices = new HashMap<>();
    int xmin = Integer.MAX_VALUE, xmax = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        for (int v : nums.get(i)) { // 把大链表中出出过的每一个值作键，值为它所存在于的子链表序号链表
            List<Integer> list = indices.getOrDefault(v, new ArrayList<>());
            list.add(i);
            indices.put(v, list);
            xmin = Math.min(xmin, v);
            xmax = Math.max(xmax, v); // 这里得到全局的最小最大值
        }
    }
    int [] freq = new int [n];
    int inside = 0; // cnt # of lists included in miniRanges
    int left = xmin, right = xmin -1;
    int resLeft = xmin, resRight = xmax;
    while (right < xmax) {
        right ++;
        if (indices.containsKey(right)) {
```

```
            for (int x : indices.get(right)) {
                freq[x]++;
                if (freq[x] == 1) inside++;
            }
            while (inside == n) { // find ONE satified solution, try to minimize the range
                if (right - left < resRight - resLeft) {
                    resLeft = left;
                    resRight = right;
                }
                if (indices.containsKey(left)) { // sliding the left size towards right
                    for (int v : indices.get(left)) {
                        freq[v]--;
                        if (freq[v] == 0) --inside;
                    }
                }
                left++;
            }
        }
    }
    return new int [] {resLeft, resRight};
}
```

5.6 Sliding Window Median

The median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle values. For examples, if arr = [2,3,4], the median is 3. For examples, if arr = [1,2,3,4], the median is (2 + 3) / 2 = 2.5. You are given an integer array nums and an integer k. There is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position. Return the median array for each window in the original array. Answers within 10-5 of the actual value will be accepted.

```
public double[] medianSlidingWindow(int[] nums, int k) {
    TreeMap<Integer, Integer> ma = new TreeMap<>();
    TreeMap<Integer, Integer> mb = new TreeMap<>();
    for (int i = 0; i < k; i++) {
        if (i % 2 == 0) {
            mb.put(nums[i], mb.getOrDefault(nums[i], 0) + 1);
            int n = mb.firstKey();
            if (mb.get(n) == 1) mb.remove(n);
            else mb.put(n, mb.get(n) - 1);
            ma.put(n, ma.getOrDefault(n, 0) + 1);
        } else {
            ma.put(nums[i], ma.getOrDefault(nums[i], 0) + 1);
            int n = ma.lastKey();
            if (ma.get(n) == 1) ma.remove(n);
            else ma.put(n, ma.get(n) - 1);
            mb.put(n, mb.getOrDefault(n, 0) + 1);
        }
    }
    double [] res = new double[nums.length-k+1];
    if (k % 2 == 1) res[0] = ma.lastKey();
    else res[0] =  (double)(((long)(ma.lastKey()) + (long)(mb.firstKey())) / 2.0);
    for (int i = 0; i + k < nums.length; i++) {
        ma.put(nums[i+k], ma.getOrDefault(nums[i+k], 0) + 1);
        int n = ma.lastKey();
        if (ma.get(n) == 1) ma.remove(n);
        else ma.put(n, ma.get(n) - 1);
        mb.put(n, mb.getOrDefault(n, 0) + 1);
        if (ma.containsKey(nums[i])) {
            if (ma.get(nums[i]) == 1) ma.remove(nums[i]);
            else ma.put(nums[i], ma.get(nums[i]) - 1);
            int v = mb.firstKey();
            if (mb.get(v) == 1) mb.remove(v);
            else mb.put(v, mb.get(v) - 1);
            ma.put(v, ma.getOrDefault(v, 0) + 1);
        } else {
            if (mb.get(nums[i]) == 1) mb.remove(nums[i]);
            else mb.put(nums[i], mb.get(nums[i]) - 1);
```

```
        }
        if (k % 2 == 1) res[i+1] = ma.lastKey();
        else res[i+1] = (double)(((long)(ma.lastKey()) + (long)(mb.firstKey())) / 2.0);
    }
    return res;
}
```

5.7 826. Most Profit Assigning Work

You have n jobs and m workers. You are given three arrays: difficulty, profit, and worker where:

difficulty[i] and profit[i] are the difficulty and the profit of the ith job, and worker[j] is the ability of jth worker (i.e., the jth worker can only complete a job with difficulty at most worker[j]). Every worker can be assigned at most one job, but one job can be completed multiple times.

For example, if three workers attempt the same job that pays $1, then the total profit will be $3. If a worker cannot complete any job, their profit is $0. Return the maximum profit we can achieve after assigning the workers to the jobs.

```
// 法一：暴力 TreeMap---O(n^2logn)
public int maxProfitAssignment(int[] difficulty, int[] profit, int[] worker) {
    TreeMap<Integer, Integer> m = new TreeMap<>();
    for (int i = 0; i < difficulty.length; i++)
        m.put(difficulty[i], i);
    int res = 0, idx;
    Integer low;
    for (int i = 0; i < worker.length; i++) {
        low = m.floorKey(worker[i]); // 使用 treemap 排序的特长
        if (low == null) continue;
        idx = m.get(low);
        for (int j = 0; j < difficulty.length; j++)
            if (difficulty[j] <= low && profit[j] >= profit[idx])
                idx = j;
        res += profit[idx];
    }
    return res;
}
// 法二：优化的 TreeMap---O(nlogn)
// 如果 TreeMap 里面保存的是每个 difficulty[i] 对应的最大的 profit，则就可以直接找 floorKey 对应的 value 就是对应的要找的 value;
// 那么只需要再遍历一次 TreeMap，将最大的到目前 key 位置最大的 value 放进去就行了
public int maxProfitAssignment(int[] difficulty, int[] profit, int[] worker) {
    TreeMap<Integer, Integer> m = new TreeMap<>();
    for (int i = 0; i < difficulty.length; i++)
        m.put(difficulty[i], Math.max(m.getOrDefault(difficulty[i], 0), profit[i]));
    int max = 0;
    for (Integer key : m.keySet()) {
        max = Math.max(max, m.get(key));
        m.put(key, max); //将最大的到目前 key 位置最大的 value 放进去
    }
    int res = 0;
    Integer low;
    for (int i = 0; i < worker.length; i++) {
        low = m.floorKey(worker[i]); // 使用 treemap 排序的特长
        if (low == null) continue;
        res += m.get(low);
    }
    return res;
}
// 法三：Sort + 双指针---O(nlogn)
// 思路就是先将 difficulty[] 和 profit 组成 pair，然后再将 list 和 worker[] 从小到大排序，然后遍历 worker，更新 tempMaxProfit，得
public int maxProfitAssignment(int[] difficulty, int[] profit, int[] worker) {
    List<int[]> list = new ArrayList<>();
    for (int i = 0; i < difficulty.length; i++)
        list.add(new int[] {difficulty[i], profit[i]});
    Collections.sort(list, (a, b) -> {return a[0] - b[0];});
    Arrays.sort(worker);
    int res = 0, tmpMaxProfit = 0;
    // i, j 同向双指针移动，更新到目前的 tmpMaxProfit
```

```
    for (int i = 0, j = 0; i < worker.length; i++) {
        while (j < list.size() && list.get(j)[0] <= worker[i]) {
            tmpMaxProfit = Math.max(tmpMaxProfit, list.get(j)[1]);
            j++;
        }
        //此时 tmpMaxProfit 是前面所有 difficulty 小于 worker[i] 的最大的 proifit
        res += tmpMaxProfit;
    }
    return res;
}
```

6 backTracking 回溯

6.1 491. Increasing Subsequences

Given an integer array nums, return all the different possible increasing subsequences of the given array with at least two elements. You may return the answer in any order.

The given array may contain duplicates, and two equal integers should also be considered a special case of increasing sequence.

```
private void dfs(int [] arr, int idx, List<Integer> l) {
    if (l.size() >= 2)
        res.add(new ArrayList<>(l));
    Set<Integer> vis = new HashSet<>();
    for (int i = idx; i < arr.length; i++) {
        if (vis.contains(arr[i])) continue;
        if (l.size() == 0 || arr[i] >= l.get(l.size()-1)) {
            vis.add(arr[i]);
            l.add(arr[i]);
            dfs(arr, i+1, l);
            l.remove(l.size()-1);
        }
    }
}
List<List<Integer>> res = new ArrayList<>();
public List<List<Integer>> findSubsequences(int[] arr) {
    if (arr == null || arr.length == 0) return res;
    dfs(arr, 0, new ArrayList<Integer>());
    return res;
}
```

7 排序与 recursion

常用排序算法							
类别	排序方法	时间复杂度			空间复杂度	稳定性	备注
		平均情况	最好情况	最坏情况	辅助存储		
插入排序	直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定	n小时较好
	shell排序	$O(n^{1.3})$	$O(n)$	$O(n^2)$	$O(1)$	不稳定	
选择排序	直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	n小时较好
	堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定	n大时较好
交换排序	冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定	n小时较好
	快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	不稳定	n大时较好
归并排序		$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	稳定	n大时较好
基数排序		$O(d(r+n))$	$O(d(n+rd))$	$O(d(r+n))$	$O(rd+n)$	稳定	
注：基数排序的复杂度中，r代表关键字的基数，d代表长度，n代表关键字的个数							

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(n\log_2 n)$	$O(n\log_2 n)$	不稳定
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

7.1 1996. The Number of Weak Characters in the Game 桶排序

You are playing a game that contains multiple characters, and each of the characters has two main properties: attack and defense. You are given a 2D integer array properties where properties[i] = [attacki, defensei] represents the properties of the ith character in the game. A character is said to be weak if any other character has both attack and defense levels strictly greater than this character's attack and defense levels. More formally, a character i is said to be weak if there exists another character j where attackj > attacki and defensej > defensei.

Return the number of weak characters.

```
public int numberOfWeakCharacters(int[][] properties) {
    int maxAttrack = 0; // 找到所有士兵中的最大值
    for (int[] p : properties)
        maxAttrack = Math.max(maxAttrack,p[0]);
    // 为每一个攻击创建一个桶的位置
    int[] bucket = new int[maxAttrack + 2];
    // 在每一个攻击力上找到最大的防御力
    for (int[] p : properties)
        bucket[p[0]] = Math.max(bucket[p[0]],p[1]);
    // 将桶的每一个位置都寻找到大于其攻击力的最大防御数值
    int rightMax = bucket[maxAttrack];
    for (int i = maxAttrack; i >= 0; i--)
        if (rightMax > bucket[i])
            bucket[i] = rightMax;
        else
            rightMax = bucket[i];
    int ans = 0;
    // 最后遍历 p 寻找所有的弱将
    for (int[] p : properties)
        if (bucket[p[0] + 1] > p[1]) ans++;  // 攻击力比当前小兵的攻击力大 1 的桶的位置存放着最大的防御力，与这个防御作比较可以得
    return ans;
}
public int numberOfWeakCharacters(int[][] properties) {
    int cnt = 0;
    int len = properties.length;
    Arrays.sort(properties, (a, b) -> (a[0] ! =  b[0] ? a[0]-b[0] : b[1]-a[1]));
    int max = properties[len-1][1];
    for(int i = len-1;i> = 0;i--){
        if(properties[i][1] < max)
            cnt++;
        max = Math.max(max,properties[i][1]);
    }
    return cnt;
}
```

7.2 632. Smallest Range Covering Elements from K Lists

You have k lists of sorted integers in non-decreasing order. Find the smallest range that includes at least one number from each of the k lists. We define the range [a, b] is smaller than range [c, d] if b - a < d - c or a < c if b - a == d - c.

```
// 时间复杂度：O(nk \log k)O(nklogk)，其中 nn 是所有列表的平均长度，kk 是列表数量。所有的指针移动的总次数最多是 nknk 次，每次从堆中
// 空间复杂度：O(k)O(k)，其中 kk 是列表数量。空间复杂度取决于堆的大小，堆中维护 kk 个元素。
public int[] smallestRange(List<List<Integer>> nums) {
    int n = nums.size();
    int resMin = 0, resMax = Integer.MAX_VALUE;
    int minRg = resMax - resMin;
    int max = Integer.MIN_VALUE;
    int [] next = new int [n]; // 各子链表中比当前 idx 位数值大的下一个数的下标，即 idx+1，初始化全为 0
    Queue<Integer> q = new PriorityQueue<>(new Comparator<Integer>() {
            public int compare(Integer idx1, Integer idx2) {
                return nums.get(idx1).get(next[idx1]) - nums.get(idx2).get(next[idx2]);
            }
        });
    for (int i = 0; i < n; i++) {
        q.offer(i); // 0, 1, 2, .... n-1 真神奇呀
        max = Math.max(max, nums.get(i).get(0));
    }
    int minIdx = 0, curRg = 0;
    while (true) {
        minIdx = q.poll(); // 取出的是最小值的子链表的序号，而子链表里的当前最小值所在子链表中的位置存于 next[minIdx] 中
        curRg = max - nums.get(minIdx).get(next[minIdx]);
        if (curRg < minRg)  {
            minRg = curRg;
            resMin = nums.get(minIdx).get(next[minIdx]);
            resMax = max;
        }
        next[minIdx]++;
```

```
        if (next[minIdx] == nums.get(minIdx).size()) break;
        q.offer(minIdx); // 加回去，但是 queue 里真正比较的值已经变了，变强大了。。。 // 更新最小值的替换值
        max = Math.max(max, nums.get(minIdx).get(next[minIdx]));  // 更新最大值
    }
    return new int [] {resMin, resMax};
}
```

- 另外一种解法的

```
// 这里的 BB 序列是什么？我们可以用一个哈希映射来表示 BB 序列—— B[i]
// B[i] 表示 ii 在哪些列表当中出现过，
// 这里哈希映射的键是一个整数，表示列表中的某个数值，
// 哈希映射的值是一个数组，这个数组里的元素代表当前的键出现在哪些列表里。
// 如果列表集合为：
// 0: [-1, 2, 3]
// 1: [1]
// 2: [1, 2]
// 3: [1, 1, 3]
// 那么可以得到这样一个哈希映射
// -1: [0]
// 1: [1, 2, 3, 3]
// 2: [0, 2]
// 3: [0, 3]
// 时间复杂度：O(nk + |V|)O(nk+V)，其中 nn 是所有列表的平均长度，kk 是列表数量，|V|V 是列表中元素的值域，在本题中 |V| \leq 2*10^5
// 5
// 。构造哈希映射的时间复杂度为 O(nk)O(nk)，双指针的移动范围为 |V|V，在此过程中会对哈希映射再进行一次遍历，时间复杂度为 O(nk)O(nk)，
// 空间复杂度：O(nk)O(nk)，即为哈希映射使用的空间。哈希映射的「键」的数量由列表中的元素个数 nknk 以及值域 |V|V 中的较小值决定，「值」
public int[] smallestRange(List<List<Integer>> nums) {
    int n = nums.size();
    Map<Integer, List<Integer>> indices = new HashMap<>();
    int xmin = Integer.MAX_VALUE, xmax = Integer.MIN_VALUE;
    for (int i = 0; i < n; i++) {
        for (int v : nums.get(i)) { // 把大链表中出出过的每一个值作键，值为它所存在于的子链表序号链表
            List<Integer> list = indices.getOrDefault(v, new ArrayList<>());
            list.add(i);
            indices.put(v, list);
            xmin = Math.min(xmin, v);
            xmax = Math.max(xmax, v); // 这里得到全局的最小最大值
        }
    }
    int [] freq = new int [n];
    int inside = 0; // cnt # of lists included in miniRanges
    int left = xmin, right = xmin -1;
    int resLeft = xmin, resRight = xmax;
    while (right < xmax) {
        right ++;
        if (indices.containsKey(right)) {
            for (int x : indices.get(right)) {
                freq[x]++;
                if (freq[x] == 1) inside++;
            }
            while (inside == n) { // find ONE satified solution, try to minimize the range
                if (right - left < resRight - resLeft) {
                    resLeft = left;
                    resRight = right;
                }
                if (indices.containsKey(left))  // sliding the left size towards right
                    for (int v : indices.get(left)) {
                        freq[v]--;
                        if (freq[v] == 0) --inside;
                    }
                left++;
            }
        }
    }
    return new int [] {resLeft, resRight};
}
```

7.3 Reverse Pairs

Given an integer array nums, return the number of reverse pairs in the array. A reverse pair is a pair (i, j) where 0 <= i < j < nums.length and nums[i] > 2 * nums[j].

```
private int mergeSortCount(long [] arr, int bgn, int end) {
if (bgn >= end) return 0;
int mid = bgn + (end-bgn)/2;
int cnt = mergeSortCount(arr, bgn, mid) + mergeSortCount(arr, mid+1, end);
    for (int i = bgn, j = mid+1; i <= mid; i++) {
        while (j <= end && arr[i] > 2*arr[j]) j++;
        cnt += j - (mid+1);
    }
    Arrays.sort(arr, bgn, end+1);
    return cnt;
}
public int reversePairs(int[] nums) {
    int n = nums.length;
    return mergeSortCount(Arrays.stream(nums).mapToLong(i -> i).toArray(), 0, n-1);
}
// bit 的解法： https://www.cnblogs.com/grandyang/p/6657956.html
```

7.4 306. Additive Number

Additive number is a string whose digits can form additive sequence.

A valid additive sequence should contain at least three numbers. Except for the first two numbers, each subsequent number in the sequence must be the sum of the preceding two.

Given a string containing only digits '0'-'9', write a function to determine if it's an additive number.

Note: Numbers in the additive sequence cannot have leading zeros, so sequence 1, 2, 03 or 1, 02, 3 is invalid.

```
public boolean isAdditiveNumber(String num) {
    int n = num.length();
    if (n < 3) return false;
    for (int i = 1; i <= num.length() >> 1; i++)
        for (int j = 1; j + i < num.length(); j++)
            if (isValid(num, num.substring(0, i), num.substring(i, i + j), i + j)) return true;
    return false;
}
private boolean isValid(String num, String first, String second, int index) {
    if (first.length() > 1 && first.startsWith("0")
        || second.length() > 1 && second.startsWith("0")) return false;
    if (index == num.length()) return true; // 如果只有两个数是有效的!!!
    long sum = Long.parseLong(first) + Long.parseLong(second);
    if (num.startsWith(sum + "", index)) // 间接检测第三个数
        if (isValid(num, second, sum + "", index + (sum + "").length())) return true;
    return false;
}
```

8 树结构：各种新型数据结构

8.1 968. Binary Tree Cameras

You are given the root of a binary tree. We install cameras on the tree nodes where each camera at a node can monitor its parent, itself, and its immediate children. Return the minimum number of cameras needed to monitor all nodes of the tree.

```
// 对于每个节点，有一下三种 case：
// case（1）：如果它有一个孩子，且这个孩子是叶子（状态 0），则它需要摄像头，res ++，然后返回 1，表示已经给它装上了摄像头。
// case（2）：如果它有一个孩子，且这个孩子是叶子的父节点（状态 1），那么它已经被覆盖，返回 2。
```

```java
// case (0): 否则，这个节点无孩子，或者说，孩子都是状态 2，那么我们将这个节点视为叶子来处理。
// 由于 dfs 最终返回后，整棵树的根节点的状态还未处理，因此需要判断，若根节点被视为叶子，需要在其上加一个摄像头。
private int dfs(TreeNode r) {
    // 空节点不需要被覆盖，归入情况 2
    if (r == null) return 2; // do not need cover
    int left = dfs(r.left);  // 递归求左右孩子的状态
    int right = dfs(r.right);
    // 获取左右孩子状态之后的处理
    // 有叶子孩子，加摄像头，归入情况 1
    if (left == 0 || right == 0) {
        res ++;
        return 1;
    }
    // 孩子上有摄像头，说明此节点已被覆盖，情况 2;
    if (left == 1 || right == 1) return 2;
    return 0;
}
int res = 0;
public int minCameraCover(TreeNode root) {
    // 若根节点被视为叶子，需要在其上加一个摄像头
    return (dfs(root) == 0 ? 1 : 0) + res;
}
```

8.2 1696. Jump Game VI

You are given a 0-indexed integer array nums and an integer k. You are initially standing at index 0. In one move, you can jump at most k steps forward without going outside the boundaries of the array. That is, you can jump from index i to any index in the range [i + 1, min(n - 1, i + k)] inclusive. You want to reach the last index of the array (index n - 1). Your score is the sum of all nums[j] for each index j you visited in the array. Return the maximum score you can get.

```java
public int maxResult(int[] nums, int k) { // O(N) DP with double ended queue
    int n = nums.length;
    int [] dp = new int[n];
    ArrayDeque<Integer> q = new ArrayDeque<>();
    for (int i = 0; i < n; i++) {
        while (!q.isEmpty() && q.peekFirst() < i-k) // 头大尾小
            q.removeFirst();
        dp[i] = nums[i] + (q.isEmpty() ? 0 : dp[q.peekFirst()]);
        while (q.size() > 0 && dp[q.peekLast()] <= dp[i])
            q.removeLast();
        q.addLast(i);
    }
    return dp[n-1];
}
public int maxResult(int[] nums, int k) { // BigO: O (NlogN)
    int n = nums.length;
    int [] dp = new int[n];
    Queue<int []> q = new PriorityQueue<>(Comparator.comparingInt(e -> -e[0]));
    for (int i = 0; i < n; i++) {
        while (!q.isEmpty() && q.peek()[1] + k < i)
            q.poll();
        dp[i] = nums[i] + (q.isEmpty() ? 0 : q.peek()[0]);
        q.add(new int[] {dp[i], i});
    }
    return dp[n-1];
}
```

8.3 Create Sorted Array through Instructions

Given an integer array instructions, you are asked to create a sorted array from the elements in instructions. You start with an empty container nums. For each element from left to right in instructions, insert it into nums. The cost of each insertion is the minimum of the following: The number of elements currently in nums that are strictly less than instructions[i].

The number of elements currently in nums that are strictly greater than instructions[i]. For example, if inserting element 3 into nums = [1,2,3,5], the cost of insertion is min(2, 1) (elements 1 and 2 are less than 3, element 5 is greater than 3) and nums will become [1,2,3,3,5]. Return the total cost to insert all elements from instructions into nums. Since the answer may be large, return it modulo 109 + 7

```
// https://blog.csdn.net/qq_28033719/article/details/112506925
private static int N = 100001;
private static int [] tree = new int [N]; // 拿元素值作为 key 对应 tree 的下标值
public int lowbit(int i) {
    return i & -i;
}
public void update(int i, int v) { // 更新父节点
    while (i <= N) {
        tree[i] += v;
        i += lowbit(i);
    }
}
public int getSum(int i) { // 得到以 i 为下标 1-based 的所有子、叶子节点的和，也就是 [1, i] 的和，1-based
    int ans = 0;
    while (i > 0) {
        ans += tree[i];
        i -= lowbit(i);
    }
    return ans;
}
public int createSortedArray(int[] instructions) {
    int n = instructions.length;
    long res = 0;
    Arrays.fill(tree, 0);
    for (int i = 0; i < n; i++) {
        //              严格小于此数的个数 严格大于此数的个数：为总个数（不含自己） - 小于自己的个数
        res += Math.min(getSum(instructions[i]-1), i-getSum(instructions[i]));
        update(instructions[i], 1);
    }
    return (int)(res % ((int)Math.pow(10, 9) + 7));
}
```

8.4 1932. Merge BSTs to Create Single BST: 这颗树我曾投入巨大热情，可是总不过，好缺德

You are given n BST (binary search tree) root nodes for n separate BSTs stored in an array trees (0-indexed). Each BST in trees has at most 3 nodes, and no two roots have the same value. In one operation, you can:

Select two distinct indices i and j such that the value stored at one of the leaves of trees[i] is equal to the root value of trees[j]. Replace the leaf node in trees[i] with trees[j]. Remove trees[j] from trees. Return the root of the resulting BST if it is possible to form a valid BST after performing n - 1 operations, or null if it is impossible to create a valid BST.

A BST (binary search tree) is a binary tree where each node satisfies the following property:

Every node in the node's left subtree has a value strictly less than the node's value. Every node in the node's right subtree has a value strictly greater than the node's value. A leaf is a node that has no children.

```
public TreeNode canMerge(List<TreeNode> trees) {
    final int size = trees.size();
    final Map<Integer, TreeNode> roots = new HashMap<>(size);
    for (final TreeNode node : trees)
        roots.put(node.val, node);
    for (final TreeNode node : trees) {
        if (roots.containsKey(node.val)) { // 这里判断：是因为接下来 buildTree 会将可以合并的子树键值对删除并回收利用建大树了
            final TreeNode root = buildTree(roots, node);
            roots.put(root.val, root);    // update root node
```

```
        }
    }
    if (roots.size() != 1) return null;   // 无法合并所有的子树
    final TreeNode root = roots.values().iterator().next(); // 只有这一颗树根
    return isValid(root, Integer.MIN_VALUE, Integer.MAX_VALUE) ? root : null;
}
private TreeNode buildTree(Map<Integer, TreeNode> roots, TreeNode node) { // 用 recursion 把所有需要/可以合并的子树建成一棵完整
    final TreeNode next = roots.remove(node.val); // map.remove() 返回值: 如果存在 key, 则删除并返回 value; 如果不存在则返回 nu
    if (next != null) {
        if (next.left != null) node.left = buildTree(roots, next.left);
        if (next.right != null) node.right = buildTree(roots, next.right);
    }
    return node;
}
private boolean isValid(TreeNode node, int min, int max) { // 这些个递归写得很传功力, 要活学活用到出神入化。。。。。
    if (node == null) return true;
    final int value = node.val;
    if (value <= min || value >= max) return false;
    return isValid(node.left, min, value) && isValid(node.right, value, max);
}
```

9 dfs

9.1 464. Can I Win

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

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

10 排列与组合

11 Union Find 并查集

11.1 1697. Checking Existence of Edge Length Limited Paths - Hard

An undirected graph of n nodes is defined by edgeList, where edgeList[i] = [ui, vi, disi] denotes an edge between nodes ui and vi with distance disi. Note that there may be multiple edges between two nodes.

Given an array queries, where queries[j] = [pj, qj, limitj], your task is to determine for each queries[j] whether there is a path between pj and qj such that each edge on the path has a distance strictly less than limitj .

Return a boolean array answer, where answer.length == queries.length and the jth value of answer is true if there is a path for queries[j] is true, and false otherwise.

```
private class DisjointSetUnion {
    private int N;
    private int [] parent, rank;
    public DisjointSetUnion( int n) {
        this.N = n;
        this.parent = new int [N];
        this.rank = new int [N];
        for (int i = 0; i < N; i++) {
            this.parent[i] = i;
            this.rank[i] = 1;
        }
    }
    public boolean areConnected(int u, int v) {
        return find(u) == find(v);
    }
    public void union(int u, int v) { // O(Log(N))
        if (u != v) {
            int p = find(u);
            int q = find(v);
            if (p != q) {
                if (rank[p] > rank[q]) {
                    parent[q] = p;
                    rank[p] += rank[q];
                } else {
                    parent[p] = q;
                    rank[q] += rank[p];
                }
            }
        }
    }
    private int find(int v) { // 我记得这里有个同步更新父数组的优化步骤，和 rank 优化冲突吗？要总结一下 // O(Log(N))
        int x = v;
        while (x != parent[x])
            x = parent[x];
        parent[v] = x;
        return x;
    }
}
class Query implements Comparable<Query> {
    public int idx, start, end, weight;
    public Query(int idx, int bgn, int end, int weight) {
        this.idx = idx;
        this.start = bgn;
        this.end = end;
        this.weight = weight;
    }
    @Override public int compareTo(Query query) {
        return this.weight - query.weight;
    }
}
// 我们可以分别对 edges 和 queries 进行一次升序排序。本题和 1170. 比较字符串最小字母出现频次 类似，都可以采取 * 离线排序优化 * 的方
// 接下来，遍历 queries。遍历 queries 的同时将权值小于 limitj 的边进行合并。
// 接下来，我们只需要判断 pj 和 qj 是否已经在同一个联通域即可。
```

```
// 因此如果 pj 和 qj 在同一个联通域，那么其联通的路径上的所有边必定都小于 limitj，其原因就是前面加粗的那句话。
// 注意到排序打乱了 queries 的索引，因此我们需要记录一下其原始索引。
public boolean[] distanceLimitedPathsExist(int n, int[][] edgeList, int[][] queries) {
    DisjointSetUnion set = new DisjointSetUnion(n);
    Arrays.sort(edgeList, (a, b) -> a[2]-b[2]); // arrange edges in ascending order of weights
    int m = queries.length, i;
    int E = edgeList.length;
    Query [] qarr = new Query[m];
    for (i = 0; i < m; i++)
        qarr[i] = new Query(i, queries[i][0], queries[i][1], queries[i][2]);
    Arrays.sort(qarr);                // arrange queries in ascending order of threshold weights
    boolean [] ans = new boolean [m];
    int idx = 0;
    for (i = 0; i < m; i++) {
        while (idx < E && edgeList[idx][2] < qarr[i].weight) {
            set.union(edgeList[idx][0], edgeList[idx][1]);
            ++idx;
        }
        ans[qarr[i].idx] = set.areConnected(qarr[i].start, qarr[i].end);
    }
    return ans;
}
// 令 m, q edges 和 queries 的长度。
//     时间复杂度：$O(mlogm + qlogq)
//     空间复杂度：$O(n + q)
// Runtime is bound by sorting: O(ElogE + NlogN + N + E);
```

11.2 2003. Smallest Missing Genetic Value in Each Subtree - Hard

There is a family tree rooted at 0 consisting of n nodes numbered 0 to n - 1. You are given a 0-indexed integer array parents, where parents[i] is the parent for node i. Since node 0 is the root, parents[3] == -1.

There are 105 genetic values, each represented by an integer in the inclusive range [1, 105]. You are given a 0-indexed integer array nums, where nums[i] is a distinct genetic value for node i.

Return an array ans of length n where ans[i] is the smallest genetic value that is missing from the subtree rooted at node i.

The subtree rooted at a node x contains node x and all of its descendant nodes.

```
private void dfs(int i, Set<Integer> visited, int [] arr) { // 图的遍历：自顶向下 ok，自底向上还不太熟悉，需要练习，还有一个没有
    if (!visited.contains(arr[i])) {
        Set<Integer> children = tree.getOrDefault(i, new HashSet<Integer>());
        for (int v : children)
            dfs(v, visited, arr);
        visited.add(arr[i]);
    }
}
Map<Integer, Set<Integer>> tree = new HashMap<>();
int [] ans;
int n;
public int[] smallestMissingValueSubtree(int[] parents, int[] nums) {
    n = parents.length;
    ans = new int [n];
    Arrays.fill(ans, 1);
    int oneIdx = -1;
    for (int i = 0; i < n; i++)
        if (nums[i] == 1) {
            oneIdx = i;
            break;
        }
    if (oneIdx == -1) return ans;
    for (int i = 1; i < n; i++) {
        tree.computeIfAbsent(parents[i], k -> new HashSet<Integer>());
        tree.get(parents[i]).add(i);
        // tree.computeIfAbsent(i, k -> new HashSet<Integer>()); // 这里要想一下：为什么双向图他只加一个方向？
        // tree.get(i).add(parents[i]);
    }
    Set<Integer> visited = new HashSet<>(); // 这个直接转化为想要的结果，很便捷
```

```
    int parentIter = oneIdx;
    int miss = 1;
    while (parentIter >= 0) { // 从值为 1 的节点向根遍历（自底向上），没有任何重复计算，只走完这一条自底向顶的路径就可以了
        dfs(parentIter, visited, nums);
        while (visited.contains(miss)) ++miss;
        ans[parentIter] = miss;
        parentIter = parents[parentIter];
    }
    return ans;
}
```

11.3 721. Accounts Merge

Given a list of accounts where each element accounts[i] is a list of strings, where the first element accounts[i][3] is a name, and the rest of the elements are emails representing emails of the account.

Now, we would like to merge these accounts. Two accounts definitely belong to the same person if there is some common email to both accounts. Note that even if two accounts have the same name, they may belong to different people as people could have the same name. A person can have any number of accounts initially, but all of their accounts definitely have the same name.

After merging the accounts, return the accounts in the following format: the first element of each account is the name, and the rest of the elements are emails in sorted order. The accounts themselves can be returned in any order.

- Similar to the most voted solution, my solution uses the index of the owner as the key for the union map, to avoid possible issue caused by different owners have the same name, thus uses one less loop.

```
private int findParent(int [] arr, int x) {
    if (arr[x] == x) return x;
    arr[x] = findParent(arr, arr[x]);
    return arr[x];
}
public List<List<String>> accountsMerge(List<List<String>> accounts) {
    Map<String, Integer> owner = new HashMap<>();
    Map<Integer, TreeSet<String>> union = new HashMap<>(); // match idx & Set<Sting emails>
    int n = accounts.size(), p = 0;
    int [] par = new int [n];
    for (int i = 0; i < n; i++) par[i] = i;
    List<String> ls = new ArrayList<>();
    for (int i = 0; i < n; i++) { // find the ownerIdx for each email address
        ls = accounts.get(i);
        for (int j = 1; j < ls.size(); j++) {
            String email = ls.get(j);
            if (owner.containsKey(email)) {
                p = findParent(par, owner.get(email));
                par[p] = i; // union accounts that belong to the same user here by updating parent relation
            }
            owner.put(email, i);
        }
    }
     // union all emails belong to the same owner
    for (String emal : owner.keySet()) {
        int ownerIdx = findParent(par, owner.get(emal));
        TreeSet<String> set = union.getOrDefault(ownerIdx, new TreeSet<>());
        set.add(emal);
        union.put(ownerIdx, set);
    }
// Generate return result
    List<List<String>> res = new ArrayList<>();
    for (int ownerIdx : union.keySet()) {
        ls = new ArrayList<>();
        ls.add(accounts.get(ownerIdx).get(0));  // get the owner name
        ls.addAll(union.get(ownerIdx));
```

```
        res.add(ls);
    }
    return res;
}
```

11.4 1202. Smallest String With Swaps

You are given a string s, and an array of pairs of indices in the string pairs where pairs[i] = [a, b] indicates 2 indices(0-indexed) of the string.

You can swap the characters at any pair of indices in the given pairs any number of times.

Return the lexicographically smallest string that s can be changed to after using the swaps.

```
int [] par;
int [] rank;
int n;
public int find(int v) {
    if (v != par[v] )
        par[v] = find(par[v]);
    return par[v];
}
public boolean union(int i, int j) {
    int ri = find(i);
    int rj = find(j);
    if (ri == rj) return false;
    if (rank[ri] < rank[rj]) par[ri] = rj;
    else if (rank[ri] > rank[rj]) par[rj] = ri;
    else {
        par[rj] = ri;
        rank[ri] ++; // 维护 rank 的值
    }
    return true;
}
public String smallestStringWithSwaps(String s, List<List<Integer>> pairs) {
    int n = s.length();
    par = new int [n];
    rank = new int [n];
    List<Queue<Character>> list = new ArrayList<>(n);
    for (int i = 0; i < n; i++) {
        par[i] = i;
        list.add(new PriorityQueue<>());
    }
    Arrays.fill(rank, 1);
    pairs.forEach(p -> union(p.get(0), p.get(1))); // Perform union for each pair.
    // Add each character to the priority queue associated with its component.
    IntStream.range(0, n).forEach(index -> list.get(find(index)).add(s.charAt(index)));
    // Build the result, by removing chars from the corresponding priority queue.
    StringBuilder buffer = new StringBuilder(n);
    IntStream.range(0, n).forEachOrdered(index -> buffer.append(list.get(find(index)).remove()));
    return buffer.toString();
}
// O(NlogN). Worst-case, all indices are part of the same component. So we will essentially be popping off from the same pr
// Space Complexity: O(N).
```

11.5 1998. GCD Sort of an Array - Hard

You are given an integer array nums, and you can perform the following operation any number of times on nums:

Swap the positions of two elements nums[i] and nums[j] if gcd(nums[i], nums[j]) > 1 where gcd(nums[i], nums[j]) is the greatest common divisor of nums[i] and nums[j]. Return true if it is possible to sort nums in non-decreasing order using the above swap method, or false otherwise.

```
private int find (int v) {
    if (!parent.containsKey(v)) {
        parent.put(v, v);
        return v;
```

```
    }
    if (parent.get(v) != v)
        parent.put(v, find(parent.get(v)));
    return parent.get(v);
}
private void union(int x, int y) {
    int rx = find(x);
    int ry = find(y);
    if (rx != ry) parent.put(rx, ry);
}
/**
   general idea (not accepted)
   we can simply union pairs of numbers which has gcd > 1 in quadratic time and then check of groups that
   are formed by union of pairs can be invidually sorted.
   improved (accpeted)
   In above approach problem is we are union-ing pairs in quadratic time. To improve upon it. We union a number
   which is present in 'nums' with its smallest prime factor. thus if two numbers has same smallest prime factor
   their gcd is guaranted to be > 1.
**/
Map<Integer, Integer> parent = new HashMap<>();
public boolean gcdSort(int[] arr) {
    int n = arr.length;
    parent = new HashMap<Integer, Integer>();
    int [] sorted = arr.clone();
    Arrays.sort(sorted);
    int max = Arrays.stream(arr).max().getAsInt();
    Set<Integer> numSet = new HashSet<>();
    numSet.addAll(Arrays.stream(arr).boxed().collect(Collectors.toList()));
    int p = 2;  // Seive algorithm
    boolean [] primes = new boolean [max + 1];
    Arrays.fill(primes, true);
    while (p < max) {
        if (primes[p]) {
            for (int i = p; i <= max; i += p) { // 我合并的是数组的索引，他优化成合并所有拥有公约数为 p 的数组中沿未合并的值
                if (numSet.contains(i)) union(p, i);
                primes[i] = false;
            }
        }
        p++;
    }
    for (int i = 0; i < n; i++)
        if (arr[i] != sorted[i] && find(sorted[i]) != find(arr[i])) return false;
    return true;
}
```

11.6 1724 给定一个 n nn 个顶点的无向带权图，要求在线回答若干询问，每次询问是个三元组 (p , q , x) (p,q,x)(p,q,x)，是问是否存在 p pp 到 q qq 的每条边都小于 x xx 的路径。

先以 Kruskal 算法求最小生成森林，显然对于任何 p pp 和 q qq，它们之间所有路径中最大边最小的那条路径的最大边一定是最小生成树的某条边（如果存在路径的话）。建树完成之后，对每个连通块，以任意顶点为根，将该连通块做成一棵有根树。对于每次询问，我们只需求 p pp 和 q qq 所有到它们的最近公共祖先所经过的边的最大边权就行了。可以考虑用倍增思想（以下的内容可以参考https://blog.csdn.net/qq_46105170/article/details/116217633f ff 和 g gg，其中 f [i] [k] f[i][k]f[i][k] 指的是从 i ii 节点向上跳 2 k 2^{k2} k 步能过走到的顶点是谁（如果跳出界了则规定值为 1 -11），g [i] [k] g[i][k]g[i][k] 指的是从 i ii 节点向上跳 2 k 2^{k2} k 的过程中经过的边的最大权值（在不会跳出界的情况下），然后从每个树根做 BFS，初始化 f [.] [0] f[.]^{3}f[.]^{3}和 g [.] [0] g[.]^{3}g[.]^3。由于在询问的时候需要知道最近公共祖先，所以还需要一个数组 d dd 记录每个顶点的深度，在 BFS 的时候可以同时求出。接下来，用倍增的思想计算 f ff 和 g gg：

```
f[i][k] = f[f[i][k-1]][k-1]
g[i][k] = max{g[i][k-1],g[f[i][k-1]][k-1]}
```

至此，所有的预处理就完成了。

接下来询问的时候，如果 p pp 与 q qq 不连通则直接返回 false。否则看一下两个顶点的深度，不妨设 p pp 更深，则将 p pp 向上跳若干步，使得 p pp 与 q qq 一样深，同时用经过的边权更新答案。此时如果 p = q p=qp=q，则答案已经求出，与 x xx 比较即可；否则，将 p pp 与 q qq 继续向上跳，一路跳到它们的最近公共祖先的孩子的那层位置，一路更新答案，最后再用最近公共祖先与它们的连边更新答案，最后将答案与 x xx 比较。代码如下：

```
public class DistanceLimitedPathsExist {
    class UnionFind {
        private int[] p;
        public UnionFind(int size) {
            p = new int[size];
            for (int i = 0; i < size; i++)
                p[i] = i;
        }
        public int find(int x) {
            if (p[x] != x)
                p[x] = find(p[x]);
            return p[x];
        }
        public void union(int x, int y) {
            int px = find(x), py = find(y);
            if (px != py)
                p[px] = py;
        }
    }
    private UnionFind uf;
    // 这里是链式前向星建图
    private int[] h, e, ne, w;
    private int idx;
    private void add(int a, int b, int c) {
        e[idx] = b;
        ne[idx] = h[a];
        w[idx] = c;
        h[a] = idx++;
    }
    // 这里是倍增法所需的数组和变量
    private int[][] f, g;
    private int[] depth;
    // n 是顶点数，log 是 n 对 2 的对数，log + 1 也是 f 的第二维应该开的长度
    private int n, log;
    public DistanceLimitedPathsExist(int n, int[][] edgeList) {
        this.n = n;
        depth = new int[n];
        h = new int[n];
        Arrays.fill(h, -1);
        // 无向图，边要开两倍
        e = new int[n << 1];
        ne = new int[n << 1];
        w = new int[n << 1];
        // 这一段是 Kruskal 算法建最小生成森林
        Arrays.sort(edgeList, (e1, e2) -> Integer.compare(e1[2], e2[2]));
        uf = new UnionFind(n);
        for (int[] e : edgeList) {
            int a = e[0], b = e[1], len = e[2];
            if (uf.find(a) != uf.find(b)) {
                uf.union(a, b);
                add(a, b, len);
                add(b, a, len);
            }
        }
        log = (int) (Math.log(n) / Math.log(2));
        f = new int[n][log + 1];
        g = new int[n][log + 1];
        for (int[] row : f) {
            Arrays.fill(row, -1);
        }
        boolean[] vis = new boolean[n];
        for (int i = 0; i < n; i++) {
            if (!vis[i]) {
                bfs(i, vis);
            }
        }
        init();
```

```
    }
    // 递推一遍 f 和 g 数组
    private void init() {
        for (int i = 1; i < log + 1; i++) {
            for (int j = 0; j < n; j++) {
                if (f[j][i - 1] != -1) {
                    f[j][i] = f[f[j][i - 1]][i - 1];
                    g[j][i] = Math.max(g[j][i - 1], g[f[j][i - 1]][i - 1]);
                }
            }
        }
    }
    // BFS 一遍 x 所在连通块，并初始化 f 和 g 数组，并求出 depth 数组
    private void bfs(int x, boolean[] vis) {
        Queue<Integer> q = new ArrayDeque<>();
        q.offer(x);
        vis[x] = true;
        while (!q.isEmpty()) {
            int u = q.poll();
            for (int i = h[u]; i != -1; i = ne[i]) {
                int v = e[i];
                if (vis[v]) continue;
                vis[v] = true;
                f[v][0] = u;
                g[v][0] = w[i];
                q.offer(v);
                depth[v] = depth[u] + 1;
            }
        }
    }
    public boolean query(int p, int q, int limit) {
        if (uf.find(p) != uf.find(q))
            return false;
        if (depth[p] < depth[q]) {
            int tmp = p;
            p = q;
            q = tmp;
        }
        // 先走到同一深度
        int diff = depth[p] - depth[q];
        int pow = 0, max = 0;
        while (diff > 0) {
            if ((diff & 1) == 1) {
                max = Math.max(max, g[p][pow]);
                p = f[p][pow];
            }

            pow++;
            diff >>= 1;
        }
        // 已经走到同一点了，那深度更浅的那个点就是最近公共祖先，max 就是经过的边的最大值
        if (p == q) return max < limit;
        // 否则跳到最近公共祖先下面一层，沿途更新答案
        for (int i = log; i >= 0; i--) {
            if (f[p][i] != f[q][i]) {
                max = Math.max(max, g[p][i]);
                max = Math.max(max, g[q][i]);
                p = f[p][i];
                q = f[q][i];
            }
        }
        // 最后别忘了用最后一步更新答案
        max = Math.max(max, g[p][0]);
        max = Math.max(max, g[q][0]);
        return max < limit;
    }
}
// 初始化时间复杂度 O ( m log  m + n log  n ) O(m\log m+n\log n)O(mlogm+nlogn)，每次询问时间 O ( log  n ) O(\log n)O(logn)，空
```

11.7 1172. 祖孙询问

给定一棵包含 n nn 个节点的有根无向树，节点编号互不相同，但不一定是 1 n 1n1n。有 m mm 个询问，每个询问给出了一对节点的编号 x xx 和 y yy，询问 x xx 与 y yy 的祖孙关系。

输入格式：输入第一行包括一个整数表示节点个数；接下来 n nn 行每行一对整数 a aa 和 b bb，表示 a aa 和 b bb 之间有一条无向边。如果 b bb 是 1 11，那么 a aa 就是树的根；第 n + 2 n+2n+2 行是一个整数 m mm 表示询问个数；接下来 m mm 行，每行两个不同的正整数 x xx 和 y yy，表示一个询问。

输出格式：对于每一个询问，若 x xx 是 y yy 的祖先则输出 1 11，若 y yy 是 x xx 的祖先则输出 2 22，否则输出 0 00。

数据范围：1 n , m 4 Œ 1 0 4 1n,m4Œ10^{41\ n},m4Œ10 4

1 v 4 Œ 1 0 4 1v4Œ10^{41\ v\ 4\times10} 4 ，v vv 是顶点编号

可以用倍增的思想来求。预处理两个数组，一个是 d [i] d[i]d[i]，指的是顶点 i ii 的深度，树根深度是 1 11，其余顶点的深度就是其与树根的路径边数；另一个是 f [i] [k] f[i][k]f[i][k]，是从顶点 i ii 向树根方向（以下均称“向上”）跳 2 k 2^{k2} k 步走到的顶点。由于顶点编号都是大于 0 00 的，我们可以人为规定一个 0 00 号节点作为哨兵，并且 k , f [r] [k] = 0 ∀ k,f[r][k]=0k,f[r][k]=0，d [0] = 0 d^3=0d3=0，其中 r rr 是树根编号。这样如果 k kk 太大导致跳出树根的话，就会得到跳到了哨兵的结论。那么对于 f ff，有 f [i] [0] = p i f[i]^3=p_{if[i]}3= 是 i ii 的父亲，并且：

f[i][k]=f[f[i][k1]][k1]

即分两次跳，一次跳 2^k 步等价于两次跳 2^k-1 步。初始化 f ff 和 d dd 数组的过程，可以用一次从树根的 BFS 来做到。

在询问的时候，比如询问 a aa 和 b bb 的公共祖先，不妨设 a aa 的深度更深，那么先让 a aa 向上跳到与 b bb 深度相同，可以先计算一下 d [a] d [b] d[a]-d[b]d[a]d[b]，然后将这个数字做二进制分解，就可以由 f ff 数组算出从 a aa 向上走到与 b bb 深度相同的时候是哪个顶点；接着再从 a aa 和 b bb 一起向上走，直到走到它们真正的最近公共祖先的下一层为止（即跳的步数是当前深度差减 1 11），此时 f [a] [0] f[a]^3f[a]^3即为最近公共祖先。代码如下：

```
const int N = 40010, M = N * 2;
int n, m;
int h[N], e[M], ne[M], idx;
int depth[N], fa[N][16];
int q[N];
void add(int a, int b) {
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}
// 从树根开始 bfs，预处理出 d 数组和 f 数组
void bfs(int root) {
    memset(depth, 0x3f, sizeof depth);
    depth[0] = 0, depth[root] = 1;
    int hh = 0, tt = 0;
    q[tt++] = root;
    while (hh < tt) {
        int t = q[hh++];
        for (int i = h[t]; ~i; i = ne[i]) {
            int j = e[i];
            if (depth[j] > depth[t] + 1) {
                depth[j] = depth[t] + 1;
                q[tt++] = j;
                // 预处理 f 数组
                fa[j][0] = t;
                for (int k = 1; k <= 15; k++)
                    fa[j][k] = fa[fa[j][k - 1]][k - 1];
            }
        }
    }
}
int lca(int a, int b) {
^^I// 强制让 a 的深度大于等于 b 的深度
    if (depth[a] < depth[b]) swap(a, b);
    // 从大到小枚举 k，将 a 向上跳到与 b 同层
    for (int k = 15; k >= 0; k--)
        if (depth[fa[a][k]] >= depth[b])
```

```
            a = fa[a][k];
^^I// 如果 a 和 b 重合了，那么说明 b 就是最近公共祖先
    if (a == b) return b;
    // 否则将 a 和 b 同时向上跳，直到跳到最近公共祖先下一层为止
    for (int k = 15; k >= 0; k--)
        if (fa[a][k] != fa[b][k])
            a = fa[a][k], b = fa[b][k];
    return fa[a][0];
}
int main() {
    scanf("%d", &n);
    memset(h, -1, sizeof h);
    int root = 0;
    for (int i = 0; i < n; i++) {
        int a, b;
        scanf("%d%d", &a, &b);
        if (b == -1) root = a;
        else add(a, b), add(b, a);
    }
    bfs(root);
    scanf("%d", &m);
    while (m--) {
        int a, b;
        scanf("%d%d", &a, &b);
        int p = lca(a, b);
        if (p == a) puts("1");
        else if (p == b) puts("2");
        else puts("0");
    }
    return 0;
}
```

11.8 某个网友列的相关题目：并查集参考：数据结构-并查集（**Disjoint-Set**）

- 261. 以图判树（全部连通 + 边数 =V-1）
- 305. 岛屿数量 II（并查集）
- 323. 无向图中连通分量的数目（并查集）
- 684. 冗余连接（并查集）
- 685. 冗余连接 II（并查集）
- 721. 账户合并（并查集）（字符串合并）
- 737. 句子相似性 II（并查集）
- 886. 可能的二分法（着色 DFS/BFS/拓展并查集）
- 947. 移除最多的同行或同列石头（并查集）
- 990. 等式方程的可满足性（并查集）
- 959. 由斜杠划分区域（并查集）
- 1061. 按字典序排列最小的等效字符串（并查集）
- 1101. 彼此熟识的最早时间（排序 + 并查集）
- 1202. 交换字符串中的元素（并查集）
- 1319. 连通网络的操作次数（BFS/DFS/并查集）
- 5510. 保证图可完全遍历（并查集）
- 程序员面试金典 - 面试题 17.07. 婴儿名字（并查集）

12 Segment Tree

12.1 1157. Online Majority Element In Subarray - Hard

Design a data structure that efficiently finds the majority element of a given subarray.

The majority element of a subarray is an element that occurs threshold times or more in the subarray.

Implementing the MajorityChecker class:

MajorityChecker(int[] arr) Initializes the instance of the class with the given array arr. int query(int left, int right, int threshold) returns the element in the subarray arr[left...right] that occurs at least threshold times, or -1 if no such element exists.

```
private class Node {
    private int bgn;
    private int end;
    private int val;
    private int cnt; // sum
    private Node left;
    private Node right;
    public Node(int bgn, int end, int val, int cnt) {
        this.bgn = bgn;
        this.end = end;
        this.val = val;
        this.cnt = cnt;
        this.left = null;
        this.right = null;
    }
    public Node(int bgn, int end, int val, int cnt, Node left, Node right) {
        this.bgn = bgn;
        this.end = end;
        this.val = val;
        this.cnt = cnt;
        this.left = left;
        this.right = right;
    }
}
public void update(int index, int val) {
    updateTree(root, index, val);
}
public int cntRange(int left, int right) {
    return cntRangeFromTree(root, left, right);
}
private int cntRangeFromTree(Node r, int i, int j) {
    if (r == null || i < r.bgn || i > r.end) return 0;
    else if (i <= r.bgn && j >= r.end) return r.cnt;
    else return cntRangeFromTree(r.left, i, j) + cntRangeFromTree(r.right, i, j);
}
private void updateTree(Node r, int i, int va) {
    if (r == null || i < r.bgn || i > r.end) return;
    else if (r.bgn == r.end && r.bgn == i) r.cnt = va;
    else {
        updateTree(r.left, i, va);
        updateTree(r.right, i, va);
        int cnt = 0;
        if (r.left != null) cnt += r.left.cnt;
        if (r.right != null) cnt += r.right.cnt;
        r.cnt = cnt;
    }
}
private Node buildTree(int [] arr, int i, int j) {
    if (i > j) return null;
    else if (i == j)
        return new Node(i, i, arr[i], 1);
    else {
        int mid = i + (j-i)/2;
        Node left = buildTree(arr, i, mid);
        Node right = buildTree(arr, mid+1, j);
        if (left.val == right.val)
            return new Node(i, j, left.val, left.cnt + right.cnt, left, right);
        else {
```

```
            if (left.cnt > right.cnt)
                return new Node(i, j, left.val, left.cnt-right.cnt, left, right);
            else return new Node(i, j, right.val, right.cnt-left.cnt, left, right);
        }
    }
}
// 排序数组中 第一个大于 tar 的下标
int upper_bound(List<Integer> list, int tar) {
    int l = 0, r = list.size();
    while (l < r) {
        int mid = l + (r-l)/2;
        if (list.get(mid) <= tar) l = mid+1;
        else r = mid;
    }
    return l;
}
// 排序数组中 第一个大于等于 tar 的下标
int lower_bound(List<Integer> list, int tar) {
    int l = 0, r = list.size()-1;
    while (l < r) {
        int mid = l + (r-l)/2;
        if (list.get(mid) < tar) l = mid+1;
        else r = mid;
    }
    return l;
}
/**
 * 构建线段树
 * @param arr 被构建数组
 * @param l 构建节点的左值 表示查询区域左边界
 * @param r 构建节点的右值 表示查询区域右边界
 * @return 以构建完成的线段树节点
 * */
private SegTreeNode buildTree(int[] arr, int l, int r) {
    if (l > r) return null;
    // 初始一个线段树节点
    SegTreeNode root = new SegTreeNode(l, r);
    // 叶子节点
    if (l == r) {
        // 众数就是当前值 计数为 1
        root.val = arr[l]; root.count = 1;
        return root;
    }

    int mid = (l+r)/2;
    // 构建左子节点
    root.left = buildTree(arr, l, mid);
    // 构建右子节点
    root.right = buildTree(arr, mid+1, r);
    // 整合父节点
    makeRoot(root);
    return root;
}
/**
 * 整合一个父节点
 * @param root 被整合节点
 * */
private void makeRoot(SegTreeNode root) {
    if (null == root) return;
    // 如果该节点有左子节点 该节点的值" 先" 等于左子节点
    if (root.left != null) {
        root.val = root.left.val;
        root.count = root.left.count;
    }
    // 如果该节点还有右子节点 融合父节点和子节点
    if (root.right != null) {
        if (root.val == root.right.val)
            root.count = root.count + root.right.count;
        else {
            if (root.count >= root.right.count)
                root.count = root.count - root.right.count;
            else {
                root.val = root.right.val;
                root.count = root.right.count - root.count;
            }
```

```
        }
    }
}
/**
 * 查询线段树
 * @param root 被查询节点
 * @param l 需要查询的范围左边界
 * @param r 需要查询的范围右边界
 * */
private void searchSegTree(Node root, int l, int r) {
    if (root == null || l > r) return;
    if (root.bgn > r || root.end < l) return;

    // 当查询边界 覆盖 节点边界 该节点就是查询区域
    if (root.bgn >= l && root.end <= r) {
        if (key == root.val) cnt += root.cnt;
        else if (cnt <= root.cnt) {
            key = root.val;
            cnt = root.cnt - cnt;
        } else cnt = cnt - root.cnt;
        return;
    }

    int mid = (root.end + root.bgn)/2;
    // root.bgn <= l <= mid 左节点也可以是查询区域
    if (l <= mid)  // 这两个查询条件再好好想想 !!!!!!!!!!!!!!!
        searchSegTree(root.left, l, r);
    // mid+1 <= r <= root.end 右节点也可以是查询区域
    if (r >= mid+1)
        searchSegTree(root.right, l, r);
}
// https://books.halfrost.com/leetcode/ChapterFour/1100~1199/1157.Online-Majority-Element-In-Subarray/ 也有一个直观图
// https://www.cnblogs.com/slowbirdoflsh/p/11381565.html 思路比较清晰
HashMap<Integer, List<Integer>> idx = new HashMap<>();
private Node root;
int key = 0, cnt = 0;
public MajorityChecker(int[] arr) {
    root = buildTree(arr, 0, arr.length-1);
    levelPrintTree(root);
    idx = new HashMap<>();
    for (int i = 0; i < arr.length; i++) {
        if (!idx.containsKey(arr[i]))
            idx.put(arr[i], new ArrayList<>());
        idx.get(arr[i]).add(i);
    }
}
public int countRangeSum(int[] nums, int lower, int upper) {
    MajorityChecker mc = new MajorityChecker(nums);
}
public int query(int left, int right, int threshold) {
    // 初始化 所查询众数 key 及辅助判断的计数 cnt
    key = 0; cnt = 0;
    // 查询线段树
    searchSegTree(root, left, right);
    // 如果查询区域没有众数 即 key 没被更改
    // 或者
    // 所查询出来的众数 在原数组中根本没有超出阈值的能力
    System.out.println("key: " + key);
    System.out.println("(idx.get(key) == null): " + (idx.get(key) == null));

    if (key == 0 || idx.get(key).size() < threshold) return -1;

    // 上确界 排序数组中 第一个大于 right 的下标
    int r = upper_bound(idx.get(key), right);
    // 下确界 排序数组中 第一个大于等于 left 的下标
    int l = lower_bound(idx.get(key), left);
    cnt = r - l;
    return cnt >= threshold ? key : -1;
}
```

---

13 Trie

应用 Trie 树最直观的定义就是 LinkedList of HashMap。所以 Trie 和 HashMap 都可以用来查询某个单词是否在字典当中。我们需要知道他们的优缺点。优点：支持字符级别的查询，比如说我们需要在 matrix 当中通过 traverse 构造单词，那么这个单词是一个一个字符形成的，我们可以在 traverse 的每一步去检验当前路径是否可以形成 valid word。另外，对于含有 regex 符号的字符串，我们需要一个字符一个字符的考虑，这种情况下我们也需要通过 trie 去查找。节省空间，相同的 prefix 只存一遍，而 HashMap 需要存很多遍。缺点：实现起来较麻烦，大部分题目使用 Trie 都是 overkill，所以除非需要支持字符级别的查询，否则 HashMap 更好。操作: 三个操作：insert search startWith 其中 insert 记得把最后一个 node 标记为 isEnd = true。其中 search 和 startWith 都可以通过同一个 searchHelper helper method 来实现，我们只需要 return 最后一个 node 就可以，如果 isEnd == true，那么说明找到一个完整的单词，否则至少找到了 prefix。别忘了使用 trie 的第一步是 preprocess，把字典里的所有 word 加入到 trie 树当中。题目

13.1 208. Implement Trie (Prefix Tree)

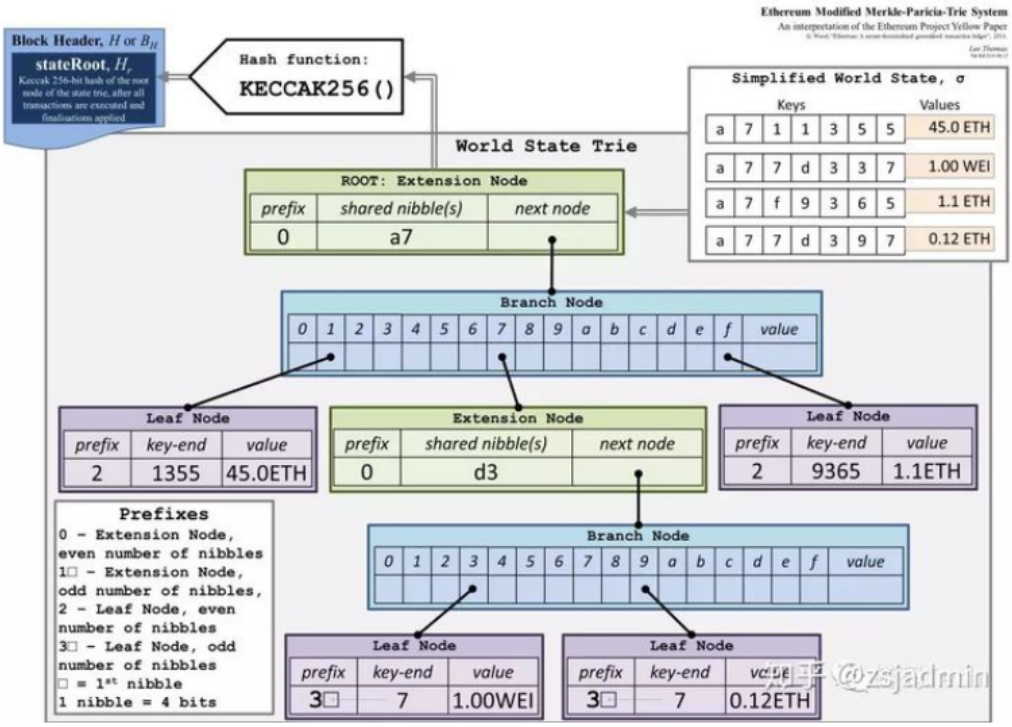
13.2 212. Word Search II

13.3 211. Add and Search Word - Data structure design (Facebook 店面)

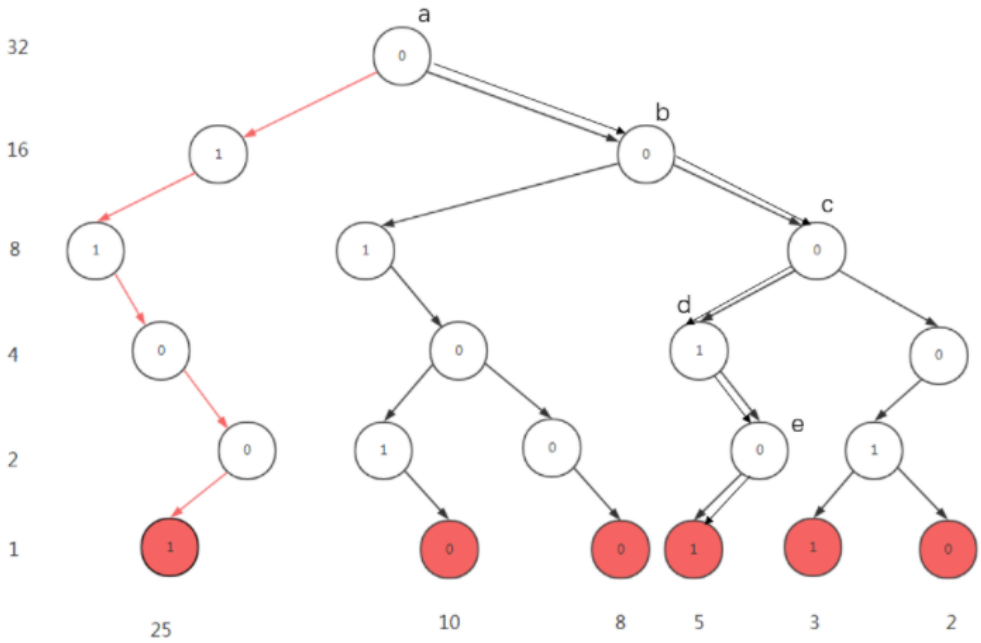
13.4 14. Longest Common Prefix (这道题可以稍作改编，比如说 **string list** 会经常 **update**，会经常 **query**，那这时很明显用 **trie** 更好)

13.5 421. Maximum XOR of Two Numbers in an Array

Given an integer array nums, return the maximum result of nums[i] XOR nums[j], where 0 <= i <= j < n.



以样例 `[3, 10, 5, 25, 2, 8]` 为例，建完树之后可以得到下图：



左儿子为 1 的分支，右儿子为 0 的分支。

然后依次枚举每个数，在 Trie 树中找到与它异或结果最大的数。

这一步可以贪心来做：

从高位到低位，依次在 Trie 树中遍历，每次尽量走到与当前位不同的分支，这样可以使得找到的数与当前数在当前二进制位的异或结果是 1，从而可以得到尽量大的结果。

如上图所示，我们用 25 来举例说明，它的二进制表示是 (11001)：

```
最初指针在根节点(编号是 a 的点)，我们从 25 的二进制表示的最高位开始枚举；
  由于最高位是 1，我们走到 0 分支，走到 b 点；
  次高位是 1，我们继续往右儿子走，走到 c 点；
  下一位是 0，我们往左走，走到 d 点；
  下一位是 0，我们希望往左走，但发现左儿子不存在，所以只能往右走，走到 e 点；
  最后一位是 1，我们希望往右走，但发现右儿子不存在，所以只能往左走，最终走到 5；
所以和 25 异或值最大的数是 5，25 ^ 5 = 28。
```

```
public class Trie {
    private class Node { // 这我自己写的乱代码，贴在这里很不相关，也需要先测试一下
        public int val;
        public boolean isExist;
        public Node [] next;
        public Node(boolean isExist) {
            this.isExist = isExist;
            next = new Node[2];
            val = 0;
        }
        public Node() { this(false); }
        public Node(int va) {
            this(true);
            this.val = va;
        }
    }
    private Node root;
    public Trie() { root = new Node(); }
    public void insert(int va) {
        Node cur = root;
        for (int i = 31; i >= 0; i--) {
            int tmp = (va >> i) & 1;
```

```
            if (cur.next[tmp] == null)
                cur.next[tmp] = new Node();
            cur = cur.next[tmp];
        }
        cur.isExist = true;
    }
    public int search(int va) {
        int max = 0;
        Node cur = root;
        for (int i = 31; i >= 0; i--) {
            int t = (va >> i) & 1;
            if (cur.next[t^1] != null) {
                max += (1 << i);
                cur = cur.next[t^1];
            } else cur = cur.next[t&1];
        }
        return max;
    }
}
```

13.5.1 另一种位操作法

- 学到了异或操作的一个重要性质：$a^b = c$, 则有 $a^c = b$，且 $b^c = a$;

我们还需要用上一个异或的特性，假设 a 和 b 产生了最终的答案 max，即 a ^ b = x，那么根据异或的特性，a ^ x = b。同理，a 和 b 的最高位（前 n 位）也有相同的性质。

先以最高位为例子，我们可以把所有的数字的最高位放到一个 HashSet 里面，然后使用 1 与 set 里面的所有数字进行异或，如果得出的结果仍然在 set 里面，那么最终结果的最高位必然为 1，否则为 0。也即，先假定结果为 1，然后与 set 中所有数字异或，假定 a 与 1 异或得到结果 b（a ^ 1 = b），而 b 仍然在 set 里面，那么说明 set 中有两个数字异或能得到 1（a ^ b = 1）。否则，set 中没有两个数字能够异或得到 1，那么最终结果的最高位为 1 的假设失败，说明最终结果的最高位为 0。以此类推可以得到第二位、第三位。。。的数字。

再做一下推广，我们将所有数字的前 N 位放到一个 HashSet 里面，然后使用之前 N-1 位得到的最大值前缀 prefix 与 set 里面的所有数字进行异或，如果得出的结果仍然在 set 中，那么第 N 位必然为 1，否则为 0。

举个例子，给定数组 [14, 11, 7, 2]，二进制表示分别为 [1110, 1011, 0111, 0010]。题目说了，数字最长不会超过 32 位，所以应从 i = 31 开始，但是所有数字中最多位 4 位数，简单起见，我直接从最高位 i=3 开始

```
[14,   11,   7,    2]
[1110, 1011, 0111, 0010]
1. i = 3, set = {1000, 0000} => max = 1000
2. i = 2, set = {1100, 1000, 0100, 0000} => max = 1100
3. i = 1, set = {1110, 1010, 0110, 0010} => max = 1100
4. i = 0, set = {1110, 1011, 0111, 0010} => max = 1100
```

```
public int findMaximumXOR(int[] nums) { // 这种解法没有用到上面的这个 trie 呀
    int n = nums.length;
    int mask = 0, max = 0;
    HashSet<Integer> s = new HashSet<>();
    for (int i = 31; i >= 0; --i) { // i == 31 时
        mask = mask | 1 << i;     // 为获取前 n 位的临时变量
        for (int va : nums)
            s.add(va & mask);     // 将所有数字的前 n 位放入 set 中
        int tmp = max | (1 << i); // 假定第 n 位为 1, 前 n-1 位 max 为之前迭代求得
        for (Integer va : s)
            if (s.contains(va ^ tmp)) { // 查看 `b`是否在 // i == 31, (va^tmp):  -2147483648
                max = tmp;              // b 存在, 第 n 位为 1
                break;
            }
        s.clear();
    }
    return max;
```

```
}
// 此解法时间复杂度为 O(32n)=O(n)，空间复杂度上，我们使用了一个 HashSet 用于存储所有数字，因此空间复杂度是 O(n)
```

13.6 472. Concatenated Words - Hard

Given an array of strings words (without duplicates), return all the concatenated words in the given list of words.

A concatenated word is defined as a string that is comprised entirely of at least two shorter words in the given array.

Example 1:

```
Input: words = ["cat","cats","catsdogcats","dog","dogcatsdog","hippopotamuses","rat","ratcatdogcat"]
Output: ["catsdogcats","dogcatsdog","ratcatdogcat"]
Explanation: "catsdogcats" can be concatenated by "cats", "dog" and "cats";
"dogcatsdog" can be concatenated by "dog", "cats" and "dog";
"ratcatdogcat" can be concatenated by "rat", "cat", "dog" and "cat".
```

- 切记：dfs 深搜 + 记忆

```
// 切记： dfs 深搜 + 记忆 // Trie with memo, Time: o(m*2^n)
public class Trie {
    boolean isWord;
    Trie [] children;
    public Trie() {
        isWord = false;
        children = new Trie[26];
    }
}
public void insert(String word) {
    Trie cur = root;
    for (int i = 0; i < word.length(); i++) {
        char c = word.charAt(i);
        if (cur.children[c-'a'] == null)
            cur.children[c-'a'] = new Trie();
        cur = cur.children[c-'a'];
    }
    cur.isWord = true;
}
public boolean isConcatenated(String word, int idx, int cnt, HashMap<Integer, Boolean> memo) {
    if (memo.containsKey(idx)) return memo.get(idx);
    if (idx == word.length()) {
        memo.put(idx, cnt > 1);
        return cnt > 1;
    }
    Trie cur = root;
    for (int i = idx; i < word.length(); i++) {
        char c = word.charAt(i);
        if (cur.children[c-'a'] == null) {
            memo.put(idx, false);
            return false;
        } else {
            cur = cur.children[c-'a'];
            if (cur.isWord && isConcatenated(word, i+1, cnt+1, memo)) {
                memo.put(idx, true);
                return true;
            }
        }
    }
    memo.put(idx, false);
    return false;
}
Trie root = new Trie();
public List<String> findAllConcatenatedWordsInADict(String[] words) {
    for (String word : words)
        insert(word);
    List<String> res = new ArrayList<>();
    for (String word : words)
        if (isConcatenated(word, 0, 0, new HashMap<Integer, Boolean>()))
```

```
            res.add(word);
    return res;
}
```

- 一种稍微优化了一下的方法，逻辑就相对复杂一点儿，参考一下

```
public class Trie { // Trie with memo, Time: o(m*2^n)
    boolean isKey;
    Trie [] child;
    public Trie() {
        this.isKey = false;
        child = new Trie[26];
    }
    public void insert(String s) {
        int [] memo = new int [s.length()];
        Trie p = this;
        char [] sArr = s.toCharArray();
        boolean added = false;
        for (int i = 0; i < sArr.length; i++) {
            char c = sArr[i];
            if (p.child[c-'a'] == null)
                p.child[c-'a'] = new Trie();
            p = p.child[c-'a'];
            if (p.isKey && isConcatenated(s, i+1, 0, memo) && !added) {
                res.add(s);
                added = true;
            }
        }
        p.isKey = true;
    }     // 这么看来，我还没能透彻理解 dfs 深搜中的重复，什么时候应该拥有记忆?!!!
    public boolean isConcatenated(String s, int start, int cnt, int [] memo) {
        if (start == s.length() && cnt > 0) return true;
        if (memo[start] != 0) return memo[start] == 1;
        Trie p = this;
        char [] sArr = s.toCharArray();
        for (int i = start; i < sArr.length; i++) {
            char c = sArr[i];
            Trie cur = p.child[c-'a'];
            if (cur == null) {
                memo[start] = -1;
                return false;
            } else {
                if (cur.isKey && isConcatenated(s, i+1, cnt+1, memo)) {
                    memo[start] = 1;
                    return true;
                }
                p = cur;
            }
        }
        memo[start] = -1;
        return false;
    }
}
// Sort the words based on length
// Use trie to store words: while adding, checking if it is concatenated
// While checking, use dfs + memo
List<String> res = new ArrayList<>();
public List<String> findAllConcatenatedWordsInADict(String[] words) {
    Arrays.sort(words, (x, y) -> Integer.compare(x.length(), y.length()));
    Trie tree = new Trie();
    for (String word : words)
        tree.insert(word);
    return res;
}
```

13.7 1948. Delete Duplicate Folders in System - Hard

Due to a bug, there are many duplicate folders in a file system. You are given a 2D array paths, where paths[i] is an array representing an absolute path to the ith folder in the file system.

For example, ["one", "two", "three"] represents the path "/one/two/three". Two folders (not necessarily on the same level) are identical if they contain the same non-empty set of identical subfolders and underlying subfolder structure. The folders do not need to be at the root level to be identical. If two or more folders are identical, then mark the folders as well as all their subfolders.

For example, folders "/a" and "/b" in the file structure below are identical. They (as well as their subfolders) should all be marked:

```
/a
/a/x
/a/x/y
/a/z
/b
/b/x
/b/x/y
/b/z
```

However, if the file structure also included the path "/b/w", then the folders "/a" and "/b" would not be identical. Note that "/a/x" and "/b/x" would still be considered identical even with the added folder.

Once all the identical folders and their subfolders have been marked, the file system will delete all of them. The file system only runs the deletion once, so any folders that become identical after the initial deletion are not deleted.

Return the 2D array ans containing the paths of the remaining folders after deleting all the marked folders. The paths may be returned in any order.

```
public class Node {
    String name;
    Map<String, Node> children = new HashMap<>();
    private String hashCode = null;
    public Node (String name) {
        this.name = name;
    }
    public void add(List<String> path) {
        Node cur = this;
        for (String file : path) {
            if (!cur.children.containsKey(file))
                cur.children.put(file, new Node(file));
            cur = cur.children.get(file);
        }
    }
    public String getHashCode() {
        if (hashCode == null)
            hashCode = compueteHash();
        return hashCode;
    }
    private String compueteHash() {
        StringBuilder sb = new StringBuilder();
        List<Node> nodes = new ArrayList<>();
        for (Node n : children.values())
            nodes.add(n);
        if (nodes.size() == 0) return null;
        nodes.sort((a, b) -> a.name.compareTo(b.name));
        for (Node n : nodes) {
            sb.append('(');
            sb.append(n.name + n.getHashCode());
            sb.append(')');
        }
        return sb.toString();
    }
}
private void getGoodFiles(Node node, Map<String, Integer> occurs, List<String> cur, List<List<String>> ans) {
    if (occurs.containsKey(node.getHashCode()) && occurs.get(node.getHashCode()) > 1) return;
    cur.add(node.name);
    ans.add(new ArrayList<>(cur));
    for (Node n : node.children.values())
```

```
        getGoodFiles(n, occurs, cur, ans);
    cur.remove(cur.size()-1);
}
private void findOccurs(Node node, Map<String, Integer> occurs) {
    String key = node.getHashCode();
    if (key != null)
        occurs.put(key, occurs.getOrDefault(node.getHashCode(), 0) + 1);
    for (Node n : node.children.values())
        findOccurs(n, occurs);
}
Node root;
public List<List<String>> deleteDuplicateFolder(List<List<String>> paths) {
    root = new Node("");
    for (List<String> path : paths)
        root.add(path);
    Map<String, Integer> occurs = new HashMap<>();
    findOccurs(root, occurs);
    List<List<String>> ans = new ArrayList<>();
    for (Node n : root.children.values())
        getGoodFiles(n, occurs, new ArrayList<>(), ans);
    return ans;
}
```

13.8 792. Number of Matching Subsequences - Medium

Given a string s and an array of strings words, return the number of words[i] that is a subsequence of s.

A subsequence of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

For example, "ace" is a subsequence of "abcde".

```
// 我们需要使用每个字典中的单词去和 S 比较，看它是否是 S 的子序列。不过这种比较非常耗费时间，因此我们需要对 S 进行一下预处理。
// 首先定义一个二维数组 arr[][]，其中 arr[i][j] 代表距离 S 中第 i 位字符最近的 j 字符的位置。
// 换句话说，我们需要遍历一边字符串，记录下字符串 S 每一位上的字符，在它右侧距离它最近的 a-z 分别在哪。
public int numMatchingSubseq(String s, String[] words) {
    int n = s.length();
    int [][] arr = new int [n][26]; // 预处理用的数组
    for (int i = n-2; i >= 0; i--) {// 预处理
        arr[i] = Arrays.copyOf(arr[i+1], 26);
        arr[i][s.charAt(i+1)-'a'] = i+1;
    }
    int res = 0, idxAtS = 0, idx = 0, cur = 0;
    for (String v : words) {        // 比较每一个单词
        idxAtS = 0;                 // 对应 S 的下标
        idx = 0;                    // 当前单词下标
        if (v.charAt(0) == s.charAt(0)) { // 如果当前单词首字符等于 S 首字符
            idx ++;                 // 当前单词下标加一
            if (v.length() == 1) res++;      // 如果当前单词长度只有 1，说明当前单词已经遍历结束，结果加一
        }
        while (idx < v.length()) {            // 继续比较单词接下来的字符，在 S 中是否存在
            cur = v.charAt(idx) - 'a';
            if (arr[idxAtS][cur] == 0) break; // 如果 indexAtS 之后不存在 c，当前单词不合法
            idxAtS = arr[idxAtS][cur]; // 将 indexAtS 更新为 c 在 S 中的位置
            if (++idx == v.length()) res++;     // index 加一，如果 index 为单词最后一位，代表单词中所有字符均在 S 中找到
        }
    }
    return res;
}
```

14 binary Search

14.1 LeetCode Binary Search Summary 二分搜索法小结

- https://www.cnblogs.com/grandyang/p/6854825.html

二分查找法作为一种常见的查找方法，将原本是线性时间提升到了对数时间范围，大大缩短了搜索时间，具有很大的应用场景，而在 LeetCode 中，要运用二分搜索法来解的题目也有很多，但是实际上二分查找法的查找目标有很多种，而且在细节写法也有一些变化。之前有网友留言希望博主能针对二分查找法的具体写法做个总结，博主由于之前一直很忙，一直拖着没写，为了树立博主言出必行的正面形象，不能再无限制的拖下去了，那么今天就来做个了断吧，总结写起来 ~ (以下内容均为博主自己的总结，并不权威，权当参考，欢迎各位大神们留言讨论指正)

根据查找的目标不同，博主将二分查找法主要分为以下五类：

14.1.1 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 初始化为了 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

14.1.2 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}

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

14.1.3 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} 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)

14.1.4 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

14.1.5 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

14.2 793. 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 * ... * 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) = 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);
}
```

15 Tree: 只是用自己相对愚钝的脑袋把题目 AC 了，还需要一再总结套路，真正掌握关于树的绝大部分题型

15.1 979. Distribute Coins in Binary Tree

You are given the root of a binary tree with n nodes where each node in the tree has node.val coins. There are n coins in total throughout the whole tree.

In one move, we may choose two adjacent nodes and move one coin from one node to another. A move may be from parent to child, or from child to parent.

Return the minimum number of moves required to make every node have exactly one coin.

```java
private int dfs(TreeNode r) { // 统计把自身，左右子树都平衡，需要移动的 coins 个数
    if (r == null) return 0;
    int left = dfs(r.left);      // 左、右子树缺多少
    int right = dfs(r.right);
    res += Math.abs(left) + Math.abs(right); // 左，右子树和自身都平衡需要的移动数
    return left + right + r.val-1;
}
int res;
public int distributeCoins(TreeNode root) {
    res = 0;
    return res;
}
```

15.2 687. Longest Univalue Path

Given the root of a binary tree, return the length of the longest path, where each node in the path has the same value. This path may or may not pass through the root.

The length of the path between two nodes is represented by the number of edges between them.

- 此题与求二叉树的最长路径边长相似，只是此题要求是节点值相同的路径，也就是说在找最长路径的时候，还需要判断节点值，要是不相同，就重置为 0，在此期间，我们使用一个全局变量来存储最长节点值相同路径的边长。

```java
private int topDownTraverse(TreeNode r) {
    if (r == null) return 0;
    int left = topDownTraverse(r.left);
    int right = topDownTraverse(r.right);
    if (r.left == null || r.left.val != r.val) left = 0;
    if (r.right == null || r.right.val != r.val) right = 0;
    max = Math.max(max, left + right);
    return Math.max(left, right) + 1;
}
int max = 0;
public int longestUnivaluePath(TreeNode root) {
    if (root == null) return 0;
    topDownTraverse(root);
    return max;
}
```

15.3 652. Find Duplicate Subtrees

Given the root of a binary tree, return all duplicate subtrees.

For each kind of duplicate subtrees, you only need to return the root node of any one of them.

Two trees are duplicate if they have the same structure with the same node values.

```java
private String duplicate(TreeNode node) {
    if(node == null) return "X";
    String l = duplicate(node.left);
    String r = duplicate(node.right);
    String s = Integer.toString(node.val) + "-" + l + "-" + r;
    map.put(s, map.getOrDefault(s, 0)+1);
    if (map.get(s) == 2)
        list.add(node);
    return s;
}
HashMap<String,Integer> map = new HashMap<>();
ArrayList list = new ArrayList<>();
public List findDuplicateSubtrees(TreeNode root) {
    duplicate(root);
    return list;
}
```

- 看一下构造的图的效果图

```
      1 -> root
    2, 3,  ->
4, #| 2, 4,  ->
#.#| 4, #| #.#|  ->
#.#|  ->

map.size(): 4
3-2-4-X-X-X-4-X-X, 1
1-2-4-X-X-X-3-2-4-X-X-X-4-X-X, 1
2-4-X-X-X, 2
4-X-X, 3

res.size(): 2
TREE Level order traversal:
      4 -> root
    #.#|  ->

TREE Level order traversal:
      2 -> root
    4, #|  ->
#.#|  ->
```

- 一种 dfs 的写法

```
HashSet<String> set, added;
List<TreeNode> list;
public List<TreeNode> findDuplicateSubtrees(TreeNode root) {
    set = new HashSet();
    added = new HashSet();
    list = new ArrayList();
    StringBuilder ret = dfs(root);
    return list;
}
private StringBuilder dfs(TreeNode root){
    if (root == null) return null;
    StringBuilder sbL = dfs(root.left), sbR = dfs(root.right);
    if (sbL == null && sbR == null){
        sbL = new StringBuilder();
        sbL.append(root.val);
    } else if (sbL != null){
        sbL.append(" " + root.val);
        if (sbR != null){
            sbL.append(' ');
            sbL.append(sbR);
        } else sbL.append(" n");
    } else if (sbL == null){
        if (sbR != null){
            sbR.insert(0, " n " + root.val);
            sbL = sbR;
        }
    }
    String temp = sbL.toString();
    if (set.contains(temp) && !added.contains(temp)){
        list.add(root);
        added.add(temp);

    }
    set.add(temp);
    return sbL;
}
```

- 这个跑起来很高效，可惜我看不懂。。。。。以后再慢慢消化吧
- https://leetcode.com/problems/find-duplicate-subtrees/discuss/1418487/Java-beats-
  5-in-time

```
Map<Integer, Integer> count;           // frequency of each subtree represented in string
Map<List<Integer>, Integer> numberMap; // ** not hashset since it cannot reserve element order
List<TreeNode> ans;
int globalNumber = 1;
public List<TreeNode> findDuplicateSubtrees(TreeNode root) {
    count = new HashMap();
    numberMap = new HashMap();
    ans = new ArrayList();
    collect(root);
    return ans;
}
public int collect(TreeNode node) {
    if (node == null) return 0;
    int leftNumber = collect(node.left);
    int rightNumber = collect(node.right);
    List<Integer> numberExp = new ArrayList<>(); // construct expression
    numberExp.add(node.val);
    numberExp.add(leftNumber);
    numberExp.add(rightNumber);
    if (!numberMap.containsKey(numberExp)) { // update numberMap
        numberMap.put(numberExp, globalNumber);
        globalNumber++;
    }
    // check number frequency. if == 2, meaning duplication then add to result
    int rootNumber = numberMap.get(numberExp).intValue();
    count.put(rootNumber, count.getOrDefault(rootNumber, 0)+1);
    if (count.get(rootNumber) == 2) // not >=2, otherwise ans will have duplicated nodes
        ans.add(node);
    return rootNumber;
}
```

```
count.size(): 4
1, 3
2, 2
3, 1
4, 1
numberMap.size(): 4
2, 1, 0,
2
3, 2, 1,
3
1, 2, 3,
4
4, 0, 0,
1
```

16 字符串相关

16.1 467. Unique Substrings in Wraparound String - Medium

We define the string s to be the infinite wraparound string of "abcdefghijklmnopqrstuvwxyz", so s will look like this:

"...zabcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyzabcd....". Given a string p, return the number of unique non-empty substrings of p are present in s.

这道题说有一个无限长的封装字符串，然后又给了我们另一个字符串 p，问我们 p 有多少非空子字符串在封装字符串中。我们通过观察题目中的例子可以发现，由于封装字符串是 26 个字符按顺序无限循环组成的，那么满足题意的 p 的子字符串要么是单一的字符，要么是按字母顺序的子字符串。这道题遍历 p 的所有子字符串会 TLE，因为如果 p 很大的话，子字符串很多，会有大量的满足题意的重复子字符串，必须要用到 trick，而所谓技巧就是一般来说你想不到的方法。我们看 abcd 这个字符串，以 d 结尾的子字符串有 abcd, bcd, cd, d，那么我们可以发现 bcd 或者 cd 这些以 d 结尾的字符串的子字符串都包含在 abcd 中，那么我们知道以某个字符结束的最大字符串包含其他以该字符结束的字符串的所有子字符串，说起来很拗口，但是理解了我上面举的例子就行。那么题目

就可以转换为分别求出以每个字符 (a-z) 为结束字符的最长连续字符串就行了，我们用一个数组 cnt 记录下来，最后在求出数组 cnt 的所有数字之和就是我们要的结果啦，

```
public int findSubstringInWraproundString(String p) {
    int n = p.length();
    int [] arr = new int [n];
    int [] cnt = new int [26];
    for (int i = 0; i < n; i++)
        arr[i] = p.charAt(i) - 'a';
    int res = 0, maxLen = 0;
    for (int i = 0; i < n; i++) {
        if (i > 0 && (arr[i-1]+1) % 26 == arr[i]) // 判断前一个位置上的字符比现位字符小 1
            ++maxLen;
        else maxLen = 1;
        cnt[arr[i]] = Math.max(cnt[arr[i]], maxLen);
    }
    for (int i = 0; i < 26; i++)
        res += cnt[i];
    return res;
}
```

17 others

17.1 Predict the Winner

You are given an integer array nums. Two players are playing a game with this array: player 1 and player 2. Player 1 and player 2 take turns, with player 1 starting first. Both players start the game with a score of 0. At each turn, the player takes one of the numbers from either end of the array (i.e., nums[3] or nums[nums.length - 1]) which reduces the size of the array by 1. The player adds the chosen number to their score. The game ends when there are no more elements in the array. Return true if Player 1 can win the game. If the scores of both players are equal, then player 1 is still the winner, and you should also return true. You may assume that both players are playing optimally.

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

17.2 Rectangle Area II

We are given a list of (axis-aligned) rectangles. Each rectangle[i] = [xi1, yi1, xi2, yi2] , where (xi1, yi1) are the coordinates of the bottom-left corner, and (xi2, yi2) are the coordinates of the top-right corner of the ith rectangle. Find the total area covered by all rectangles in the plane. Since the answer may be too large, return it modulo 109 + 7.

```
private void addRectange(List<int[]> l, int [] arr, int idx) {
    if (idx >= l.size()) {
        l.add(arr);
        return;
    }
    int [] r = l.get(idx);
    // 新增矩形 处在 左 右 上 下 四侧完全不重叠的区域
    if (arr[2] <= r[0] || arr[0] >= r[2] || arr[1] >= r[3] || arr[3] <= r[1]) {
        addRectange(l, arr, idx+1);
```

```
        return;
    }
    if (arr[0] < r[0]) // 新增矩形 左侧 有交叠
        addRectange(l, new int[]{arr[0], arr[1], r[0], arr[3]}, idx+1);
    if (arr[2] > r[2]) // 新增矩形 右侧 也可以有 交叠
        addRectange(l, new int[]{r[2], arr[1], arr[2], arr[3]}, idx+1);
    if (arr[1] < r[1]) // 新增矩形 下侧 也可以有 交叠：注意左侧、右侧前面已经加进去了，现在只加中间部分，不要重复计算
        addRectange(l, new int[]{Math.max(arr[0], r[0]), arr[1], Math.min(arr[2], r[2]), r[1]}, idx+1);
    if (arr[3] > r[3]) // 新增矩形 上侧 也可以有 交叠：注意左侧、右侧、下侧 前面已经加进去了，现在只加中间部分，不要重复计算
        addRectange(l, new int[]{Math.max(arr[0], r[0]), r[3], Math.min(arr[2], r[2]), arr[3]}, idx+1);
}
public int rectangleArea(int[][] rectangles) {
    int mod = (int)Math.pow(10, 9)+7;
    long res = 0;
    List<int[]> rlist = new ArrayList<>();
    for (int [] val : rectangles)
        addRectange(rlist, val, 0);
    for (int [] arr : rlist)
        res = (res + ((long)(arr[2]-arr[0])*(long)(arr[3]-arr[1]))) % mod;
    return (int)res % mod;
}
```

17.3 Construct Binary Tree from Preorder and Postorder Traversal

Given two integer arrays, preorder and postorder where preorder is the preorder traversal of a binary tree of distinct values and postorder is the postorder traversal of the same tree, reconstruct and return the binary tree. If there exist multiple answers, you can return any of them.

```
public TreeNode constructFromPrePost(int[] preorder, int[] postorder) {
    int n = preorder.length;
    TreeNode r = new TreeNode(preorder[0]);
    if (n == 1) return r;
    Stack<TreeNode> s = new Stack<>();
    s.push(r);
    int idx = 0;
    for (int i = 1; i < n; i++) {
        TreeNode cur = new TreeNode(preorder[i]);
        if (s.peek().left == null) s.peek().left = cur;
        else s.peek().right = cur;
        s.push(cur);
        while (idx < n && postorder[idx] == s.peek().val) {
            s.pop();
            ++idx;
        }
    }
    return r;
}
```

17.4 Path Sum III

Given the root of a binary tree and an integer targetSum, return the number of paths where the sum of the values along the path equals targetSum. The path does not need to start or end at the root or a leaf, but it must go downwards (i.e., traveling only from parent nodes to child nodes).

```
private int solve(TreeNode r, int t, int value) {
    if (r == null) return 0;
    if (value + r.val == t)
        return 1 + solve(r.left, 0, 0) + solve(r.right, 0, 0);
    return solve(r.left, t, value + r.val) + solve(r.right, t, value + r.val);
}
public int pathSum(TreeNode root, int targetSum) {
    if (root == null) return 0;
    return solve(root, targetSum, 0) + pathSum(root.left, targetSum) + pathSum(root.right, targetSum);
}
```

17.5 Critical Connections in a Network

- There are n servers numbered from 0 to n - 1 connected by undirected server-to-server connections forming a network where connections[i] = [ai, bi] represents a connection between servers ai and bi. Any server can reach other servers directly or indirectly through the network.
- A critical connection is a connection that, if removed, will make some servers unable to reach some other server.
- Return all critical connections in the network in any order.

```
static class Eg {
    int u, v, next;
    // int w;
    boolean cut;
    // int num;
}
public Eg[] egs;
public int cnt;
public int [] fir; // 边的出发点
int [] low;
int [] dfn;
int recdfn;
void tarjanAddEg(int u, int v, int w) {
    egs[cnt] = new Eg();
    egs[cnt].u = u;
    egs[cnt].v = v;
    // egs[cnt].w = w;
    egs[cnt].cut = false;
    // egs[cnt].num = 0;
    egs[cnt].next = fir[u]; // ?
    fir[u] = cnt++;         // ?
}
private void initTarjan(int nodeSize, int edgeSize) {
    cnt = 0;
    egs = new Eg [edgeSize];
    low = new int [nodeSize];
    dfn = new int [nodeSize];
    fir = new int [edgeSize];
    Arrays.fill(fir, -1);
}
private void tarjan(int u, int fa) { // fa: father
    low[u] = ++recdfn;
    dfn[u] = recdfn;
    int have = 0;
    for (int i = fir[u]; i != -1; i = egs[i].next) {
        int v = egs[i].v;
        if (have == 0 && v == fa) { // 走过你来时的路
            have++;
            continue;
        }
        if (dfn[v] == 0) { // dfs 过程中还未经过该点
            tarjan(v, u);
            low[u] = Math.min(low[u], low[v]);
            if (dfn[u] < low[v]) { // 连通世外桃源与外界的路
                // 当 dfn[x] < low[y] 的时候:
                // --- 我们发现从 yy 节点出发, 在不经过 (x,y)(x,y) 的前提下, 不管走哪一条边, 我们都无法抵达 xx 节点, 或者比 xx 节
                // --- 此时我们发现 yy 所在的子树似乎形成了一个封闭圈, 那么 (x,y)(x,y) 自然也就是桥了.
                egs[i].cut = true;
                egs[i^1].cut = true; // ???
            }
        } else {
            low[u] = Math.min(low[u], dfn[v]); // 取已访问的节点的 dfs 序的最小值
        }
    }
}
private boolean findEdgeCut(int l, int r) {
    Arrays.fill(low, 0);
    Arrays.fill(dfn, 0);
    recdfn = 0;
```

```
    tarjan(l, l);
    for (int i = l; i <= r; i++) {
        if (dfn[i] == 0) return false;
    }
    return true;
}
public List<List<Integer>> criticalConnections(int n, List<List<Integer>> connections) {
    initTarjan(n, connections.size()*2);
    for (List<Integer> eg : connections) {
        tarjanAddEg(eg.get(0), eg.get(1), 1);
        tarjanAddEg(eg.get(1), eg.get(0), 1);
    }
    // boolean ans = findEdgeCut(0, n-1);
    Arrays.fill(low, 0);
    Arrays.fill(dfn, 0);
    recdfn = 0;
    tarjan(0, 0);
    List<List<Integer>> res = new ArrayList<>();
    int l = connections.size();
    for (int i = 0; i < l * 2; i += 2) { // i += 2 skipped egs[i^1] ?
        Eg eg = egs[i];
        if (eg != null && eg.cut) {
            List<Integer> t = new ArrayList<>();
            t.add(eg.u);
            t.add(eg.v);
            res.add(t);
        }
    }
    return res;
}
```