

LeetCode Online Coding Interview Questions – 错题集 v2 精减版图形  
算法基本概念——旁听 Graph Theory/Algorithm 课，概念消化  
LeetCode Online Coding Interview Questions – 错题集

deepwaterooo

October 7, 2024



# Contents

<b>1 Dynamic Programming, 动态规划</b>	7
1.1 第二次仍不会的题号记这里 . . . . .	7
1.1.1 518. Coin Change 2 . . . . .	7
1.1.2 464. Can I Win 这个题：为什么顺序无关了？ . . . . .	7
1.1.3 494. Target Sum - Medium . . . . .	8
1.1.4 801. Minimum Swaps To Make Sequences Increasing - Hard . . . . .	11
1.1.5 1639. Number of Ways to Form a Target String Given a Dictionary - Hard . . . . .	11
1.2 区间型 DP . . . . .	12
1.2.1 1039. Minimum Score Triangulation of Polygon - Medium . . . . .	13
1.2.2 312. Burst Balloons 区间型动态规划的典型代表 . . . . .	13
1.2.3 546. Remove Boxes - Hard: 带隐含信息，需要第三维参数加入的 . . . . .	13
1.2.4 664. Strange Printer - Hard . . . . .	14
1.2.5 1591. Strange Printer II - Hard todo . . . . .	14
1.3 扫描线类、时间戳、一维线性 DP/ 单序列/ 接龙型 . . . . .	15
1.3.1 805. Split Array With Same Average 【活宝妹就是一定要嫁给亲爱的表哥!!!】 . . . . .	15
1.3.2 805. Split Array With Same Average 【这里是怎麼又重复了一遍？】需要合并一下 . . . . .	17
1.3.3 2008. Maximum Earnings From Taxi - Medium . . . . .	18
1.3.4 1713. Minimum Operations to Make a Subsequence - Hard LIS 经典题型，需要吃透 . . . . .	18
1.3.5 1883. Minimum Skips to Arrive at Meeting On Time - Hard . . . . .	19
1.3.6 1911. Maximum Alternating Subsequence Sum - Medium todo: 还需要总结题解 . . . . .	20
1.3.7 1928. Minimum Cost to Reach Destination in Time - Hard . . . . .	20
1.3.8 730. Count Different Palindromic Subsequences - Hard . . . . .	21
1.3.9 1125. Smallest Sufficient Team - Hard 这个题要多写几遍 . . . . .	23
1.3.10 1575. Count All Possible Routes - Hard . . . . .	24
1.3.11 1012. Numbers With Repeated Digits - Hard 数位 DP + 压缩状态经典 . . . . .	25
1.3.12 600. Non-negative Integers without Consecutive Ones - Hard . . . . .	27
1.3.13 514. Freedom Trail - Hard . . . . .	28
1.3.14 78. Subsets - Medium 典型 subset 题目 new ArrayList<>(){ add(v); } . . . . .	30
1.3.15 368. Largest Divisible Subset - Medium 要求返回序列的接龙型 . . . . .	30
1.3.16 673. Number of Longest Increasing Subsequence . . . . .	32
1.3.17 1987. Number of Unique Good Subsequences - Hard . . . . .	32
1.3.18 847. Shortest Path Visiting All Nodes . . . . .	34
1.3.19 926. Flip String to Monotone Increasing - Medium . . . . .	35
1.3.20 313. Super Ugly Number . . . . .	36
1.3.21 2879. Profitable Schemes - Hard 0-1 背包问题 . . . . .	36
1.3.22 3377. Combination Sum IV 没能认出这个题目是考 DP . . . . .	37
1.3.23 409. Last Stone Weight II . . . . .	38
1.3.24 1049. Chalkboard XOR Game - Hard . . . . .	39
1.3.25 1449. Form Largest Integer With Digits That Add up to Target . . . . .	41
1.3.27 516. Longest Palindromic Subsequence . . . . .	41
1.3.28 1143. Longest Common Subsequence . . . . .	41
1.3.29 1092. Shortest Common Supersequence - Hard . . . . .	42
1.3.30 546. Remove Boxes - Hard . . . . .	43
1.3.31 11531. String Compression II - Hard . . . . .	43
1.3.32 1039. Minimum Score Triangulation of Polygon . . . . .	44
1.3.33 1478. Allocate Mailboxes - Hard . . . . .	44
1.3.34 1771. Maximize Palindrome Length From Subsequences . . . . .	46

1.3.351896. Minimum Cost to Change the Final Value of Expression - Hard	46
1.3.36823. Binary Trees With Factors	47
1.3.37363. Max Sum of Rectangle No Larger Than K	47
1.3.381884. Egg Drop With 2 Eggs and N Floors - Medium	48
1.3.39887. Super Egg Drop - Hard	49
1.3.401981. Minimize the Difference Between Target and Chosen Elements	51
1.3.41801. Minimum Swaps To Make Sequences Increasing - Hard	52
1.3.42837. New 21 Game - Medium	52
1.3.431105. Filling Bookcase Shelves - Medium	53
1.3.44790. Domino and Tromino Tiling	54
1.3.451997. First Day Where You Have Been in All the Rooms - Medium	55
1.3.46943. Find the Shortest Superstring - Hard	55
1.3.47964. Least Operators to Express Number - Hard	57
1.3.481955. Count Number of Special Subsequences - Hard 把这些个递推公式记住	58
1.3.49446. Arithmetic Slices II - Subsequence - Hard	59
1.3.50639. Decode Ways II - Hard	61
1.3.51629. K Inverse Pairs Array - Hard	62
1.3.521787. Make the XOR of All Segments Equal to Zero - Hard	63
1.3.531735. Count Ways to Make Array With Product - Hard 乘积为 K 的质因子数排列组合的总个数：分解质因数	65
1.3.541359. Count All Valid Pickup and Delivery Options - Hard	67
1.3.551187. Make Array Strictly Increasing - Hard 需要重写	67
1.4 多维数个数、数种类数多一维 k 介入的 dp[i][j][k]	68
1.4.1 920. Number of Music Playlists - Hard	68
1.4.2 1866. Number of Ways to Rearrange Sticks With K Sticks Visible - Hard	69
1.4.3 1916. Count Ways to Build Rooms in an Ant Colony - Hard	69
1.4.4 1504. Count Submatrices With All Ones - Medium	72
1.5 BitMask 掩码相关的【这是第三次不会的】	73
1.5.1 1595. Minimum Cost to Connect Two Groups of Points	73
1.6 【第三次】不会的	74
1.6.1 1187. Make Array Strictly Increasing: 【动规】思路独特	74
1.6.2 416. Partition Equal Subset Sum 同硬币和问题，感觉总容易忘记，有点儿提示才能回想起来	76
<b>2 dfs 记忆化搜索</b>	<b>79</b>
2.0.1 1977. Number of Ways to Separate Numbers - Hard	79
2.0.2 87. Scramble String - Hard 非常经典：要韧熟于心	82
2.0.3 2065. Maximum Path Quality of a Graph - Hard 记忆化搜索 + 重复遍历	85
2.0.4 1397. Find All Good Strings - Hard 记忆化搜索	86
2.0.5 2060. Check if an Original String Exists Given Two Encoded Strings - Hard dfs 记忆化搜索	87
2.0.6 638. Shopping Offers - Medium 记忆化搜索 or 背包动态规划	88
2.0.7 474. Ones and Zeroes - Medium	89
2.0.8 1900. The Earliest and Latest Rounds Where Players Compete - Hard	90
2.0.9 488. Zuma Game - Hard	93
<b>3 Segment Tree 与 Binary Index Tree 线段树与树状数组</b>	<b>97</b>
3.0.1 1157. Online Majority Element In Subarray - Hard	97
3.0.2 1825. Finding MK Average - Hard	99
3.0.3 315. Count of Smaller Numbers After Self - Hard	102
3.0.4 327. Count of Range Sum - Hard 重点这几个题要好好再理解消化几遍	103
3.0.5 699. Falling Squares - Hard	106
3.0.6 1483. Kth Ancestor of a Tree Node - Hard 倍增法 binary lifting	109
3.0.7 236 二叉树的最近公共祖先	110
3.0.8 1505. Minimum Possible Integer After at Most K Adjacent Swaps On Digits - Hard BIT 树状数组	110
<b>4 字符串</b>	<b>113</b>
4.1 总结：几种【字符串算法】关键字	113
4.2 KMP 算法	113
4.2.1 1392. Longest Happy Prefix - Hard : Use Longest Prefix Suffix (KMP-table) or String Hashing.	113
4.2.2 214. Shortest Palindrome - Hard KMP 算法	114
4.2.3 1910. Remove All Occurrences of a Substring - Medium 可用 KMP 算法	115

4.2.4 1316. Distinct Echo Substrings - Hard . . . . .	115
4.2.5 467. Unique Substrings in Wraparound String - Medium Rabin-Karp Rolling Hash 算法 . . . . .	118
4.2.6 1044. Longest Duplicate Substring - Hard . . . . .	118
4.2.7 1156. Swap For Longest Repeated Character Substring - Medium . . . . .	119
4.2.8 395. Longest Substring with At Least K Repeating Characters - Medium . . . . .	121
4.2.9 1830. Minimum Number of Operations to Make String Sorted - Hard 排列组合费小马快速幂 . . . . .	122
4.2.10 1960. Maximum Product of the Length of Two Palindromic Substrings - Hard 马拉车算法 todo . . . . .	123
4.2.11 854. K-Similar Strings - Hard . . . . .	123
4.2.12 Queue Array Deque LinkedList 效率比较 . . . . .	124
4.3 AC 自动机: . . . . .	124
4.3.1 3292. Minimum Number of Valid Strings to Form Target II . . . . .	124
<b>5 Bit Manipulations</b>	<b>127</b>
5.1 基本概念: 原码、反码、与补码 (对负数尤其重要) . . . . .	127
5.1.1 1、原码: . . . . .	127
5.1.2 2、反码 . . . . .	127
5.1.3 3、补码 . . . . .	127
5.2 数组中不重复的两个元素 . . . . .	127
5.3 371. Sum of Two Integers . . . . .	128
5.3.1 201. Bitwise AND of Numbers Range . . . . .	129
5.3.2 1835. Find XOR Sum of All Pairs Bitwise AND - Hard . . . . .	129
5.3.3 982. Triples with Bitwise AND Equal To Zero 平生不识 TwoSum, 刷尽 LeetCode 也枉然 . . . . .	130
5.3.4 187. Repeated DNA Sequences - Medium . . . . .	130
5.3.5 1915. Number of Wonderful Substrings - Medium . . . . .	131
5.3.6 782. Transform to Chessboard- Hard . . . . .	131
5.3.7 1803. Count Pairs With XOR in a Range - Hard . . . . .	133
5.3.8 1734. Decode XORed Permutation - Medium . . . . .	134
5.3.9 957. Prison Cells After N Days - Medium . . . . .	134
<b>6 Bit Masks</b>	<b>137</b>
6.0.1 总结一下 . . . . .	137
6.0.2 1655. Distribute Repeating Integers - Hard . . . . .	137
6.0.3 1659. Maximize Grid Happiness - Hard . . . . .	138
6.0.4 1755. Closest Subsequence Sum - Hard 分成两半: 由 O(2^N) 降为 O(NlogN) . . . . .	141
6.0.5 2035. Partition Array Into Two Arrays to Minimize Sum Difference - Hard 上一题: 分成两半的套娃题 todo: + binarySearch 等解法 . . . . .	142
6.0.6 956. Tallest Billboard: 折腰轨半第三次: 重要的题型重复三遍 . . . . .	143
6.0.7 1982. Find Array Given Subset Sums - Hard . . . . .	146
6.0.8 1815. Maximum Number of Groups Getting Fresh Donuts - Hard . . . . .	148
6.0.9 691. Stickers to Spell Word - Hard . . . . .	149
6.0.10 1994. The Number of Good Subsets . . . . .	151
6.0.11 1494. Parallel Courses II - Hard . . . . .	151
6.0.12 分析一下 . . . . .	153
6.0.13 1349. Maximum Students Taking Exam - Hard . . . . .	155
6.0.14 1434. Number of Ways to Wear Different Hats to Each Other - Hard . . . . .	156
6.0.15 1595. Minimum Cost to Connect Two Groups of Points - Hard 自顶向下 (dfs + 记忆数组); 自底向上: DP table . . . . .	158
6.0.16 自顶向下与自底向上 . . . . .	159
<b>7</b>	<b>161</b>
<b>8 Graph</b>	<b>163</b>
8.1 拓扑排序 . . . . .	163
8.1.1 1857. Largest Color Value in a Directed Graph - Hard . . . . .	163
8.1.2 1203. Sort Items by Groups Respecting Dependencies - Hard . . . . .	164
8.1.3 2045. Second Minimum Time to Reach Destination - Hard . . . . .	165
8.1.4 1334. Floyd 算法 - Find the City With the Smallest Number of Neighbors at a Threshold Distance - Medium . . . . .	166
8.1.5 1129. Shortest Path with Alternating Colors - Medium . . . . .	167
8.1.6 882. Reachable Nodes In Subdivided Graph - Hard . . . . .	168
8.1.7 1782. Count Pairs Of Nodes - Hard . . . . .	169

8.1.8 2115. Find All Possible Recipes from Given Supplies . . . . .	170
8.2 基环内向树 . . . . .	170
8.2.1 2127. Maximum Employees to Be Invited to a Meeting - Hard 基环内向树 . . . . .	170
8.3 Tarjan 算法 . . . . .	173
8.3.1 算法简介 . . . . .	174
8.3.2 1192. Critical Connections in a Network- Hard Tarjan 算法 Tarjan's algorithm Kosaraju 算法 – todo: 这个题不太懂 . . . . .	174
8.4 无向图 . . . . .	175
8.4.1 2508. Add Edges to Make Degrees of All Nodes Even . . . . .	175
8.4.2 2538. Difference Between Maximum and Minimum Price Sum . . . . .	176
8.5 欧拉回路: Hierholzer 算法, Fleury 算法 . . . . .	177
8.5.1 753. Cracking the Safe - Hard . . . . .	178
8.5.2 332. Reconstruct Itinerary - Medium 欧拉回路 Hierholzer 算法 . . . . .	180
8.5.3 2097. Valid Arrangement of Pairs - Hard 欧拉回路 . . . . .	182
8.5.4 1591. Strange Printer II - Hard . . . . .	183
8.6 双端队列 BFS . . . . .	185
8.6.1 1368. Minimum Cost to Make at Least One Valid Path in a Grid - Hard . . . . .	185
8.6.2 126. Word Ladder II - Hard BFS . . . . .	187
8.7 环与入度等 . . . . .	189
8.7.1 685. Redundant Connection II - Hard . . . . .	189
8.8 普通 BFS, 但 state 状态很重要 . . . . .	190
8.8.1 1293. Shortest Path in a Grid with Obstacles Elimination - Hard . . . . .	190
8.9 最小生成树 . . . . .	191
8.9.1 1489. Find Critical and Pseudo-Critical Edges in Minimum Spanning Tree - Hard . . . . .	191
8.10 跳表 Skiplist . . . . .	193
8.10.1 1206. Design Skiplist - Hard . . . . .	193
8.11 不知道归纳到哪里, 又很巧妙的题 . . . . .	195
8.11.1 2322. Minimum Score After Removals on a Tree - Hard . . . . .	195
8.11.2 2258. Escape the Spreading Fire: 多源头 BFS . . . . .	197
8.11.3 3311. Construct 2D Grid Matching Graph Layout . . . . .	198
<b>9 扫描线</b>	<b>201</b>
9.0.1 435. Non-overlapping Intervals - Medium 动态规划贪心 . . . . .	201
9.0.2 2054. Two Best Non-Overlapping Events - Medium . . . . .	202
9.0.3 2055. Plates Between Candles - Medium . . . . .	203
9.0.4 218. The Skyline Problem - Hard . . . . .	204
9.0.5 715. Range Module - Hard . . . . .	206
9.0.6 352. Data Stream as Disjoint Intervals - Hard . . . . .	206
9.0.7 1419. Minimum Number of Frogs Croaking - Medium . . . . .	207
<b>10 单调栈</b>	<b>211</b>
10.0.1 1975. Odd Even Jump - Hard . . . . .	211
10.0.2 1793. Maximum Score of a Good Subarray - Hard . . . . .	212
<b>11 字符串</b>	<b>215</b>
11.1 总结 . . . . .	215
11.2 KMP 算法 . . . . .	215
11.2.1 1392. Longest Happy Prefix - Hard : Use Longest Prefix Suffix (KMP-table) or String Hashing . . . . .	215
11.2.2 21910. Remove All Occurrences of a Substring - Medium 可用 KMP 算法 . . . . .	216
11.2.3 1316. Distinct Echo Substrings - Hard . . . . .	216
11.2.4 4467. Unique Substrings in Wraparound String - Medium Rabin-Karp Rolling Hash 算法 . . . . .	219
11.2.5 1044. Longest Duplicate Substring - Hard . . . . .	219
11.2.6 11156. Swap For Longest Repeated Character Substring - Medium . . . . .	220
11.2.7 395. Longest Substring with At Least K Repeating Characters - Medium . . . . .	222
11.2.8 1830. Minimum Number of Operations to Make String Sorted - Hard 排列组合费小马快速幂 . . . . .	223
11.2.9 1960. Maximum Product of the Length of Two Palindromic Substrings - Hard 马拉车算法 todo . . . . .	224

<b>12 PreSum 差分数组</b>	<b>227</b>
12.11674. Minimum Moves to Make Array Complementary - Medium 差分数组 . . . . .	227
12.2798. Smallest Rotation with Highest Score . . . . .	228
12.31074. Number of Submatrices That Sum to Target - Hard 二维数组转化为一维数组: 数组中 target Sum 的 2D 版、presum 思想的 2d 版 . . . . .	230
12.3.1Java O(NlgN) optimized Brute Force with Fenwick Tree O(NlogN) . . . . .	232
<b>13 Greedy</b>	<b>235</b>
13.11585. Check If String Is Transformable With Substring Sort Operations - Hard . . . . .	235
13.21330. Reverse Subarray To Maximize Array Value - Hard . . . . .	235
<b>14 others</b>	<b>237</b>
14.1 Predict the Winner . . . . .	237
14.2 Rectangle Area II . . . . .	237
14.3 Construct Binary Tree from Preorder and Postorder Traversal . . . . .	238
14.4 Path Sum III . . . . .	238
14.5 Critical Connections in a Network . . . . .	238
14.6891. Sum of Subsequence Widths - Hard 考 sorting 和对 subsequence 的理解 . . . . .	239
14.7335. Self Crossing . . . . .	240
14.8391. Perfect Rectangle - Hard . . . . .	241
<b>15 HashMap</b>	<b>245</b>
15.1 数有多少个数组数不清楚 . . . . .	245
15.21392. Longest Happy Prefix - Hard . . . . .	245
15.3336. Palindrome Pairs - Hard . . . . .	245
15.3.1 解题思路与分析 . . . . .	245
15.4 KMP 算法 . . . . .	246
15.51172. Dinner Plate Stacks - Hard . . . . .	248
15.62080. Range Frequency Queries - Medium . . . . .	249
15.6.1 解题思路与分析: 最简单粗暴的 . . . . .	250
15.6.2 解题思路与分析: binary Search . . . . .	250
15.6.3 解题思路与分析: segment tree . . . . .	250
<b>16 binary Search</b>	<b>253</b>
16.1 LeetCode Binary Search Summary 二分搜索法小结 . . . . .	253
16.1.1 标准二分查找 . . . . .	253
16.1.2 二分查找左边界 . . . . .	253
16.1.3 二分查找右边界 . . . . .	255
16.1.4 二分查找左右边界 . . . . .	255
16.1.5 二分查找极值 . . . . .	255
16.1.6 第一类: 需查找和目标值完全相等的数 . . . . .	257
16.1.7 第二类: 查找第一个不小于目标值的数, 可变形为查找最后一个小于目标值的数 . . . . .	257
16.1.8 第三类: 查找第一个大于目标值的数, 可变形为查找最后一个不大于目标值的数 . . . . .	258
16.1.9 第四类: 用子函数当作判断关系 (通常由 mid 计算得出) . . . . .	258
16.1.10 第五类: 其他 (通常 target 值不固定) . . . . .	258
16.2793. Preimage Size of Factorial Zeros Function . . . . .	259
16.32040. Kth Smallest Product of Two Sorted Arrays - Hard . . . . .	260
<b>17 Tree 树结构: 各种新型数据结构</b>	<b>263</b>
17.1 最大深度到树的最大直径 (不一定经过根节点) . . . . .	263
17.1.1104. Maximum Depth of Binary Tree . . . . .	263
17.1.2543. Diameter of Binary Tree . . . . .	263
17.1.32385. Amount of Time for Binary Tree to Be Infected . . . . .	264
17.1.4979. Distribute Coins in Binary Tree . . . . .	265
17.1.51719. Number Of Ways To Reconstruct A Tree - Hard . . . . .	265
17.1.61766. Tree of Coprimes - Hard . . . . .	267
17.1.71028. Recover a Tree From Preorder Traversal: 栈 + 迭代, 递归 - Hard . . . . .	268
17.1.81932. Merge BSTs to Create Single BST . . . . .	269
17.1.9687. Longest Univalue Path . . . . .	270
17.1.1052. Find Duplicate Subtrees . . . . .	270
17.1.1 Create Sorted Array through Instructions . . . . .	272

17.1.12696. Jump Game VI . . . . .	272
17.1.13345. Jump Game IV - Hard . . . . .	273
17.1.1468. Binary Tree Cameras . . . . .	274
17.1.1512. Path Sum . . . . .	274
<b>18 Segment Tree 与 Binary Index Tree 线段树与树状数组</b>	<b>277</b>
18.1求和 Sum 的线段树 . . . . .	277
18.1.1327. Count of Range Sum - Hard 重点这几个题要好好再理解消化几遍 . . . . .	277
18.1.22407. Longest Increasing Subsequence II: 【线段树】: 【贴出来方便自己查询, 解题印象深刻】活宝妹就是一定要嫁给亲爱的表哥!!! . . . . .	285
18.1.31157. Online Majority Element In Subarray - Hard . . . . .	287
18.1.41825. Finding MK Average - Hard . . . . .	289
18.1.5315. Count of Smaller Numbers After Self - Hard . . . . .	292
18.1.6699. Falling Squares - Hard . . . . .	293
18.1.71483. Kth Ancestor of a Tree Node - Hard 倍增法 binary lifting . . . . .	296
18.1.8236 二叉树的最近公共祖先 . . . . .	297
18.1.91505. Minimum Possible Integer After at Most K Adjacent Swaps On Digits - Hard BIT 树状数组 . . . . .	297
18.2求最大最小值、位操作值的线段树 . . . . .	297

# Chapter 1

## Dynamic Programming, 动态规划

### 1.1 第二次仍不会的题号记这里

- 329 494 131 518 1723(need summary)
- 需要重写的: 1187, 2035, 1000
- 根据 CLRS, 动态规划分为两种:
- top-down with memoization (递归记忆化搜索)

等价于带缓存的, 搜索树上的 DFS 比较贴近于新问题正常的思考习惯

- bottom-up (自底向上循环迭代)

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

#### 1.1.1 518. 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];  
}
```

#### 1.1.2 464. Can I Win 这个题: 为什么顺序无关了?

In the "100 game" two players take turns adding, to a running total, any integer from 1 to 10. The player who first causes the running total to reach or exceed 100 wins. What if we change the game so that players cannot re-use integers? For example, two players might take turns drawing from a common pool of numbers from 1 to 15 without replacement until they reach a total  $\geq 100$ . 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)
    Arrays.fill(dp, -1);
    return dfs(maxChoosableInteger, desiredTotal, 0, 0);
}

```

- 另外这第二次又看见的解法

```

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

```

- // 下面这个效率更高

```

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

```

### 1.1.3 494. 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  $\text{nums} = [2, 1]$ , 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，在面试中很可能考到。该题有多种解法。
- 第一种解法：DFS，brute force。我们对  $\text{nums}$  数组中的每个数字，都尝试在其前面添加正号和负号，最后暴力求解，统计数组中各数字组合值为 target 的情况。（该理解是错误的，我们可以使用带备忘录机制的自顶向下的 DP 方法，代码见下）

### 1. 回溯 $O(2^N)$

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

### 2. 解题思路与分析: dfs 记忆化搜索

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

- 上面的方法比较慢，下面这个效率更好一点儿

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

### 3. DP

```
// sum[p] + sum[n] = sum[nums];
// sum[p] - sum[n] = S;
// 2sum[p] = sum[nums] + S
// sum[p] = (sum[nums] + S) / 2
public int findTargetSumWays(int[] a, int S) {
    int sum = Arrays.stream(a).sum(), target = (sum + S) / 2; // 根据推导公式，计算出 target
```

```

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

```

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

$$V_0 = \{0\} \quad // 表示前 0 个数的和为 0$$

$$V_i = \{V_{i-1} + a_i\} \cup \{V_{i-1} - a_i\}$$

$V_n$  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  $2^{\text{sum}} + 1$  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,  $m = 0, 1, \dots, \text{nums.length}$ 。可参考下图

#### Why DP works?

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

$$\pm a_1 \pm a_2 \pm a_3 \dots \pm a_n = \text{target}$$

$O(2^n)$  combinations, but

$$S = 2 * \text{sum}(a) + 1 \text{ total possible sums}, S \leq 2000 + 1$$

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

$$V_0 = \{0\}$$

$$V_i = \{V_{i-1} + a_i\} \cup \{V_{i-1} - a_i\}$$

Check target in  $V_n$

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

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

input: [1,1,1,1,1], target = 3

$2^5 = 32$  combination, but total 6 distinct values, max is 11 ( $2^5 + 1$ )

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

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

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

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

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

<https://www.cnblogs.com/cnoodle/p/14869498.html> [https://leetcode.com/problems/target-sum/discuss/97334/Java-\(15-ms\)-C++-\(3-ms\)-O\(ns\)-iterative-DP-solution-using-subset-sum-with-explanation/239290](https://leetcode.com/problems/target-sum/discuss/97334/Java-(15-ms)-C++-(3-ms)-O(ns)-iterative-DP-solution-using-subset-sum-with-explanation/239290) <http://www.>

[noteanddata.com/leetcode-494-Target-Sum-javascript-solution-note.html](http://noteanddata.com/leetcode-494-Target-Sum-javascript-solution-note.html) <https://www.i4k.xyz/article/gqk289/>  
[54709004 https://github.com/cherryljr/LeetCode/blob/master/Target%20Sum.java](https://github.com/cherryljr/LeetCode/blob/master/Target%20Sum.java)

5. 718. 最长重复子数组 (类似题目, 只是由字符串变为数组)

6. 72. 编辑距离

7. 1143. 最长公共子序列

8. 583. 两个字符串的删除操作

9. 727. 最小窗口子序列

你会发现这些都是求 2 个字符串 (或数组) 之间的某种关系的题目

#### 1.1.4 801. Minimum Swaps To Make Sequences Increasing - Hard

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

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

An array `arr` is strictly increasing if and only if  $\text{arr}^1 < \text{arr}^2 < \text{arr}^3 < \dots < \text{arr}[\text{arr.length} - 1]$ .

1. 解题思路与分析

```
// 设 dp[0][i] 表示不交换 A[i] 和 B[i] 在下标 i 的交换次数
// 设 dp[1][i] 表示交换 A[i] 和 B[i] 在下标 i 的交换次数
// 可以看到交换与否只取决于前一个状态, 可以将空间复杂度压缩到 O(1)
// 时间复杂度为 O(n), 空间复杂度为 O(1)
public int minSwap(int[] a, int[] b) {
    int n = a.length;
    int [][] dp = new int [n][2]; // 0: 不换, 1: 换
    for (int i = 0; i < n; i++)
        Arrays.fill(dp[i], Integer.MAX_VALUE);
    dp[0][0] = 0;
    dp[0][1] = 1;
    for (int i = 1; i < n; i++) {
        if (a[i] > a[i-1] && b[i] > b[i-1]) {
            dp[i][0] = Math.min(dp[i][0], dp[i-1][0]); // 不换, 取一个较小值
            dp[i][1] = Math.min(dp[i][1], dp[i-1][1] + 1); // 换就两个都换
        }
        if (a[i] > b[i-1] && b[i] > a[i-1]) {
            dp[i][0] = Math.min(dp[i][0], dp[i-1][1]);
            dp[i][1] = Math.min(dp[i][1], dp[i-1][0] + 1);
        }
    }
    return Math.min(dp[n-1][0], dp[n-1][1]);
}
```

#### 1.1.5 1639. Number of Ways to Form a Target String Given a Dictionary - Hard

You are given a list of strings of the same length `words` and a string `target`.

Your task is to form `target` using the given words under the following rules:

`target` should be formed from left to right. To form the  $i$ th character (0-indexed) of `target`, you can choose the  $k$ th character of the  $j$ th string in `words` if `target[i] = words[j][k]`. Once you use the  $k$ th character of the  $j$ th string of `words`, you can no longer use the  $x$ th character of any string in `words` where  $x \leq k$ . In other words, all characters to the left of or at index  $k$  become unusable for every string. Repeat the process until you form the string `target`. Notice that you can use multiple characters from the same string in `words` provided the conditions above are met.

Return the number of ways to form `target` from `words`. Since the answer may be too large, return it modulo 109 + 7.

1. 解题思路与分析: dp

思路:  
 $dp[i][j]$  表示: `words` 字符串列表的前  $j$  列来构造目标字符串 `target` 的前  $i$  个字符;  
 $cnt[i][j]$  表示: `words` 字符串列表的第  $i$  列一共有多少字符  $j$ ;  
那公式就很好推出来了:  
1. 第  $i$  个字符不使用第  $j$  列时, 即通过前  $j - 1$  列得到  
 $dp[i][j] = dp[i][j-1];$   
2. 第  $i$  个字符使用第  $j$  列时  
\*  $dp[i][j] = dp[i-1][j-1] * cnt[j][第 i 个字符];$   
==> $dp[i][j] = dp[i-1][j-1] + dp[i-1][j-1] * cnt[j][第 i 个字符]$

<sup>1</sup>DEFINITION NOT FOUND.

<sup>2</sup>DEFINITION NOT FOUND.

<sup>3</sup>DEFINITION NOT FOUND.

```

static final int mod = (int)1e9 + 7;
public int numWays(String[] words, String target) {
    int m = target.length(), n = words[0].length();
    char [] s = target.toCharArray();
    int [][] cnt = new int [n][26];
    for (String w : words)
        for (int j = 0; j < n; j++)
            cnt[j][w.charAt(j)-'a']++;
    // long [][] dp = new long [m][n];
    // dp[0][0] = cnt[0][s[0]-'a'];
    // for (int i = 1; i < n; i++) // 初始化: 由前 i 列来构成 target 第一个字符的方案数
    //     dp[0][i] = (dp[0][i] + dp[0][i-1] + cnt[i][s[0]-'a']) % mod;
    // for (int i = 1; i < m; i++)
    //     for (int j = i; j < n; j++)
    //         dp[i][j] = (dp[i][j-1] + dp[i-1][j-1] * cnt[j][s[i]-'a']) % mod;
    // return (int)dp[m-1][n-1];
    long [] dp = new long [m+1][n+1];
    Arrays.fill(dp[0], 1L);
    // dp[0] = LongStream.range(0, n+1).map(e->1).toArray(); // 上下两行, 效果差不多, filling first row of array with 1
    for (int i = 1; i <= m; i++)
        for (int j = i; j <= n + i - m; j++)
            dp[i][j] = (dp[i][j-1] + dp[i-1][j-1] * cnt[j-1][s[i-1]-'a']) % mod;
    return (int)dp[m][n];
}

```

- dp 降维, 压缩空间

```

static final int mod = (int)1e9 + 7;
public int numWays(String[] words, String target) { // dp 降维, 压缩空间, 但二维 dp 仍然是思路最为清晰好理解的
    int m = target.length(), n = words[0].length();
    char [] s = target.toCharArray();
    long [] dp = new long [m];
    for (int i = 0; i < n; i++) { // 遍历字符串数组的各列
        int [] cnt = new int [26]; // 当前-列-所有字符的出现次数
        for (String w : words)
            cnt[w.charAt(i)-'a']++;
        for (int j = Math.min(i, m-1); j >= 0; j--) // 记住: 降维就容易产生脏数据, 需要倒序遍历
            dp[j] = (dp[j] + (j > 0 ? dp[j-1] : 1) * cnt[s[j]-'a']) % mod;
    }
    return (int)dp[m-1];
}

```

## 1.2 区间型 DP

- <https://leetcode-cn.com/problems/minimum-cost-to-merge-stones/solution/yi-dong-you-yi-dao-nan-yi-bu-bu/>

区间 dp 问题, 旨在通过动态规划去求一个区间的最优解, 通过将大区间划分为很多个小区间, 再由小区间间的解来组合出大区间的解, 这体现了分治的思想。

- 区间动态规划三部曲

- 定义状态:  $dp[i, j]$  为区间  $[i, j]$  的最优解
- 定义状态转移方程: 最常见的写法为:  $dp[i, j] = \max/\min\{dp[i, j], dp[i, k] + dp[k+1, j] + cost\}$ 。选取  $[i, j]$  之间的一个分界点  $k$ , 分别计算  $[i, k]$  和  $[k+1, j]$  的最优解, 从而组合出  $[i, j]$  的最优解。
- 初始化:  $dp[i][i] = \text{常数}$ 。区间长度为 1 时的最优解应当是已知的。

假设要求的区间最优解为  $dp[1, n]$ , 区间 dp 问题有两种编码方法:

- 第一种:

```

for (int i = n; i >= 1; --i)
    for (int j = i + 1; j <= n; ++j)
        for (int k = i; k < j; ++k)
            dp[i, j] = max/min(dp[i, j], dp[i, k] + dp[k+1, j] + cost)

```

这种写法就是常规的 dp 写法, 枚举  $i$  为子区间左边界, 枚举  $j$  为子区间右边界, 枚举  $k$  为分界点。要注意由于要求的是  $dp[1, n]$ , 所以  $i$  必须从大往小遍历,  $j$  必须从小往大遍历。这样在状态转移方程中利用的就是已求解的 dp 状态。

- 第二种:

```

for (int len = 2; len <= n; ++len)
    for (int i = 1; i + len - 1 <= n; ++i) {
        int j = i + len - 1;
        for (int k = i; k < j; ++k)
            dp[i, j] = max/min(dp[i, j], dp[i, k] + dp[k+1, j] + cost);
    }

```

这种写法最常见, 枚举  $len$  为区间长度, 枚举  $i$  为区间左端点, 由此可以计算出区间右端点  $j$ , 枚举  $k$  为分界点。区间长度从 2 到  $n$ , 跟上一种写法相同。这种写法的正确性可能不如上一种那么直观, 它从小到大枚举出所有区间, 在求解大区间时, 状态转移方程中利用的状态都是小区间的状态, 必定在它之前被求解, 所以也是正确的。

### 1.2.1 1039. Minimum Score Triangulation of Polygon - Medium

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.

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

### 1.2.2 312. 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  $i$ th balloon, you will get  $\text{nums}[i - 1] * \text{nums}[i] * \text{nums}[i + 1]$  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]; // 没有这些个边界条件
    // if (x == y-1)
    //     return dp[x][y] = arr[x] * arr[y] + Math.max(arr[x], arr[y]);
    int max = 0;
    for (int i = x; i <= y; i++) {
        max = Math.max(max, memorizedSearch(arr, x, i-1) + memorizedSearch(arr, i+1, y) + arr[x-1]*arr[i]*arr[y+1]);
    }
    return dp[x][y] = max;
}
int[][] dp;
int n;
public int maxCoins(int[] nums) {
    int n = nums.length + 2;
    dp = new int[n][n];
    int[] arr = new int[n];
    System.arraycopy(nums, 0, arr, 1, n-2);
    arr[0] = arr[n-1] = 1;
    return memorizedSearch(arr, 1, n-2);
}
```

### 1.2.3 546. Remove Boxes - Hard: 带隐含信息，需要第三维参数加人的

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

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

Return the maximum points you can get.

### 1. 解题思路与分析

```
public int removeBoxes(int[] b) { // 区间型 dp
    n = b.length;
    dp = new int[n][n][n];
    return dfs(b, 0, n-1, 0);
}
int [[[ ] dp;
int n;
private int dfs(int[] a, int i, int j, int k) {
    if (i > j) return 0;
    if (dp[i][j][k] > 0) return dp[i][j][k];
    int ans = dfs(a, i, j-1, 0) + (k+1) * (k+1); // 消除 [i, j-1] 区间后, (k+1) 个 a[j] 就可以连续消除
    for (int x = i; x < j; x++) {
        if (a[x] == a[j]) // 试图先消除掉 [x+1, j-1] 范围内的数, 然后剩下 a[x], a[j] 以及 j 后面有 k 个连续与 a[j] 相等的数
            ans = Math.max(ans, dfs(a, x+1, j-1, 0) + dfs(a, i, x, k+1)); // [x+1, j-1] 消除后, a[x] 后面就跟了 k+1 个连续与 a[x] 相等的数
    }
    return dp[i][j][k] = ans;
}
```

### 1.2.4 664. Strange Printer - Hard

There is a strange printer with the following two special properties:

The printer can only print a sequence of the same character each time. At each turn, the printer can print new characters starting from and ending at any place and will cover the original existing characters. Given a string  $s$ , return the minimum number of turns the printer needed to print it.

### 1. 解题思路与分析

```
public int strangePrinter(String t) { // dfs + memo
    n = t.length();
    s = t.toCharArray();
    dp = new int[n][n];
    return dfs(0, n-1);
}
int [[[ ] dp;
char[] s;
int n;
private int dfs(int i, int j) {
    if (i > j) return 0;
    if (dp[i][j] > 0) return dp[i][j];
    int ans = dfs(i+1, j) + 1; // 初始化为先打 i 位置, 再打 [i+1, j] 区间覆盖原 [i, j] 区间
    for (int k = i+1; k <= j; k++) {
        if (s[i] == s[k])
            ans = Math.min(ans, dfs(i+1, k-1) + dfs(k, j));
    }
    return dp[i][j] = ans;
}
public int strangePrinter(String s) { // dp
    int n = s.length();
    int [[[ ] dp = new int[n][n];
    for (int i = n-1; i >= 0; i--)
        for (int j = i; j < n; j++) {
            dp[i][j] = i == j ? 1 : 1 + dp[i+1][j]; // 同样是先打出 [i, j] 区间一次, 再用 [i+1, j] 区间覆盖
            for (int k = i+1; k <= j; k++)
                if (s.charAt(k) == s.charAt(i)) // 如果存在相同的字符, 就可以进一步地优化
                    dp[i][j] = Math.min(dp[i][j], dp[i+1][k-1]+dp[k][j]);
        }
    return dp[0][n-1];
}
```

### 1.2.5 1591. Strange Printer II - Hard todo

There is a strange printer with the following two special requirements:

On each turn, the printer will print a solid rectangular pattern of a single color on the grid. This will cover up the existing colors in the rectangle. Once the printer has used a color for the above operation, the same color cannot be used again. You are given a  $m \times n$  matrix  $\text{targetGrid}$ , where  $\text{targetGrid}[\text{row}][\text{col}]$  is the color in the position  $(\text{row}, \text{col})$  of the grid.

Return true if it is possible to print the matrix  $\text{targetGrid}$ , otherwise, return false.

### 1. 解题思路与分析

关于含有隐藏信息的 dp 题目, 感觉巅峰就属于拣樱桃那题 Cherry Pickup ???

## 1.3 扫描线类、时间戳、一维线性 DP/ 单序列/ 接龙型

### 1.3.1 805. Split Array With Same Average 【活宝妹就是一定要嫁给亲爱的表哥!!!】

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. 折半搜索

##### 思路与算法

设数组 `nums` 的长度为  $n$ , 假设我们移动了  $k$  个元素到数组 `A` 中, 移动了  $n - k$  个元素到数组 `B` 中, 此时两个数组的平均值相等。设  $\text{sum}(A), \text{sum}(B), \text{sum}(\text{nums})$  分别表示数组 `A, B, nums` 的元素和, 由于数组 `A, B` 的平均值相等, 则有  $\frac{\text{sum}(A)}{k} = \frac{\text{sum}(B)}{n - k}$ , 上述等式可以进行变换为如下:

$$\begin{aligned} & \text{sum}(A) \times (n - k) = \text{sum}(B) \times k \\ \Leftrightarrow & \text{sum}(A) \times n = (\text{sum}(A) + \text{sum}(B)) \times k \\ \Leftrightarrow & \text{sum}(A) \times n = (\text{sum}(\text{nums})) \times k \\ \Leftrightarrow & \frac{\text{sum}(A)}{k} = \frac{\text{sum}(\text{nums})}{n} \end{aligned}$$

即数组 `A` 的平均值与数组 `nums` 的平均值相等。此时我们可以将数组 `nums` 中的每个元素减去 `nums` 的平均值, 这样数组 `nums` 的平均值则变为 0。此时题目中的问题则可以转变为: 从 `nums` 中找出若干个元素组成集合 `A`, 使得 `A` 的元素之和为 0, 剩下的元素组成集合 `B`, 它们的和也同样为 0。

比较容易想到的思路是, 从  $n$  个元素中取出若干个元素形成不同的子集, 则此时一共有  $2^n$  种方法 (即每个元素取或不取), 我们依次判断每种方法选出的元素之和是否为 0。但题目中的  $n$  可以达到 30, 此时  $2^{30} = 1,073,741,824$ , 组合的数据非常大。

因此我们考虑将数组平均分成两部分 `A0` 和 `B0`, 它们均含有  $\frac{n}{2}$  个元素 (不失一般性, 这里假设  $n$  为偶数。如果  $n$  为奇数, 在 `A0` 中多放一个元素即可), 此时这两个数组分别有  $2^{\frac{n}{2}}$  种不同的子集选择的方法。设 `A0` 中所有选择的方法得到的不同子集的元素之和的集合为 `left`, `B0` 中所有选择的方法得到的不同子集的元素之和的集合为 `right`, 那么我们只需要在 `left` 中找出一个子集 `A'_0` 的元素之和为  $x$ , 同时在 `right` 中包含一个子集 `B'_0` 的元素之和为  $-x$ , 则子集 `A'_0, B'_0` 构成的集合的元素之和为 0, 此时则找到了一种和为 0 的方法。

- 需要注意的是, 我们不能同时选择 `A0` 和 `B0` 中的所有元素, 这样数组 `B` 就为空了。

此外, 将数组 `nums` 中每个元素减去它们的平均值, 这一步会引入浮点数, 可能会导致判断的时候出现误差。一种解决方案是使用分数代替浮点数, 但这样代码编写起来较为麻烦。更好的解决方案是先将 `nums` 中的每个元素乘以 `nums` 的长度, 则此时每个元素 `nums[i]` 变为  $n \times \text{nums}[i]$ , 这样数组 `nums` 的平均值一定为整数, 同时也不影响题目中的平均值相等的要求。

```
// 第一种方法: 原始, 相对比较土。。。
public boolean splitArraySameAverage(int[] a) {
    if (a.length == 1) return false;
    int n = a.length, sum = Arrays.stream(a).sum(), m = n / 2;
    // 把原数组和变 0, 并且不引入任何浮点数
    for (int i = 0; i < n; i++) a[i] = a[i] * n - sum;
    // // 把原数组分成两个大小最接近的数组【再说一遍: 概念上的, 并不真的需要物理上的拆分为两个数组。。。】
    // int [] l = Arrays.copyOfRange(a, 0, n/2); // 【l-inclusive, r-exclusive】
    // int [] r = Arrays.copyOfRange(a, n/2, n);
    // // 求一个数组可能取得的和 sum 集合
    Set<Integer> lsum = new HashSet<>();
    for (int i = 1; i < (1 << m); i++) {
        int s = 0;
        for (int j = 0; j < m; j++) // 这里可能会遍历多了
            if (((i >> j) & 1) == 1) s += a[j];
        // 【快速返回】这里是优化提速的地方。左边非空集合和为 0, 那么左边非空补集的和也一定为 0
        if (s == 0) return true;
        lsum.add(s);
    }
    // 遍历: 求右边数组可能取得的和 s, 并一一去找左边是否存在和 -s
    int rsum = 0; // 先求右边的和
    for (int i = 1; i < (1 << (n - m)); i++) {
        int s = 0;
        for (int j = m; j < n; j++) // 这里可能会遍历多了
            if (((i >> (j - m)) & 1) == 1) s += a[j];
        if (lsum.contains(-s)) return true;
    }
}
```

```

for (int i = m; i < n; i++) rsum += a[i];
for (int i = 1; i < (1 << n - m); i++) {
    int s = 0;
    for (int j = m; j < n; j++) // 这里可能会遍历多了
        if (((i >> (j - m)) & 1) == 1) s += a[j];
    // if (s == 0 || lsum.contains(-s)) return true; // 【BUG:】
    if (s == 0 || s != rsum && lsum.contains(-s)) return true; // 【为什么 s != rsum 重要?】【3, 1】==>【2, -2】
}
return false;
}

```

- 复杂度分析

### 复杂度分析

- 时间复杂度:  $O(n \times 2^{\frac{n}{2}})$ , 其中  $n$  表示数组的长度。我们需要求出每个子集的元素之和, 数组的长度为  $n$ , 一共有  $2^{\frac{n}{2}}$  个子集, 求每个子集的元素之和需要的时间为  $O(n)$ , 因此总的时间复杂度为  $O(n \times 2^{\frac{n}{2}})$ 。
- 空间复杂度:  $O(2^{\frac{n}{2}})$ 。一共有  $2^{\frac{n}{2}}$  个子集的元素之和需要保存, 因此需要的空间为  $O(2^{\frac{n}{2}})$ 。

## 2. 动态规划

### 方法二: 动态规划

#### 思路与算法

根据方法一的结论, 设数组  $nums$  的平均数为  $avg = \frac{sums}{n}$ , 我们移动了  $k$  个元素到数组  $A$  中, 则此时已经数组  $A$  的平均数也为  $avg$ , 则此时数组  $A$  的元素之和为  $sum(A) = k \times avg$ , 则该问题可以等价于在数组中取  $k$  个数, 使得其和为  $k \times avg$ 。对应的「[0-1背包问题](#)」则为是否可以取  $k$  件物品使得背包的重量为  $k \times avg$ 。我们设  $dp[i][x]$  表示当前已从数组  $nums$  取出  $i$  个元素构成的和为  $x$  的可能性:

- 如果  $dp[i][x] = true$ , 表示当前已经遍历过的元素中可以取出  $i$  个元素构成的和为  $x$ ;
- 如果  $dp[i][x] = false$ , 表示当前已经遍历过的元素中不存在取出  $i$  个元素的和等于  $x$ ;

假设前  $j - 1$  个元素中存在长度为  $i$  的子集且子集的和为  $x$ , 则此时  $dp[i][x] = true$ , 我们当前遍历  $nums[j]$  时, 则可以推出一定存在长度为  $i + 1$  的子集且满足子集的和为  $x + nums[j]$ , 可以得到状态转移方程为:

$$dp[i + 1][x + nums[j]] = dp[i][x]$$

根据题意可知:  $\frac{sum(A)}{k} = \frac{sum(nums)}{n}$ , 可以变换为:  $sum(A) \times n = sum(nums) \times k$ , 所以我们只需要找到一个元素个数为  $k$  的子集  $A$  满足上述条件即可, 因此我们利用上述递推公式求出所有可能长度的子集和组合即可, 此时需要的时间复杂度约为  $n^2 \times sum(nums)$ , 按照题目给定的测试用例可能会超时, 所以需要一些减枝技巧, 具体的减枝技巧如下:

- 根据题意可以推出  $sum(A) = \frac{sum(nums) \times k}{n}$ , 则此时如果满足题目要求, 则一定存在整数  $k \in [1, n]$ , 使得  $[sum(nums) \times k] \bmod n = 0$ , 因此我们可以提前是否存在  $k$  满足上述要求, 如果不存在则可以提前终止。
- 根据题目要求可以划分为两个子数组  $A, B$ , 则可以知道两个子数组中一定存在一个数组的长度小于等于  $\frac{n}{2}$ , 因此我们只需检测数组长度小于等于  $\frac{n}{2}$  的子数组中是否存在其和满足  $subsum \times n = nums \times k$  即可。

```

public boolean splitArraySameAverage(int[] a) {
    if (a.length == 1) return false;
    int n = a.length, m = n / 2, sum = Arrays.stream(a).sum();
    boolean isPossible = false;
    for (int i = 1; i <= m; i++) {
        if (sum * i % n == 0) {
            isPossible = true;
            break;
        }
    }
    if (!isPossible) return false;
    Set<Integer>[] f = new HashSet[m + 1];
    Arrays.setAll(f, z -> new HashSet<z>());
    f[0].add(0);
    for (int v : a)
        for (int i = m; i >= 1; i--) {
            for (int x : f[i - 1]) {
                int cur = x + v;
                if (cur <= sum && cur % n == 0)
                    f[i].add(cur);
            }
        }
}

```

```

        if (cur * n == sum * i) return true;
        f[i].add(cur);
    }
    return false;
}

```

- 复杂度分析：

### 复杂度分析

- 时间复杂度： $O(n^2 \times \text{sum}(\text{nums}))$ ，其中  $n$  表示数组的长度， $\text{sum}(\text{nums})$  表示数组  $\text{nums}$  的和。我们需要求出给定长度下所有可能的子集元素之和，数组的长度为  $n$ ，每种长度下子集的和最多有  $\text{sum}(\text{nums})$  个，因此总的时间复杂度为  $O(n^2 \times \text{sum}(\text{nums}))$ 。
- 空间复杂度： $O(n \times \text{sum}(\text{nums}))$ 。一共有  $n$  种长度的子集，每种长度的子集和最多有  $\text{sum}(\text{nums})$  个，因此需要的空间为  $O(n \times \text{sum}(\text{nums}))$ 。

## 1.3.2 805. Split Array With Same Average 【这里是怎们又重复了一遍？】需要合并一下

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

```

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

```

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

```

### 1.3.3 2008. Maximum Earnings From Taxi - Medium

There are  $n$  points on a road you are driving your taxi on. The  $n$  points on the road are labeled from 1 to  $n$  in the direction you are going, and you want to drive from point 1 to point  $n$  to make money by picking up passengers. You cannot change the direction of the taxi.

The passengers are represented by a 0-indexed 2D integer array  $\text{rides}$ , where  $\text{rides}[i] = [\text{start}_i, \text{end}_i, \text{tip}_i]$  denotes the  $i$ th passenger requesting a ride from point  $\text{start}_i$  to point  $\text{end}_i$  who is willing to give a  $\text{tip}_i$  dollar tip.

For each passenger  $i$  you pick up, you earn  $\text{end}_i - \text{start}_i + \text{tip}_i$  dollars. You may only drive at most one passenger at a time.

Given  $n$  and  $\text{rides}$ , return the maximum number of dollars you can earn by picking up the passengers optimally.

Note: You may drop off a passenger and pick up a different passenger at the same point.

#### 1. 解题思路与分析

```

public long maxTaxiEarnings(int n, int[][] rides) {
    Arrays.sort(rides, (a, b) -> (a[0] != b[0] ? a[0] - b[0] : a[1] - b[1]));
    Map<Integer, Set<int []>> m = new HashMap<>();
    for (int [] r : rides)
        m.computeIfAbsent(r[1], z -> new HashSet<>()).add(r);
    long [] dp = new long [n+1];
    for (int i = 1; i <= n; i++) {
        dp[i] = dp[i-1];
        if (m.containsKey(i))
            for (int [] r : m.get(i))
                dp[r[1]] = Math.max(dp[r[1]], dp[r[0]] + r[1] - r[0] + r[2]);
    }
    return dp[n];
}

```

#### 2. 解题思路与分析

```

// Similar to 1235. Maximum Profit in Job Scheduling
// Sort by the end time to get non-overlapping intervals.
// Use the treemap to find the previous ride before the current ride.
public long maxTaxiEarnings(int n, int[][] rides) {
    if (rides == null || rides.length == 0) return 0;
    for (int[] r : rides)
        r[2] = r[1] - r[0] + r[2];
    Arrays.sort(rides, (a, b) -> (a[1] - b[1]));
    TreeMap<Long, Long> map = new TreeMap<>();
    map.put((long)0, (long)0);
    for (int[] r : rides) {
        long cur = map.floorEntry((long)r[0]).getValue() + r[2];
        if (cur > map.lastEntry().getValue()) {
            map.put((long)r[1], cur);
        }
    }
    return map.lastEntry().getValue();
}

```

### 1.3.4 1713. Minimum Operations to Make a Subsequence - Hard LIS 经典题型，需要吃透

You are given an array  $\text{target}$  that consists of distinct integers and another integer array  $\text{arr}$  that can have duplicates.

In one operation, you can insert any integer at any position in arr. For example, if arr = [1,4,1,2], you can add 3 in the middle and make it [1,4,3,1,2]. Note that you can insert the integer at the very beginning or end of the array.

Return the minimum number of operations needed to make target a subsequence of arr.

A subsequence of an array is a new array generated from the original array by deleting some elements (possibly none) without changing the remaining elements' relative order. For example, [2,7,4] is a subsequence of [4,2,3,7,2,1,4] (the underlined elements), while [2,4,2] is not.

### 1. 解题思路与分析

```
public int minOperations(int[] t, int[] a) {
    int n = t.length;
    Map<Integer, Integer> m = new HashMap<Integer, Integer>();
    for (int i = 0; i < n; ++i)
        m.put(t[i], i);
    List<Integer> d = new ArrayList<Integer>();
    for (int val : a)
        if (m.containsKey(val)) {
            int idx = m.get(val);
            int it = binarySearch(d, idx);
            if (it != d.size())
                d.set(it, idx);
            else
                d.add(idx);
        }
    return n - d.size();
}
public int binarySearch(List<Integer> li, int t) {
    int size = li.size();
    if (size == 0 || li.get(size - 1) < t)
        return size;
    int low = 0, high = size - 1;
    while (low < high) {
        int mid = (high - low) / 2 + low;
        if (li.get(mid) < t)
            low = mid + 1;
        else
            high = mid;
    }
    return low;
}
```

### 1.3.5 1883. Minimum Skips to Arrive at Meeting On Time - Hard

You are given an integer hoursBefore, the number of hours you have to travel to your meeting. To arrive at your meeting, you have to travel through n roads. The road lengths are given as an integer array dist of length n, where dist[i] describes the length of the ith road in kilometers. In addition, you are given an integer speed, which is the speed (in km/h) you will travel at.

After you travel road i, you must rest and wait for the next integer hour before you can begin traveling on the next road. Note that you do not have to rest after traveling the last road because you are already at the meeting.

For example, if traveling a road takes 1.4 hours, you must wait until the 2 hour mark before traveling the next road. If traveling a road takes exactly 2 hours, you do not need to wait. However, you are allowed to skip some rests to be able to arrive on time, meaning you do not need to wait for the next integer hour. Note that this means you may finish traveling future roads at different hour marks.

For example, suppose traveling the first road takes 1.4 hours and traveling the second road takes 0.6 hours. Skipping the rest after the first road will mean you finish traveling the second road right at the 2 hour mark, letting you start traveling the third road immediately. Return the minimum number of skips required to arrive at the meeting on time, or -1 if it is impossible.

### 1. 解题思路与分析

```
// dp[i][j] 表示途径 i 条道路跳过 j 次休息情况下的最小用时，遍历过程中根据上一道路是否休息选取最小值，结合状态转移方程求解。
public int minSkips(int[] dist, int speed, int hoursBefore) {
    int n = dist.length;
    double[][] dp = new double[n+1][n+1]; // dp[i][j]: 途径 i 条道路，跳过 j 次休息下的最小用时
    for (int i = 0; i <= n; i++)
        Arrays.fill(dp[i], Integer.MAX_VALUE);
    dp[0][0] = 0;
    double eps = 1e-8; // eps 用于避免浮点数计算误差导致向上取整后出现错误，inf 作为最大值初始化动态规划数组
    for (int i = 1; i <= n; i++) {
        double t = (double)dist[i-1] / speed; // 第 i 条道路耗时
        dp[i][0] = Math.ceil(dp[i-1][0] - eps) + t; // 单独计算不跳过休息时的值
        dp[i][i] = dp[i-1][i-1] + t; // 单独计算跳过所有休息时的值
        for (int j = i-1; j > 0; j--) // 根据上一条路是否休息，来优化最小值
            dp[i][j] = Math.min(Math.ceil(dp[i-1][j] - eps) + t, dp[i-1][j-1] + t);
    }
    for (int i = 0; i <= n; i++)
        if (dp[n][i] <= hoursBefore + eps) return i;
    return -1;
}
```

### 1.3.6 1911. Maximum Alternating Subsequence Sum - Medium todo: 还需要总结题解

The alternating sum of a 0-indexed array is defined as the sum of the elements at even indices minus the sum of the elements at odd indices.

For example, the alternating sum of [4,2,5,3] is  $(4 + 5) - (2 + 3) = 4$ . Given an array nums, return the maximum alternating sum of any subsequence of nums (after reindexing the elements of the subsequence).

A subsequence of an array is a new array generated from the original array by deleting some elements (possibly none) without changing the remaining elements' relative order. For example, [2,7,4] is a subsequence of [4,2,3,7,2,1,4] (the underlined elements), while [2,4,2] is not.

#### 1. 解题思路与分析: DP

设计两个长整数 evenDp 和 oddDp, 分别记录上一元素为偶数下标、奇数下标时当前的最大交替和。根据是否添加当前元素, 状态转移方程为:

$$\text{evenDp} = \text{Math.max}(\text{上一 evenDp}, \text{上一 oddDp} + \text{当前元素})$$

$$\text{oddDp} = \text{Math.max}(\text{上一 oddDp}, \text{上一 evenDp} + \text{当前元素})$$

最终得到的 evenDp 即为最大交替和。

```
public long maxAlternatingSum(int[] a) {
    long odd = 0, evn = a[0]; // 上一元素为偶数下标、奇数下标时的最大交替和
    for (int i = 1; i < a.length; i++) {
        evn = Math.max(evn, odd + a[i]); // 偶数下标交替和转移
        odd = Math.max(odd, evn - a[i]); // 奇数下标交替和转移
    }
    return evn;
}
```

#### 2. 解题思路与分析: 最大股票收益 参考 Leetcode 题解, 发现有一个方法很巧妙。将样例 [6,2,1,2,4,5] 转化为 [0,6,2,1,2,4,5], 那么题面就转化为模拟股票交易, 数组中的数为股票价格, index 为天数。

你可以在第  $i$  天买入股票, 第  $j$  天卖出股票, 其中  $i \leq j$ 。

那么其实我们可以用上帝视角来看, 只要股票价格后一天比当天高, 我们就当天买入, 后一天卖出。

那么就如下所示:

买入	卖出	收益
第0天	第1天	6-0=6
第3天	第4天	2-1=1
第4天	第5天	4-2=2
第5天	第6天	5-4=1

那么总收益为  $6+1+2+1=10$ , 即  $6-0+2-1+4-2+5-4$ , 抵消之后就是  $6-1+5$ , 就是样例中的最优子序列 [6,1,5]~

```
public long maxAlternatingSum(int[] a) {
    int [] b = new int [a.length+1];
    System.arraycopy(a, 0, b, 1, a.length);
    long ans = 0;
    for (int i = 1; i < b.length; i++)
        if (b[i] - b[i-1] > 0) ans += b[i] - b[i-1];
    return ans;
}
```

### 1.3.7 1928. Minimum Cost to Reach Destination in Time - Hard

There is a country of  $n$  cities numbered from 0 to  $n - 1$  where all the cities are connected by bi-directional roads. The roads are represented as a 2D integer array edges where  $\text{edges}[i] = [\text{xi}, \text{yi}, \text{time}_i]$  denotes a road between cities  $\text{xi}$  and  $\text{yi}$  that takes  $\text{time}_i$  minutes to travel. There may be multiple roads of differing travel times connecting the same two cities, but no road connects a city to itself.

Each time you pass through a city, you must pay a passing fee. This is represented as a 0-indexed integer array passingFees of length  $n$  where  $\text{passingFees}[j]$  is the amount of dollars you must pay when you pass through city  $j$ .

In the beginning, you are at city 0 and want to reach city  $n - 1$  in  $\text{maxTime}$  minutes or less. The cost of your journey is the summation of passing fees for each city that you passed through at some moment of your journey (including the source and destination cities).

Given  $\text{maxTime}$ , edges, and passingFees, return the minimum cost to complete your journey, or -1 if you cannot complete it within  $\text{maxTime}$  minutes.

#### 1. 解题思路与分析

```

// 设计一个动态规划数组 dp[maxTime + 1][n]，其中 dp[t][i] 表示第 t 分钟到达城市 i 时的最少费用，则状态转移方程为：
// dp[t][c1] = Math.min(dp[t][c1], dp[t - time][c2] + passingFees[c1])
// dp[t][c2] = Math.min(dp[t][c2], dp[t - time][c1] + passingFees[c2])
public int minCost(int maxTime, int[][] edges, int[] passingFees) {
    int n = passingFees.length;
    int [][] dp = new int [maxTime+1][n];
    for (int i = 0; i <= maxTime; i++)
        Arrays.fill(dp[i], Integer.MAX_VALUE / 2);
    dp[0][0] = passingFees[0];
    for (int t = 0; t <= maxTime; t++)
        for (int [] e : edges) {
            if (e[2] > t) continue;
            int u = e[0], v = e[1], time = e[2];
            dp[t][u] = Math.min(dp[t][u], dp[t-time][v] + passingFees[u]); // v --> u
            dp[t][v] = Math.min(dp[t][v], dp[t-time][u] + passingFees[v]); // u --> v
        }
    int ans = Integer.MAX_VALUE / 2;
    for (int i = 1; i <= maxTime; i++)
        ans = Math.min(ans, dp[i][n-1]);
    return ans == Integer.MAX_VALUE / 2 ? -1 : ans;
}

```

### 1.3.8 730. Count Different Palindromic Subsequences - Hard

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

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

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

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

$\text{count("bccb")} = 2 * \text{count("cc")} + 2 = 6$ $\text{count("cc")} = 2 * \text{count("")} + 2 = 2$	“c”, “cc” “bcb”, “bccb” + “b”, “bb” “c”, “cc”
--	---

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

<b>Base case:</b> $\text{count}(x) = 1 \text{ if } x \in \{"a", "b", "c", "d"\}$ $= 0 \text{ if } x \text{ is empty}$
---

count("bcbcb") = 2 * count("cbc") + 1 = 9	"b" "cbc" "c", "cc"   "bbb" "bcabc" "ccb" "bccb"   "bb"
count("cbc") = 2 * count("b") + 2 = 4	"b"   "cbc"   "c" "cc"
count("bbcabbb")=2*count("bcab")-count("ca")=10	"bcab" "bab" "b" "bb" "c" "a"   "bbcab" "bbcab" "bbb" "bbbb" "beb" "bab"
count("bcab") =2*count("ca")+2 = 6	"c" "a"   "bcb" "bab"   "b" "bb"
$dp[i][j] = \text{count of different palindromic subseq of } s[i] \sim s[j] \text{ (inclusive)}$ <pre> dp[i][j] =     if s[i] != s[j]:         dp[i][j - 1] + dp[i + 1][j] - dp[i + 1][j - 1],      # count("abcd")         # count("abc") + count("bcd") - count("bc")      if s[i] == s[j]:         dp[i + 1][j - 1] * 2 + 2,                          s[i] not in s[i+1:j-1]         dp[i + 1][j - 1] * 2 + 1,                          count(s[i+1:j-1], s[i]) == 1         dp[i + 1][j - 1] * 2 - dp[1 + 1][r - 1]          l, r is the first/last pos of s[i] in s[i+1:j-1]</pre>	

```

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

- dp

```

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

### 1.3.9 1125. Smallest Sufficient Team - Hard 这个题要多写几遍

In a project, you have a list of required skills `req_skills`, and a list of people. The  $i$ th 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 `people1`, `people2`, and `people4`. 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.

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

- Java soution using Bit DP 10ms

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

<sup>4</sup>DEFINITION NOT FOUND.

```
    return res;
}
```

- DFS + Memorization (A real O( $2^{\text{skill}} * \text{people}$ ) Solution) Java 8ms

- [https://leetcode.com/problems/smallest-sufficient-team/discuss/1011135/DFS-%2B-Memorization-\(A-real-O\(2skill-\\*-people\)-Solution\)-Java-8ms](https://leetcode.com/problems/smallest-sufficient-team/discuss/1011135/DFS-%2B-Memorization-(A-real-O(2skill-*-people)-Solution)-Java-8ms)

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

- Recursion + Memoization + bit mask , with Simple JAVA solution

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

上面的这些方法相对较偏，就暂时顾不上了

### 1.3.10 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  $j \neq i$  and  $0 \leq j < \text{locations.length}$  and move to city j. Moving from city i to city j reduces the amount of fuel you have by  $|\text{locations}[i] - \text{locations}[j]|$ . Please notice that  $|x|$  denotes the absolute value of x.

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

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

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

```

// 自顶向下 (记忆化搜索)
// 每个 dfs 搜索当前状态为城市 i, 油量 f 到达终点的方案数。这样决策的时候就很直观：当前这个状态的方案数，由可去的城市的，且油量为剩余油量的到达终点方案数加起来。
// 初始化：每个状态都初始化为 -1。
// 当走到终点时，这个状态的可走到终点的方案数 +1。
private int dfs(int[] arr, int end, int idx, int fu) {
    if (dp[idx][fu] != -1) return dp[idx][fu];
    dp[idx][fu] = 0;
    if (idx == end) {
        dp[idx][fu] += 1;
        dp[idx][fu] %= mod;
    }
    for (int i = 0; i < n; i++) {
        if (i == idx || Math.abs(arr[i] - arr[idx]) > fu) continue;
        dp[idx][fu] = (dp[idx][fu] + dfs(arr, end, i, fu - Math.abs(arr[i] - arr[idx]))) % mod;
    }
    return dp[idx][fu];
}
int mod = (int)1e9 + 7;
int[][] dp;
int n;
public int countRoutes(int[] locations, int start, int finish, int fuel) {
    n = locations.length;
    if (fuel < Math.abs(locations[start] - locations[finish])) return 0;
    dp = new int[n][fuel+1];
    for (int i = 0; i < n; i++)
        Arrays.fill(dp[i], -1);
    dfs(locations, finish, start, fuel);
    return dp[start][fuel];
}

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

```

### 1.3.11 1012. Numbers With Repeated Digits - Hard 数位 DP + 压缩状态经典

Given an integer n, return the number of positive integers in the range [1, n] that have at least one repeated digit.

题意：统计 1-N 中，满足每个位置都不同的数有几个。

思路：数位 DP。通过一个 1«10 的 mask 表示当前这个数，1-9 哪些数被用了。

比赛的时候，一直想通过一个 dfs 直接找到不重复的数，一直不对。

赛后发现，别人都是通过一个 dfs 找重复的数，然后总个数减去。

```

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

```

```

int [][] dp;
int [] bit;
public int numDupDigitsAtMostN(int N) {
    int sum = N + 1;
    bit = new int [19];
    dp = new int [19][1 << 10][2];
    int idx = 0;
    while (N > 0) {
        bit[++idx] = N % 10;
        N /= 10;
    }
    return sum - dfs(idx, 1, 0);
}

```

## 1. 解题思路与分析: 降维一下数位 DP + 压缩状态经典

```

public int numDupDigitsAtMostN(int n) {
    if (n <= 10) return 0;
    int m = n + 1, r = 1 << 10, idx = 0; // r = 1 << 10 表示 n 值最多会有 10 个位, 通过记忆化暴搜每个位的可能性、来数 <=n 的不重复数的个数
    d = new int[10]; // n 转化为数组
    while (n != 0) {
        d[idx++] = n % 10;
        n /= 10;
    }
    dp = new int [r][idx];
    for (int i = 0; i < r; i++) Arrays.fill(dp[i], -1);
    return m - dfs(idx-1, 0, 1); // 自底向上
}

int [][] dp;
int [] d; // dfs: 返回不得复数的个数
private int dfs(int idx, int r, int l) { // l: limit flag: 当第一次搜到某数位, 该数位能取的最大值是受限制的
    if (idx == -1) return 1; // l: limit 有没有限制, 这个参数结合两种方法看得还是有些迷糊
    if (dp[r][idx] != -1 && l == 0) return dp[r][idx];
    int up = l == 1 ? d[idx] : 9, ans = 0; // 当前位的最大取值, 求出当前位最高可以枚举到哪个数字
    for (int i = 0; i <= up; i++) { // 遍历当前位的所有可能的取值: [0, 1, 2, ..., up]
        // 首先当前位的状态没有出现过
        // (本体计算的是不满足 至少两次的所有情况 逆向思维)
        if ((r & (1 << i)) == 0) { // 当前位第 i 位在 r 的状态里还没有出现过
            if (i == 0 && r == 0) // 001 的情况: 有前导 0, 所以 0 不能统计, 不更新 mask r(就是这个最高位为 0 的数不计入结果, 去遍历下一个低位的数。。)
                ans += dfs(idx-1, r, 0);
            else // 当前数没有前导 0、完全合法, 计入结果, 并进一步统计
                ans += dfs(idx-1, r | (1 << i), (l == 1 && i == d[idx] ? 1 : 0));
        }
    }
    if (l == 0) dp[r][idx] = ans; // 如果没有限制, 代表搜满了, 可以记忆化, 否则就不能
    return ans;
}

```

## 2. 解题思路与分析

这道题给了一个正整数 N, 让返回所有不大于 N 且至少有一个重复数字的正整数的个数, 题目中给的例子也可以很好的帮助我们理解。要求的是正整数的位数上至少要有一个重复数字, 当然最简单暴力的方法就是从 1 遍历到 N, 然后对于每个数字判断是否有重复数字, 看了一眼题目难度 Hard, 想都不用想, 肯定是超时的。这道题需要更高效的解法, 首先来想, 若是直接求至少有一个重复数字的正整数, 由于并不知道有多少个重复数字, 可能 1 个, 2 个, 甚至全是重复数字, 这样很难找到规律。有时候直接求一个问题不好求, 可以考虑求其相反的情况, 至少有一个重复数字反过来就是一个重复数字都没有, 所以这里可以求不大于 N 且一个重复数字都没有的正整数的个数, 然后用 N 减去这个数字即为所求。好, 接下来看怎么求, 对于任意一个 N, 比如 7918, 是个四位数, 而所有的三位数, 两位数, 一位数, 都一定比其小, 所以可以直接求出没有重复数字的三位数, 两位数, 和一位数。比如三位数, 由于百位上不能有 0, 则只有 9 种情况, 十位上可以有 0, 则有 9 种情况, 个位上则有 8 种情况, 所以就是  $9 \times 9 \times 8$ 。可以归纳出没有重复数字的 n 位数的个数, 最高位去除 0 还有 9 种, 剩余的 n-1 位则依次是 9, 8, 7... 则后面的 n-1 位其实是个全排列, 从 9 个数中取出 n-1 个数字的全排列, 初中就学过的。这里写一个全排列的子函数, 求从 m 个数字中取 n 个数字的全排列, 方便后面计算。算完这些后, 还要来算符合题意的四位数, 由于第一位是 7, 若千位上是小于 7 的数字 (共有 6 种, 千位上不能是 0), 则后面的百位, 十位, 个位又都可以全排列了, 从 9 个数字中取 3 个数字的全排列, 再乘以千位上小于 7 的 6 种情况。若当千位固定为 7, 则百位上可以放小于 9 的数字 (共有 8 种, 百位不能放 7, 但可以放 0), 则后面的十位和个位都可以全排列了, 从 8 个数字种取出 2 个数字的全排列, 再乘以百位上小于 9 的 8 种情况。需要注意的是, 遍历给定数字的各个位时, 有可能出现重复数字, 一旦出现了之后, 则该 prefix 就不能再用了, 因为已经不合题意了。所以要用一个 HashSet 来记录访问过的数字, 一旦遇到重复数字后就直接 break 掉。最后还有一个小 trick 需要注意, 由于 N 本身也需要计算进去, 所以再计算的时候, 使用 N+1 进行计算的话, 就可以把 N 这种情况算进去了

```

private int A(int m, int n) {
    return n == 0 ? 1 : A(m, n-1) * (m-n+1);
}

public int numDupDigitsAtMostN(int n) {
    List<Integer> digits = new ArrayList<>();
    Set<Integer> vis = new HashSet<>();
    for (int i = n+1; i > 0; i /= 10)
        digits.add(i % 10);
    int res = 0, m = digits.size();
    for (int i = 1; i < m; i++) res += 9 * A(9, i-1);
    for (int i = 0; i < m; i++) {

```

```

        for (int j = i > 0 ? 0 : 1; j < digits.get(i); ++j) {
            if (vis.contains(j)) continue;
            res += A(9-i, m-i-1);
        }
        if (vis.contains(digits.get(i))) break;
        vis.add(digits.get(i));
    }
    return n - res;
}

```

### 1.3.12 600. Non-negative Integers without Consecutive Ones - Hard

Given a positive integer  $n$ , return the number of the integers in the range  $[0, n]$  whose binary representations do not contain consecutive ones.

#### 1. 解题思路与分析

我们就可以通过 DP 的方法求出长度为  $k$  的二进制数的无连续 1 的数字个数。由于题目给我们的并不是一个二进制数的长度，而是一个二进制数，比如 100，如果我们按长度为 3 的情况计算无连续 1 点个数个数，就会多计算 101 这种情况。所以我们的目标是要将大于  $num$  的情况去掉。下面从头来分析代码，首先我们要把十进制数转为二进制数，将二进制数存在一个字符串中，并统计字符串的长度。然后我们利用这个帖子中的方法，计算该字符串长度的二进制数所有无连续 1 的数字个数，然后我们从倒数第二个字符开始往前遍历这个二进制数字符串，如果当前字符和后面一个位置的字符均为 1，说明我们并没有多计算任何情况，不明白的可以带例子来看。如果当前字符和后面一个位置的字符均为 0，说明我们有多计算一些情况，就像之前举的 100 这个例子，我们就多算了 101 这种情况。我们怎么确定多了多少种情况呢，假如给我们的数字是 8，二进制为 1000，我们首先按长度为 4 算出所有情况，共 8 种。仔细观察我们十进制转为二进制字符串的写法，发现转换结果跟真实的二进制数翻转了一下，所以我们的  $t$  为“0001”，那么我们从倒数第二位开始往前遍历，到  $i=1$  时，发现有两个连续的 0 出现，那么  $i=1$  这个位置上能出现 1 的次数，就到  $one$  数组中去找，那么我们减去 1，减去的就是 0101 这种情况，再往前遍历， $i=0$  时，又发现两个连续 0，那么  $i=0$  这个位置上能出 1 的次数也到  $one$  数组中去找，我们再减去 1，减去的是 1001 这种情况

```

public int findIntegers(int n) {
    int cnt = 0;
    String t = "";
    while (n > 0) {
        ++cnt;
        t += (n & 1) == 1 ? "1" : "0"; // 这里把 t 倒过来了
        n >>= 1;
    }
    char[] s = t.toCharArray();
    int[] one = new int[cnt], zero = new int[cnt];
    one[0] = zero[0] = 1;
    for (int i = 1; i < cnt; i++) {
        zero[i] = zero[i-1] + one[i-1];
        one[i] = zero[i-1];
    }
    int ans = zero[cnt-1] + one[cnt-1]; // 长度为 cnt 的所有不含重复 1 的数字的个数，但数多了
    for (int i = cnt-2; i >= 0; i--) {
        if (s[i] == '1' && s[i+1] == '1') break;
        if (s[i] == '0' && s[i+1] == '0') ans -= one[i];
    }
    return ans;
}

```

#### 2. 解题思路与分析

其实长度为  $k$  的二进制数字符串没有连续的 1 的个数是一个斐波那契数列  $f(k)$ 。比如当  $k=5$  时，二进制数的范围是 00000-11111，我们可以将其分为两个部分，00000-01111 和 10000-10111，因为任何大于 11000 的数字都是不成立的，因为有开头已经有了两个连续 1。而我们发现其实 00000-01111 就是  $f(4)$ ，而 10000-10111 就是  $f(3)$ ，所以  $f(5) = f(4) + f(3)$ ，这就是一个斐波那契数列啦。那么我们要做的首先就是建立一个这个数组，方便之后直接查值。我们从给定数字的最高位开始遍历，如果某一位是 1，后面有  $k$  位，就加上  $f(k)$ ，因为如果我们把当前位变成 0，那么后面  $k$  位就可以直接从斐波那契数列中取值了。然后标记  $pre$  为 1，再往下遍历，如果遇到 0 位，则  $pre$  标记为 0。如果当前位是 1， $pre$  也是 1，那么直接返回结果。最后循环退出后我们要加上数字本身这种情况

```

public int findIntegers(int n) {
    int k = 31, pre = 0, ans = 0;
    int[] dp = new int[32];
    dp[0] = 1;
    dp[1] = 2;
    for (int i = 2; i < 32; i++)
        dp[i] = dp[i-1] + dp[i-2];
    while (k >= 0) {
        if ((n & (1 << k)) > 0) {
            ans += dp[k];
            if (pre == 1) return ans;
            pre = 1;
        } else pre = 0;
        k--;
    }
}

```

```

    }
    return ans + 1;
}

```

### 1.3.13 514. Freedom Trail - Hard

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

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

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

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

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

1. 解题思路分析: dfs + 记忆数组 (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)

- 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 findRotateSteps(String ring, String key) { // dfs
    m = ring.length();
    n = key.length();
    char[] s = ring.toCharArray();
    for (int i = 0; i < m; i++) {
        if (key.indexOf(s[i]) == -1) continue; // 只记录在 key 中出现过的字母的位置
        idx.computeIfAbsent(s[i], z -> new ArrayList<>()).add(i);
    }
    dp = new int[m][n];
    return dfs(0, 0, ring, key);
}

Map<Character, List<Integer>> idx = new HashMap<>();
int[][] dp;
int m, n;

private int dfs(int i, int j, String s, String t) {
    if (j == n) return 0;
    if (dp[i][j] > 0) return dp[i][j];
    int ans = Integer.MAX_VALUE;
    for (Integer v : idx.get(t.charAt(j)))
        ans = Math.min(ans, dfs(v, j+1, s, t) + getDist(v, i) + 1); // + 1 for confirm push
    return dp[i][j] = ans;
}

private int getDist(int i, int j) {
    int min = Math.min(i, j), max = Math.max(i, j);
    return Math.min(Math.abs(i - j), Math.abs(m - max + min));
}

```

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<sup>i-1</sup> 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) { // dfs
    int m = key.length(), n = ring.length();
    char [] s = key.toCharArray();
    char [] t = ring.toCharArray();
    int [][] dp = new int [m+1][n];
    int dif = 0, cnt = 0;
    for (int i = m-1; i >= 0; i--)
        for (int j = 0; j < n; j++) { // j 是固定在 12 点钟表盘的位置
            dp[i][j] = Integer.MAX_VALUE;
            for (int k = 0; k < n; k++)
                if (s[i] == t[k]) {
                    dif = Math.abs(j - k);
                    cnt = Math.min(dif, n - dif);
                    dp[i][j] = Math.min(dp[i][j], cnt + dp[i+1][k]);
                }
        }
    return dp[0][0] + m;
}
```

### 3. 解题思路分析: dfs + 记忆数组 todo: 这个写得太繁琐了, 没看

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);
}

```

### 1.3.14 78. Subsets - Medium 典型 subset 题目 new ArrayList<>(){add(v);}

Given an integer array nums of unique elements, return all possible subsets (the power set).

The solution set must not contain duplicate subsets. Return the solution in any order.

#### 1. 解题思路与分析: Bitmasking

```

public List<List<Integer>> subsets(int[] a) { // 这种应该是最简单最快的了
    int n = a.length, r = (1 << n);
    List<List<Integer>> ll = new ArrayList<>();
    for (int i = 0; i < r; i++) {
        List<Integer> l = new ArrayList<>();
        if (i == 0) {
            ll.add(l);
            continue;
        }
        for (int j = 0; j < n; j++)
            if (((i >> j) & 1) == 1) l.add(a[j]);
        ll.add(l);
    }
    return ll;
}

```

#### 2. 解题思路与分析: Cascading

```

public List<List<Integer>> subsets(int[] a) { // 在后面的动态规划中也会常用到，还是需要熟悉一下
    int n = a.length, r = (1 << n);
    List<List<Integer>> ll = new ArrayList<>();
    ll.add(new ArrayList<>());
    for (int v : a) {
        List<List<Integer>> tmp = new ArrayList<>();
        for (List<Integer> l : ll)
            tmp.add(new ArrayList<>(l) {{ add(v); }}); // 这种写法，见过一次，上次找了半天也没能找出来
        for (List<Integer> l : tmp)
            ll.add(l);
    }
    return ll;
}

```

#### 3. 解题思路与分析: backtracking

```

public List<List<Integer>> subsets(int[] a) { // 这个最原始本能的、回溯的写法被我忘记了。。。。。。
    n = a.length;
    vis = new boolean [n];
    getASubset(new ArrayList<>(), 0, a);
    return ll;
}
List<List<Integer>> ll = new ArrayList<>();
boolean [] vis;
int n;
void getASubset(List<Integer> l, int idx, int [] a) {
    if (idx == n) {
        ll.add(new ArrayList<>(l));
        return ;
    }
    getASubset(l, idx+1, a);
    l.add(a[idx]);
    getASubset(l, idx+1, a);
    l.remove(l.size()-1);
}

```

### 1.3.15 368. Largest Divisible Subset - Medium 要求返回序列的接龙型

Given a set of distinct positive integers nums, return the largest subset answer such that every pair (answer[i], answer[j]) of elements in this subset satisfies:

$\text{answer}[i] \% \text{answer}[j] = 0$ , or  $\text{answer}[j] \% \text{answer}[i] = 0$  If there are multiple solutions, return any of them.

#### 1. 解题思路与分析: 万能的 dfs 记忆化搜索

```

public List<Integer> largestDivisibleSubset(int[] a) { // 出来一个新题：您是认不是它是 dfs 记忆化搜索，岂不是很悲催？
    Arrays.sort(a);
    n = a.length;
    dp = new List [n];
    for (int i = 0; i < n; i++) {

```

```

        List<Integer> cur = dfs(a, i, 1);
        if (cur.size() > ans.size())
            ans = cur;
    }
    return ans;
}
List<Integer> ans = new ArrayList<>();
List<Integer> [] dp;
int n;
// pre: dfs 路径中前一节点
// 返回值: 当前节点的倍数数组 (包含当前节点)
private List<Integer> dfs(int [] a, int i, int pre) {
    int cur = a[i];
    if (cur % pre != 0) return new ArrayList<>();
    if (dp[i] != null) return dp[i]; // 记忆数组存在当前结果的话直接返回, 避免重复计算
    List<Integer> maxList = new ArrayList<>();
    for (int j = i+1; j < n; j++) {
        List<Integer> tmp = dfs(a, j, cur);
        if (tmp.size() > maxList.size())
            maxList = new ArrayList<>(tmp);
        // maxList = tmp; // bug:
    }
    maxList.add(cur); // 将当前节点加入到集合中, 有点儿回溯的样子, 把路径中的数字反序加了回来
    dp[i] = maxList; // 将当前结果存入记忆数组
    return maxList;
}

```

## 2. 解题思路与分析: 接龙型动态规划

属于接龙型动态规划, 类似于 Longest Increasing Subsequence 的做法。这个题不同之处有两点:

需要记录具体方案  $O(N^2)$  的时间复杂度不能通过测试

具体的优化在于找上一个接龙的数的时候, 不是 for 循环所有比他小的数, 而是直接 for 循环他的因子。而获取因子可以用  $O(n)$  的时间做到。

```

public List<Integer> largestDivisibleSubset(int[] a) { // todo: 感觉想得还不是很透
    if (a == null || a.length == 0) return new ArrayList();
    Arrays.sort(a);
    int n = a.length;
    HashMap<Integer, Integer> dp = new HashMap();
    HashMap<Integer, Integer> pre = new HashMap();
    for (int i = 0; i < n; i++) {
        dp.put(a[i], 1);
        pre.put(a[i], -1);
    }
    int lastNum = a[0];
    for (int i = 0; i < n; i++) {
        int num = a[i];
        for (Integer factor : getFactors(num)) {
            if (!dp.containsKey(factor)) continue;
            if (dp.get(num) < dp.get(factor) + 1) {
                dp.put(num, dp.get(factor) + 1);
                pre.put(num, factor);
            }
        }
        if (dp.get(num) > dp.get(lastNum))
            lastNum = num;
    }
    return getPath(pre, lastNum);
}
private List<Integer> getPath(HashMap<Integer, Integer> pre, int lastNum) {
    List<Integer> path = new ArrayList();
    while (lastNum != -1) {
        path.add(lastNum);
        lastNum = pre.get(lastNum);
    }
    Collections.reverse(path);
    return path;
}
private List<Integer> getFactors(int num) {
    List<Integer> factors = new ArrayList();
    if (num == 1) return factors;
    int factor = 1;
    while (factor * factor <= num) {
        if (num % factor == 0) {
            factors.add(factor);
            if (factor != 1 && num / factor != factor) // 这里会把所有质因子的倍数也加进去, 怕数组中不存在最小质因子, 而是存在其某个倍数?
                factors.add(num / factor);
        }
        factor++;
    }
    return factors;
}

```

## 3. 解题思路与分析: 一个极为简洁的 dp

这里有个很简单的数学性质，就是整除的传递性，如果  $a \% b == 0$  且  $b \% c = 0$ ，那么  $a \% c = 0$ ，说白了如果  $c$  是  $b$  的因子， $b$  又是  $a$  的因子，那么  $c$  肯定是  $a$  的因子。这样我们就可以在数组中找出很多整除链 ( $a \rightarrow b \rightarrow c \rightarrow d$ ，其中  $b$  是  $a$  的因子， $c$  是  $b$  的因子， $d$  是  $c$  的因子)，这样的链条就满足两两整除的条件，题目就变成了求最长的链条。

```
public List<Integer> largestDivisibleSubset(int[] nums) {
    int n = nums.length;
    List<Integer> ans = new ArrayList<Integer>();
    if (n < 2) {
        if (n == 0) return ans;
        ans.add(nums[0]);
        return ans;
    }
    Arrays.sort(nums);
    int[] preFactIdx = new int[n];
    int[] factCnt = new int[n];
    int maxlen = 0;
    int maxnum = 0;
    for (int i = n-1; i >= 0; i--) {
        for (int j = i; j < n; j++) {
            if (nums[j] % nums[i] == 0 && factCnt[i] < factCnt[j]+1) {
                factCnt[i] = factCnt[j]+1;
                preFactIdx[i] = j;
                if (factCnt[i] > maxlen) {
                    maxlen = factCnt[i];
                    maxnum = i;
                }
            }
        }
        for (int i = 0; i < maxlen; i++) {
            ans.add(nums[maxnum]);
            maxnum = preFactIdx[maxnum];
        }
    }
    return ans;
}
```

首先我先对  $\text{nums}$  排序，这里我用了两个数组  $\text{prefactors}[]$  和  $\text{factorcount}[]$ ， $\text{prefactors}[i]$  里其实保存的是以  $\text{nums}[i]$  未结尾的整除链前面一个数的下标， $\text{factorcount}[i]$  存的是以  $\text{nums}[i]$  结尾的整除链长度。这里我们就可以用动态规划的方式求出  $\text{factorcount}[i]$  的值了，取最大的一个，然后再根据  $\text{prefactors}[i]$  推算出整除链中所有的元素。

这里开两个脑洞，发散下思维。

- 脑洞 1：其实所有整除链可以合并为一个多叉整除树，这里得增加一个额外的根节点，使得根节点可以被  $\text{nums}$  中任何一个数整除。这个整除树有个很重要的性质——除根节点以为任意节点可以整除其父节点。这道题就会演变成求整除树中最深的路径。对于树的问题，我们往往能用递归的方式解决，代码也会变得比较好理解。
- 脑洞 2：看下代码，是不是很像 01 背包，其实我觉得这到题可以认为是带限制条件的 01 背包，限制条件可以简化为放入背包的数必须能整除背包中最小的一个数，背包为无穷大，每个数的价值为 1。

### 1.3.16 673. Number of Longest Increasing Subsequence

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

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

### 1.3.17 1987. Number of Unique Good Subsequences - Hard

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

Find the number of unique good subsequences of binary.

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

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

这里的思路是：扫出从左起第一个 1 所在的位置，从左到右遍历，对于当前字符来说就是 **用当前字符或 不用当前字符** 两种选择，然后会有重复的部分，需要把重复的部分减掉

小写字母a的ascii编码值为97, 小写字母z的ascii编码值为122,  
将数字97表示为小写字母a, 那么每一个字母都可以由97+i来表示, i=0~25  
例如 b --> 97+1  
所以, `dp[i]`的下标i就能用来表示字母97+i, 这样只需要26个空间即可表示所有的26个字母  
然后`dp[i]`, 这里表示的是以97+i结尾的子序列的个数  
下面是对字符串S, 从左到右进行遍历  
这样的遍历, 有这样的特点, 以字符串"abcsade"为例子来描述  
当遍历到字符s的时候, 显然前面的字符"abc"都已经遍历了一次,  
注意! 子问题出现:  
假设前面的"abc", 已经找出了所有的不重复的子序列:a,b,c,ab,ac,bc  
那么当遍历到's'时, 这个's'可以添加到所有上述子序列的末尾, 构成新的子序列  
由于原字序列不重复, 显然在其所有子序列后添加一个字符's'得到的新的一系列子序列,  
依然是不重复的。  
所以能发现, 以's'结尾的新的子序列实际就是上述这些末尾添加了's'的子序列  
但是考虑到's'字符可以单独作为一个子序列, 因此还需要加上1, 表示这一特例  
那么总结上述推断, 假设`dp[i]`表示当前遍历到字符97+i时, 以此字符结尾的子序列的个数  
`dp[i] = 1+dp[0]+dp[1]+dp[2]+...+dp[i-1] = 1+sum(dp)`

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

## Intuition

An easier or harder version of 940. Distinct Subsequences II

## Explanation

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

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

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

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

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

The final result is `ends1 + ends2 + has0`.

## Complexity

Time  $O(n)$

Space  $O(1)$

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

- 还有一个没有看懂的

- <https://leetcode-cn.com/problems/number-of-unique-good-subsequences/solution/ju-yi-fan-san-by-average-1/>
- <https://leetcode-cn.com/problems/distinct-subsequences-ii/solution/dong-tai-gui-hua-cong-fen-xi-dai-ma/>

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

### 1.3.18 847. 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[i]` 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;
}
```

### 1.3.19 926. Flip String to Monotone Increasing - Medium

A binary string is monotone increasing if it consists of some number of 0's (possibly none), followed by some number of 1's (also possibly none).

You are given a binary string  $s$ . You can flip  $s[i]$  changing it from 0 to 1 or from 1 to 0.

Return the minimum number of flips to make  $s$  monotone increasing.

#### 1. 解题思路与分析：动态规划

这道题给了我们一个只有 0 和 1 的字符串，现在说是可以将任意位置的数翻转，即 0 变为 1，或者 1 变为 0，让组成一个单调递增的序列，即 0 必须都在 1 的前面，博主刚开始想的策略比较直接，就是使用双指针分别指向开头和结尾，开头的指针先跳过连续的 0，末尾的指针向前跳过连续的 1，然后在中间的位置分别记录 0 和 1 的个数，返回其中较小的那个。这种思路乍看上去没什么问题，但是实际上是有问题的，比如对于这个例子"1001111110010111011"，如果按这种思路的话，就应该将所有的 0 变为 1，从而返回 6，但实际上更优的解法是将第一个 1 变为 0，将后 4 个 0 变为 1 即可，最终可以返回 5。

这说明了之前的解法是不正确的。这道题可以用动态规划 Dynamic Programming 来做，需要使用两个 dp 数组，其中  $cnt1[i]$  表示将范围是  $[0, i-1]$  的子串内最小的将 1 转为 0 的个数，从而形成单调字符串。同理， $cnt0[j]$  表示将范围是  $[j, n-1]$  的子串内最小的将 0 转为 1 的个数，从而形成单调字符串。这样最终在某个位置使得  $cnt0[i]+cnt1[i]$  最小的时候，就是变为单调串的最优解，这样就可以完美的解决上面的例子，子串"100"的最优解是将 1 变为 0，而后面的"11111110010111011"的最优解是将 4 个 0 变为 1，总共加起来就是 5，参见代码如下：

```
// 可以用动态规划 Dynamic Programming 来做，需要使用两个 dp 数组，其中 cnt1[i] 表示将范围是 [0, i-1] 的子串内最小的将 1 转为 0 的个数，从而形成单调字符串。
// 同理，cnt0[j] 表示将范围是 [j, n-1] 的子串内最小的将 0 转为 1 的个数，从而形成单调字符串。
// 这样最终在某个位置使得 cnt0[i]+cnt1[i] 最小的时候，就是变为单调串的最优解，
public int minFlipsMonoIncr(String t) { // bug
    int n = t.length(), ans = Integer.MAX_VALUE;
    char [] s = t.toCharArray();
    int [] l = new int [n+1], r = new int [n+1];
    for (int i = 1, j = n-1; i <= n; i++, -j) {
        l[i] = l[i-1] + (s[i-1] == '0' ? 0 : 1); // cnt left --> right i--> 0
        r[j] = r[j+1] + (s[j] == '0' ? 1 : 0);
    }
    for (int i = 0; i <= n; i++)
        ans = Math.min(l[i] + r[i], ans);
    return ans;
}
```

#### 2. 解题思路与分析：空间压缩与简化

我们可以进一步优化一下空间复杂度，用一个变量  $cnt1$  来记录当前位置时 1 出现的次数，同时  $res$  表示使到当前位置的子串变为单调串的翻转次数，用来记录 0 的个数，因为遇到 0 就翻 1 一定可以组成单调串，但不一定是最优解，每次都要和  $cnt1$  比较以下，若  $cnt1$  较小，就将  $res$  更新为  $cnt1$ ，此时保证了到当前位置的子串变为单调串的翻转次数最少，并不关心到底是把 0 变为 1，还是 1 变为 0 了，其实核心思想跟上面的解法很相近

```
// 用一个变量 cnt1 来记录当前位置时 1 出现的次数，同时 res 表示使到当前位置的子串变为单调串的翻转次数，
// 用来记录 0 的个数，因为遇到 0 就翻 1 一定可以组成单调串，但不一定是最优解，
// 每次都要和 cnt1 比较以下，若 cnt1 较小，就将 res 更新为 cnt1，
```

```
// 此时保证了到当前位置的子串变为单调串的翻转次数最少，并不关心到底是把 0 变为 1，还是 1 变为 0 了
public int minFlipsMonoIncr(String t) { // bug
    int n = t.length(), res = 0, cntOne = 0;
    char [] s = t.toCharArray();
    for (int i = 0; i < n; i++) {
        if (s[i] == '0') res++;
        else cntOne++;
        res = Math.min(res, cntOne);
    }
    return res;
}
```

早上的时候不是刚写过一个这样的题吗？与 1493 的第二种解法有什么区别？

### 1.3.20 313. 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^I
    }
    return ans[n-1]; ^I^I^I
}
```

### 1.3.21 backpack III

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

```

        dp[i] = Math.max(dp[i], dp[i-A[j]] + V[j]);
    }
}
return dp[m];
}

```

### 1.3.22 879. Profitable Schemes - Hard 0-1 背包问题

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

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

Return the number of schemes that can be chosen. Since the answer may be very large, return it modulo  $10^9 + 7$ .

#### 1. 解题思路与分析: DP

**Solution: DP**

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

state: (`current_profit`, `people_used`)

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

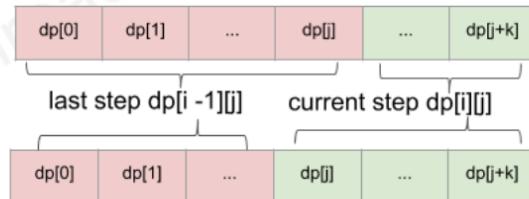
Init: `dp[0][0][0] = 1`  
 Transition:  
`p, g = profit[k - 1], group[k - 1]`  
`dp[k][i][j] = dp[k-1][i][j]`  
`+ dp[k-1][i-p][j-g] if j >= g else 0`  
`0 <= i <= P, 0 <= j <= G`

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

Time complexity:  $O(KPG)$   
 Space complexity:  $O(KPG) \rightarrow O(PG)$

Dimensionality reduction  
 If only  $\text{dp}[i-1][*]$  is needed for  $\text{dp}[i][*]$

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



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

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

现在来推导状态转移方程，整个规划的核心是买卖，总共买卖的个数是固定的，每多干一票买卖，可能的分配方法就可能增加，但不可能减少的，因为假如当前已经算出来做  $k-1$  次买卖的分配方法总数，再做一次买卖，之前的分配方法不会减少，顶多是人数不够，做不成当前这票买卖而已，所以我们的  $\text{dp}[k][i][j]$  可以先更新为  $\text{dp}[k-1][i][j]$ ，然后再来看这第  $k$  个买卖还能不能做，我们知道假设这第  $k$  个买卖需要  $g$  个人，能获得利润  $p$ ，只有当我们现在的人数  $i$  大于等于  $g$  的时候，才有可能做这个任务，我们要用  $g$  个人来做任务  $k$  的话，那么其余的  $k-1$  个任务只能由  $i-g$  个人来做，而且由于整个需要产生利润  $j$ ，第  $k$  个任务能产生利润  $p$ ，所以其余的  $k-1$  个任务需要产生利润  $j-p$ ，由于利润不能是负值，所以我们还需要跟 0 比较，取二者的最大值。

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

```

static final int mod = (int)1e9 + 7;
public int profitableSchemes(int n, int minProfit, int[] group, int[] profit) {
    int m = group.length, ans = 0;
    int [[[dp = new int [m+1][n+1][minProfit+1];
    dp[0][0][0] = 1;
    for (int i = 1; i <= m; i++) { // 遍历每桩案件
        int g = group[i-1], p = profit[i-1];
        for (int j = 0; j <= n; j++) // 遍历人数
            for (int k = 0; k <= minProfit; k++) {
                dp[i][j][k] = dp[i-1][j][k];
                if (j >= g) // 在本桩案件人手够用的前提下，办与不办此案的方案数相加
                    dp[i][j][k] += dp[i-1][j-g][k-p];
    }
    }
}

```

```

        dp[i][j][k] = (dp[i][j][k] + dp[i-1][j-g][Math.max(0, k-p)]) % mod;
    }
}
for (int i = 0; i <= n; i++)
    ans = (ans + dp[m][i][minProfit]) % mod;
return ans;
}

```

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

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

```

static final int mod = (int)1e9 + 7;
public int profitableSchemes(int n, int minProfit, int[] group, int[] profit) {
    int m = group.length, ans = 0;
    int [][] dp = new int [n+1][minProfit+1];
    dp[0][0] = 1;
    for (int i = 1; i <= m; i++) { // 遍历每桩案件
        int g = group[i-1], p = profit[i-1];
        for (int j = n; j >= g; j--) // 当压缩了一维空间, 从后往前遍历以免产生脏数据
            for (int k = minProfit; k >= 0; k--)
                dp[j][k] = (dp[j][k] + dp[j-g][Math.max(0, k-p)]) % mod;
    }
    for (int i = 0; i <= n; i++)
        ans = (ans + dp[i][minProfit]) % mod;
    return ans;
}

```

## 3. 递归 dfs + 记忆数组

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

```

public int profitableSchemes(int n, int minProfit, int[] group, int[] profit) {
    m = group.length;
    this.n = n;
    this.minProfit = minProfit;
    dp = new int [m+1][n+1][minProfit+1];
    for (int i = 0; i <= m; i++)
        for (int j = 0; j <= n; j++)
            Arrays.fill(dp[i][j], Integer.MIN_VALUE);
    return dfs(m, n, minProfit, group, profit);
}
static final int mod = (int)1e9 + 7;
int m, n, minProfit;
int [[[] dp;
private int dfs(int i, int j, int k, int [] group, int [] profit) {
    if (i == 0) return k <= 0 ? 1 : 0;
    if (k < 0) k = 0;
    if (dp[i][j][k] != Integer.MIN_VALUE) return dp[i][j][k];
    int g = group[i-1], p = profit[i-1];
    int ans = dfs(i-1, j, k, group, profit);
    if (j >= g)
        ans = (ans + dfs(i-1, j-g, Math.max(0, k-p), group, profit)) % mod;
    return dp[i][j][k] = ans;
}

```

## 1.3.23 377. 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.

### 1. 解题思路与分析: dfs 记忆化搜索

思路: 一开始想到用 DFS 来做, 但是有个问题就是这种方式得到的答案各个数字排列是无序的, 也就是 1, 3 和 3, 1 这种只是一个答案,

```

public int combinationSum4(int[] a, int target) { // 带记忆化的来搜索, 用数组还是用图, 可能也看数据规模吧
    return helper(a, target);
}
Map<Integer, Integer> dp = new HashMap<>();
private int helper(int[] a, int target) {
    if (target < 0) return 0;
    if (target == 0) return 1;
    if (dp.containsKey(target)) return dp.get(target);
    int ans = 0;
    for (int num : a)
        ans += helper(a, target - num);
    dp.put(target, ans);
    return ans;
}

```

```

int ans = 0;
for (int i = 0; i < a.length; i++) {
    ans += helper(a, target-a[i]);
    dp.put(target, ans);
    return ans;
}

```

## 2. 解题思路与分析: DP

然后又想把数字保存起来, 在得到一个答案的时候对这些数字再求一次总共排列的个数, 这种方式还有问题就是在求总排列个数的时候比如 2, 1, 1 三个加一起等于 4, 总的排列个数即为  $(3! / 2!)$ , 但是当数字个数很多的时候阶乘太大, 根本无法计算.

然后就想到可以用动态规划来做, 也是一个背包问题, 求出  $[1, \text{target}]$  之间每个位置有多少种排列方式, 这样将问题分化为子问题. 状态转移方程可以得到为:

$$\text{dp}[i] = \sum(\text{dp}[i - \text{nums}[j]]), (\text{i} - \text{nums}[j] > 0);$$

如果允许有负数的话就必须要限制每个数能用的次数了, 不然的话就会得到无限大的排列方式, 比如 1, -1, target = 1;

```

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];
}

```

- 针对适当情况的优化: 如果 target 远大于 nums 数组的个数的话, 上面的算法可以做适当的优化, 先给 nums 数组排个序, 然后从 1 遍历到 target, 对于 i 小于数组中的数字 x 时, 直接 break 掉, 因为后面的数更大, 其余地方不变

```

public int combinationSum4(int[] a, int target) { // if target 远大于 数组元素个数
    int n = a.length;
    Arrays.sort(a);
    int [] dp = new int [target + 1];
    dp[0] = 1;
    for (int i = 1; i <= target; i++)
        for (Integer v : a) {
            if (i < v) break;
            dp[i] += dp[i-v];
        }
    return dp[target];
}

```

## 1.3.24 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  $x \leq y$ . The result of this smash is: If  $x = y$ , both stones are destroyed, and If  $x \neq y$ , the stone of weight x is destroyed, and the stone of weight y has new weight  $y - x$ . At the end of the game, there is at most one stone left. Return the smallest possible weight of the left stone. If there are no stones left, return 0.

```

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

```

## 1.3.25 810. Chalkboard XOR Game - Hard

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

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

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

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

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

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

Case 3- XOR!=0 and nums.length is odd:

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

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

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

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

```

for (int i = 0; i < n; i++)
    xor[i] = (i == 0 ? 0 : xor[i-1]) ^ arr[i];
return helper(arr, 0, xor[n-1]); // i: turn
}

```

### 1.3.26 1449. Form Largest Integer With Digits That Add up to Target

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

```

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

```

### 1.3.27 516. Longest Palindromic Subsequence

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

```

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

```

### 1.3.28 1143. Longest Common Subsequence

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

```

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

```

### 1.3.29 1092. Shortest Common Supersequence - Hard

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

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

- 参考的标准答案:

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

- 自己写的

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

```

        j++;
    }
    return res + s.substring(i) + t.substring(j);
}

```

### 1.3.30 546. Remove Boxes - Hard

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

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

Return the maximum points you can get.

```

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

```

### 1.3.31 1531. String Compression II - Hard

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

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

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

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

```

public int getLengthOfOptimalCompression(String t, int k) {
    n = t.length();
    s = t.toCharArray();
    dp = new Integer[n][n - k + 1]; // int [n][n-k+1] 而不是 [n][k+1]
    return dfs(0, n - k);
}
Integer[][] dp;
char[] s;
int n;
private int dfs(int idx, int k) { // 求从下标 i 开始向后，所有长度为 k 的子序列中，编码后的最小长度
    if (k == 0) return 0; // 当下标越界时还未找到长度为 k 的子序列
    if (idx == n) return Integer.MAX_VALUE;
    if (dp[idx][k] != null) return dp[idx][k];
    int ans = Integer.MAX_VALUE, cnt = 0;
    boolean[] vis = new boolean[26];
    for (int i = idx; i < n; i++) {
        if (vis[s[i] - 'a']) continue; // 优化：当前字母是已处理过的字母，遍历 26 个字母的可能性，重复的跳过
        if (idx > 0 && s[i] == s[idx - 1]) continue; // 另一种重复的处理
        cnt = 0;
        for (int j = i; j < n; j++) {
            if (s[j] != s[i]) continue;
            cnt++; // 数左半片段中，与 s[i] 相同的字母的个数
            if (k - cnt < 0) break; // 如果左半部分长度大于子序列长度，退出
            int rite = dfs(j + 1, k - cnt);
            if (rite == Integer.MAX_VALUE) continue;
            if (left == ("").length());
            int left = ("." + cnt).length();
            ans = Math.min(ans, left + rite + (left == 1 && cnt == 1 ? 0 : 1));
        }
    }
    return dp[idx][k] = ans;
}

```

#### 1. 解题思路与分析: 动态规划

```

public int getLengthOfOptimalCompression(String s, int k) { // 与上面的思路差不多, 这里自顶向下, 上面的 dfs 自底向上
    int n = s.length();
    int [][] f = new int[n+1][k+1];
    for (int i = 0; i <= n; i++) {
        Arrays.fill(f[i], Integer.MAX_VALUE >> 1);
    }
    f[0][0] = 0;
    for (int i = 1; i <= n; ++i) {
        for (int j = 0; j <= k && j <= i; ++j) {
            if (j > 0) // 先初始化为删除当前字符为删除的第 j 个字符的情况
                f[i][j] = f[i - 1][j - 1];
            int same = 0, diff = 0;
            for (int x = i; x >= 1 && diff <= j; x--) {
                if (s.charAt(x-1) == s.charAt(i-1)) {
                    same++;
                    f[i][j] = Math.min(f[i][j], f[x-1][j - diff] + calc(same));
                } else diff++;
            }
        }
    }
    return f[n][k];
}
public int calc(int x) {
    if (x == 1) return 1;
    if (x < 10) return 2;
    if (x < 100) return 3;
    return 4;
}

```

## 2. 决策类 DP 总结

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

### 1.3.32 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.

```

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

```

### 1.3.33 1478. Allocate Mailboxes - Hard

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

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

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

解题思路分析:

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

- 当只有1栋房屋, 1个邮箱时, 显然将邮箱放在房屋处最为合理, 这时邮箱与房屋的距离为0。
- 当有2栋房屋, 1个邮箱时, 比如房屋1在坐标0处, 房屋2在坐标10处, 此时如果将邮箱放在坐标0的话, 它与两栋房屋的距离和为10。放在坐标10的情况下距离和也为10。另外我们可以看出, 不论邮箱放在两栋房屋之间的任意位置上, 它与房屋的距离和都是10。因此通过此例可得出, 中位数的说法虽然正确, 但并不全面, 不过这不影响本题解题, 对于本题, 我们统一将邮箱安排在房屋位置上是为了方便计算, 因此才得出中位数的说法 (本例房屋1和2都可以看做是中位数)。
- 当有3栋房屋, 1个邮箱时, 此时通过上面的例子可知, 对于两侧的房子, 将邮箱放在他们之间的任何位置对于结果没有任何影响, 距

离和都是两栋房子间的距离。但邮箱的位置会对中间的房子产生影响，因此，将其放置在中间房子的坐标上最为合理，这样邮箱与中间房屋的距离为0，可使得全局总距离最小。而中间的房屋正是3个房屋的中位数。  
 4. 当有4栋房屋，1个邮箱时，与上例同理，对于两侧的房子，将邮箱放在他们之间的任何位置对于结果没有任何影响，因此邮箱可以考虑放在中间两个房屋的任何一个位置上。另外对于中间两个房屋，不论邮箱放置在其任何一个位置上，对于总距离都不会产生影响  
 (这相当于第2条)。

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

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

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

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

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

```

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

private int dfs(int[] arr, int idx, int k) { // idx: 待分割大子区间的起始坐标; k: 待分割成的子区间的个数
    if (idx == n || idx == n-k) return 0;
    if (dp[idx][k] > 0) return dp[idx][k];
    if (k == 1) return dp[idx][k] = getDist(arr, idx, n-1);
    int res = Integer.MAX_VALUE;
    for (int i = idx; i < n-(k-1); i++)
        res = Math.min(res, getDist(arr, idx, i) + dfs(arr, i+1, k-1));
    return dp[idx][k] = res;
}

int[][] dp;
int[][] dist; // 这也是一种记忆数组优化
int n;
public int minDistance(int[] houses, int k) {
    n = houses.length;
    dist = new int[n][n];
    dp = new int[n][k+1];
    Arrays.sort(houses);
    return dfs(houses, 0, k);
}

```

## 1. 解题思路与分析：动态规划

- 复杂度分析

时间复杂度： $O(n^2k)$ ，其中n是数组houses的长度。在预处理部分，需要的时间为 $O(n^2)$ ；在动态规划部分，我们需要 $O(nk)$ 的时间枚举每个状态 $f[i][j]$ ，并使用 $O(n)$ 的时间枚举进行状态转移，相乘即可得到总时间复杂度。

空间复杂度： $O(n^2 + nk)$ ，即为存储cost以及动态规划的状态需要的空间。

```

public int minDistance(int[] h, int k) {
    int n = h.length;
    Arrays.sort(h);
    int[][] medsum = new int[n][n];
    for (int i = n - 2; i >= 0; --i)
        for (int j = i + 1; j < n; ++j)
            medsum[i][j] = medsum[i + 1][j - 1] + h[j] - h[i];
    int[][] f = new int[n][k + 1];

```

<sup>5</sup>DEFINITION NOT FOUND.

```

for (int i = 0; i < n; ++i)
    Arrays.fill(f[i], Integer.MAX_VALUE / 2);
for (int i = 0; i < n; ++i) {
    f[i][1] = medsum[0][i];
    for (int j = 2; j <= k && j <= i + 1; ++j)
        for (int x = 0; x < i; x++)
            f[i][j] = Math.min(f[i][j], f[x][j-1] + medsum[x+1][i]);
}
return f[n - 1][k];
}

```

### 1.3.34 1771. Maximize Palindrome Length From Subsequences

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

```

public int longestPalindrome(String ss, String tt) { // 比较喜欢 2 维 dp, 比较直接直观
    int m = ss.length(), n = m + tt.length(), ans = 0;
    char [] s = (ss + tt).toCharArray(); // 先求这个联接合并字符串的最长回文子序列
    int [][] dp = new int [n][n];
    for (int i = n-2; i >= 0; i--) {
        dp[i][i] = 1;
        for (int j = i+1; j < n; j++) {
            if (s[i] == s[j]) {
                dp[i][j] = dp[i+1][j-1] + 2;
                if (i < m && j >= m) // 确认来自于两个不同的字符串
                    ans = Math.max(ans, dp[i][j]);
            } else dp[i][j] = Math.max(dp[i+1][j], dp[i][j-1]);
        }
    }
    return ans;
}

```

- 下面是降了一维空间的

```

public int longestPalindrome(String ss, String tt) {
    int m = ss.length(), n = m + tt.length(), ans = 0;
    char [] s = (ss + tt).toCharArray(); // 先求这个联接合并字符串的最长回文子序列
    int [] dp = new int[n];
    Arrays.fill(dp, 1);
    int max = 0;
    for (int i = n - 1; i >= 0; i--) {
        int curMax = 0;
        for (int j = i + 1; j < n; j++) {
            int mem = dp[j]; // remember prev dp[j] val
            // curMax = Math.max(curMax, dp[j]); // bug: curMax 不可以提前更新
            if (s[i] == s[j]) {
                dp[j] = curMax + 2; // 要用更新前的值
                if (i < m && j >= m)
                    max = Math.max(max, dp[j]);
            }
            curMax = Math.max(mem, curMax);
        }
    }
    return max;
}

```

### 1.3.35 1896. Minimum Cost to Change the Final Value of Expression - Hard

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

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

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

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

```

private int [] getMinOperations(int va, int vb, int ca, int cb, char sign) {
    if (sign == '&') {
        if (va == 1 && vb == 1) // 1&1, 将其中一个 1 反转为 0
            return new int [] {1, Math.min(ca, cb)};
    }
}

```

```

    else if (va == 0 && vb == 0) // 0&0, 将其中一个 0 反转为 1, 并将 & 反转为 |
        return new int [] {0, Math.min(ca, cb) + 1};
    else return new int [] {0, 1}; // 1&0, 将 & 反转为 1
} else {
    if (va == 1 && vb == 1)      // 1|1, 将其中一个 1 反转为 0, 并将 | 反转为 &
        return new int [] {1, Math.min(ca, cb) + 1};
    else if (va == 0 && vb == 0) // 0|0, 将其中一个 0 反转为 1
        return new int [] {0, Math.min(ca, cb)};
    else return new int [] {1, 1}; // 1|0, 将 | 反转为 &
}
}

public int minOperationsToFlip(String expression) {
    Stack<Integer> res = new Stack<>();
    Stack<Character> sgn = new Stack<>();
    Stack<Integer> cnt = new Stack<>();
    for (char c : expression.toCharArray()) {
        if (c == '(' || c == '&' || c == '|') {
            sgn.push(c);
            continue;
        } else if (c == ')') sgn.pop();
        else {
            res.push((int)(c - '0'));
            cnt.push(1);
        }
        if (res.size() > 1 && sgn.peek() != '(') {
            int [] loc = getMinOperations(res.pop(), res.pop(), cnt.pop(), cnt.pop(), sgn.pop());
            res.push(loc[0]); // expr results
            cnt.push(loc[1]); // min operations
        }
    }
    return cnt.peek();
}
}

```

### 1.3.36 823. Binary Trees With Factors

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

```

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

```

### 1.3.37 363. 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.

1. 解题思路与分析: prefix sum 在二维矩阵上的应用

一维数组 prefix sum 在二维矩阵上的应用, 考虑先降一下维度

```

// 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 x = mat.length, y = mat[0].length, ans = Integer.MIN_VALUE;
    boolean flag = x <= y;
    int m = Math.min(x, y), n = Math.max(x, y);
    int [] sum = new int [n]; // 定义一维大的数组
    TreeSet<Integer> ts = new TreeSet<>();
    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(sum, 0);
        for (int j = i; j >= 0; j--) { // 遍历行: 行的和的累加是通过每列列的和的累加 sum 数组的值来实现的
            ts.clear();
            ts.add(0);
            for (int k = 0; k < n; k++) {
                sum[k] += mat[j][k];
                if (ts.contains(sum[k])) {
                    ans = Math.max(ans, sum[k]);
                }
                ts.add(sum[k]);
            }
        }
    }
    return ans;
}

```

```

    int cur = 0;
    // 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, sum - target <= setElement(要 set 中的元素, 尽可能地小)
    // 所以也就是在求大于等于 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: 返回大于或等于 e 的最小元素; 如果没有这样的元素, 则返回 null
        for (int k = 0; k < n; k++) { // 遍历列: 原始矩阵的行的和, 或者是列的和, 比较长的大的那一维的和
            if (flag) sum[k] += mat[j][k];
            else sum[k] += mat[k][j];
            cur += sum[k];
            Integer tmp = ts.ceiling(cur - target);
            if (tmp != null) ans = Math.max(ans, cur - tmp);
            ts.add(cur);
        }
    }
}
return ans;
}
+END_SRC
- 再稍微简洁一点儿的代码
#+BEGIN_SRC csharp
public int maxSumSubmatrix(int[][] mat, int target) { // 这么再看一下, 就清楚楚啦
    int m = mat.length, n = mat[0].length, ans = Integer.MIN_VALUE;
    int M = Math.min(m, n), N = Math.max(m, n);
    for (int i = 0; i < M; i++) {
        int[] sum = new int[N];
        for (int j = i; j < M; j++) {
            TreeSet<Integer> ts = new TreeSet<>();
            int cur = 0;
            for (int k = 0; k < N; k++) {
                sum[k] += m > n ? mat[k][j] : mat[j][k];
                cur += sum[k];
                if (cur <= target) ans = Math.max(ans, cur);
                Integer tmp = ts.ceiling(cur - target);
                if (tmp != null) ans = Math.max(ans, cur - tmp);
                ts.add(cur);
            }
        }
    }
    return ans;
}

```

## 2. 解题思路与分析: 暴力 + 优化

这道题给了我们一个二维数组, 让求和不超过的 K 的最大子矩形, 那么首先可以考虑使用 brute force 来解, 就是遍历所有的子矩形, 然后计算其和跟 K 比较, 找出不超过 K 的最大值即可。就算是暴力搜索, 也可以使用优化的算法, 比如建立累加和, 参见之前那道题 Range Sum Query 2D - Immutable, 可以快速求出任何一个区间和, 下面的方法就是这样的, 当遍历到 (i, j) 时, 计算 sum(i, j), 表示矩阵 (0, 0) 到 (i, j) 的和, 然后遍历这个矩阵中所有的子矩形, 计算其和跟 K 相比, 这样既可遍历到原矩阵的所有子矩阵, 参见代码如下:

```

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[r][c];
                    if (r > 0) d -= pre[r-1][c];
                    if (c > 0) d -= pre[r][c-1];
                    if (r > 0 && c > 0) d += pre[r-1][c-1];
                    if (d <= k) res = Math.max(res, d);
                }
            }
        }
    }
    return res;
}

```

### 1.3.38 1884. Egg Drop With 2 Eggs and N Floors - Medium

You are given two identical eggs and you have access to a building with n floors labeled from 1 to n.

You know that there exists a floor f where  $0 \leq f \leq n$  such that any egg dropped at a floor higher than f will break, and any egg dropped at or below floor f will not break.

In each move, you may take an unbroken egg and drop it from any floor x (where  $1 \leq x \leq n$ ). If the egg breaks, you can no longer use it. However, if the egg does not break, you may reuse it in future moves.

Return the minimum number of moves that you need to determine with certainty what the value of f is.

```
// 思路: https://zhuanlan.zhihu.com/p/41257286 可以借这个思路理解一下
// DP[i][j] 表示用 i 个鸡蛋，j 层楼的情况下最坏情况下所需扔鸡蛋的最少数目。
// 可知初始条件为：
// DP[1][0] = 0; DP[2][0] = 0;
// DP[1][1] = 1; DP[2][1] = 1;
// DP[1][i] = i; // i = 1 ... n
// 对于 DP[2][i], i = 2 ... n 的情况，我们可以这样考虑：
// 遍历 j=2...i, 求 DP[2][j]，分两种情况：
// 如果第 1 个鸡蛋在第 j-1 层摔破了，则我们在第 j 层只需摔第 2 个鸡蛋一次即可，此时总摔鸡蛋数为 DP[1][j-1]+1。
// 注意上面的 1 是因为第 j 层需要摔第 2 个鸡蛋 1 次。为什么 DP[1][j-1] 不能写成 1 呢？因为第 1 个鸡蛋在第 j-1 层摔破了，我们不能肯定在第 1, 2, ..., j-2 层不会摔破。
// 如果第 1 个鸡蛋在第 j-1 层没有摔破，则我们在第 j 到 i 层有 2 个鸡蛋可以摔，此时退化到 DP[2][i-j] 的情况。该种情形下总共扔 1+DP[2][i-j]。那个 1 就是表示第一次尝试。
// 因为是求最坏情况下的数目，所以 DP[2][j]=1 + max(DP[1][j-1]+1, DP[2][i-j])。
// 而我们是要求所有最坏情况下的最少数目，所以 DP[2][j]=min(1 + max(DP[1][j-1]+1, DP[2][i-j])). i = 2...n, j = 2...i。
public int twoEggDrop(int n) { // 1 + 2 + 3 + ... + x >= n ==> get x
    if (n <= 1) return n;
    int [][] dp = new int [3][n+1]; // DP[i][j] 表示用 i 个鸡蛋，j 层楼的情况下最坏情况下所需扔鸡蛋的最少数目。
    for (int i = 0; i < 3; i++)
        Arrays.fill(dp[i], Integer.MAX_VALUE);
    dp[1][0] = 0;
    dp[1][1] = 1;
    dp[2][0] = 0;
    dp[2][1] = 1;
    for (int i = 1; i <= n; i++) dp[1][i] = i;
    for (int i = 2; i <= n; i++)
        for (int j = 2; j <= i; j++)
            dp[2][i] = Math.min(dp[2][i], 1 + Math.max(dp[1][j-1], dp[2][i-j])); // 1: 这个鸡蛋在 j-i 层扔了一次，要统计入结果
    return dp[2][n];
}
public int twoEggDrop(int n) { // 1 + 2 + 3 + ... + x >= n ==> get x
    if (n <= 2) return n;
    return (int)(Math.ceil((-1 + Math.sqrt((long)n * 8 + 1)) / 2.0));
}
```

### 1.3.39 887. Super Egg Drop - Hard

You are given k identical eggs and you have access to a building with n floors labeled from 1 to n.

You know that there exists a floor f where  $0 \leq f \leq n$  such that any egg dropped at a floor higher than f will break, and any egg dropped at or below floor f will not break.

Each move, you may take an unbroken egg and drop it from any floor x (where  $1 \leq x \leq n$ ). If the egg breaks, you can no longer use it. However, if the egg does not break, you may reuse it in future moves.

Return the minimum number of moves that you need to determine with certainty what the value of f is.

#### 1. 解题思路与分析

- <https://charlesliuyx.github.io/2018/10/11/%E3%80%90%E7%9B%B4%E8%A7%82%E7%AE%97%E6%B3%95%E3%80%91Egg%20Puzzle%20%E9%B8%A1%E8%9B%8B%E9%9A%BE%E9%A2%98/>
- <https://www.shuzhiduo.com/A/D8548M4Q5E/>

#### 2. 解题思路与分析 $1 \leq K \leq 100, 1 \leq N \leq 10000$

这道题说给了我们 K 个鸡蛋，还有一栋共 N 层的大楼，说是鸡蛋有个临界点的层数 F，高于这个层数扔鸡蛋就会碎，否则就不会，问我们找到这个临界点最小需要多少操作，注意这里的操作只有当前还有没碎的鸡蛋才能进行。这道题是基于经典的扔鸡蛋的问题改编的，原题是有 100 层楼，为了测鸡蛋会碎的临界点，最少可以扔几次？答案是只用扔 14 次就可以测出来了，讲解可以参见 [油管上的这个视频](<https://www.youtube.com/watch?v=NGtt7GJ1uiM>)，这两道题看着很相似，其实是有不同的。这道题限制了鸡蛋的个数 K，假设我们只有 1 个鸡蛋，碎了就不能再用了，这时我们要测 100 楼的临界点的时候，只能一层一层去测，当某层鸡蛋碎了之后，就知道临界点了，所以最坏情况要测 100 次，注意要跟经典题目中扔 14 次要区分出来。那么假如有两个鸡蛋呢，其实需要的次数跟经典题目中的一样，都是 14 次，这是为啥呢？因为在经典题目中，我们是分别间隔 14, 13, 12, ..., 2, 1，来扔鸡蛋的，当我们有两个鸡蛋的时候，我们也可以这么扔，第一个鸡蛋仍在 14 楼，若碎了，说明临界点一定在 14 楼以内，可以用第二个鸡蛋去一层一层的测试，所以最多操作 14 次。若第一个鸡蛋没碎，则下一次扔在第 27 楼，假如碎了，说明临界点在 (14, 27] 范围内，用第二个鸡蛋去一层一层测，总次数最多 13 次。若第一个鸡蛋还没碎，则继续按照 39, 50, ..., 95, 99，等层数去测，总次数也只能越来越少，不会超过 14 次的。但是照这种思路分析的话，博主就不太清楚有 3 个鸡蛋，在 100 楼测，最少的步骤数，答案是 9 次，博主不太会分析怎么测的，各位看官大神知道的话一定要告诉博主啊。

其实这道题比较好的解法是用动态规划 Dynamic Programming，因为这里有两个变量，鸡蛋数 K 和楼层数 N，所以就要使用一个二维数组 DP，其中  $dp[i][j]$  表示有 i 个鸡蛋，j 层楼要测需要的最小操作数。那么我们在任意 k 层扔鸡蛋的时候就有两种情况（注意这里的 k 跟鸡蛋总数 K 没有任何关系，k 的范围是  $[1, j]$ ）：

鸡蛋碎掉：接下来就要用  $i-1$  个鸡蛋来测  $k-1$  层，所以需要  $dp[i-1][k-1]$  次操作。  
鸡蛋没碎：接下来还可以用  $i$  个鸡蛋来测  $j-k$  层，所以需要  $dp[i][j-k]$  次操作。

因为我们每次都要面对最坏的情况，所以在第 j 层扔，需要  $\max(dp[i-1][k-1], dp[i][j-k])+1$  步，状态转移方程为：

$$dp[i][j] = \min(dp[i][j], \max(dp[i-1][k-1], dp[i][j-k])+1) \quad (1 \leq k \leq j)$$

```

public int superEggDrop(int k, int n) { // tle: O(KN^2) 还需要优化一下
    if (k < 1 || n < 1) return 0;
    int [] pre = new int [n+1]; // 上一层备忘录, 存储鸡蛋数量-1 的 n 层楼条件下的最优化尝试次数
    int [] cur = new int [n+1]; // 当前备忘录, 存储当前鸡蛋数量的 n 层楼条件下的最优化尝试次数
    for (int i = 1; i <= n; i++) // 把备忘录每个元素初始化成最大的尝试次数
        cur[i] = i;
    for (int i = 2; i <= k; i++) {
        pre = cur.clone(); // 当前备忘录拷贝给上一次备忘录, 并重新初始化当前备忘录
        for (int j = 1; j <= n; j++) cur[j] = j;
        for (int j = 1; j <= n; j++) // 需要想办法去优化时间复杂度。这种写法里面我们枚举了 [1, j] 范围所有的 k 值, 总时间复杂度为 O(KN^2),
            for (int x = 1; x < j; x++)
                cur[j] = Math.min(cur[j], 1 + Math.max(pre[x-1], cur[j-x]));
        // 扔鸡蛋的楼层从 1 到 m 枚举一遍, 如果当前算出的尝试次数小于上一次算出的尝试次数, 则取代上一次的尝试次数。
        // 这里可以打印 k 的值, 从而知道第一个鸡蛋是从第几次扔的。
        cur[j] = Math.min(cur[j], 1 + Math.max(pre[x-1], cur[j-x]));
    }
    return cur[n];
}

```

这种写法会超时 Time Limit Exceeded, 代码请参见评论区 1 楼, OJ 对时间卡的还是蛮严格的, 所以我们就需要想办法去优化时间复杂度。这种写法里面我们枚举了  $[1, j]$  范围所有的  $k$  值, 总时间复杂度为  $O(KN^2)$ , 若我们仔细观察  $dp[i-1][k-1]$  和  $dp[i][j-k]$ , 可以发现前者是随着  $k$  递增, 后者是随着  $k$  递减, 且每次变化的值最多为 1, 所以只要存在某个  $k$  值使得二者相等, 那么就能得到最优解, 否则取最相近的两个  $k$  值做比较, 由于这种单调性, 我们可以在  $[1, j]$  范围内对  $k$  进行二分查找, 找到第一个使得  $dp[i-1][k-1]$  不小于  $dp[i][j-k]$  的  $k$  值, 然后用这个  $k$  值去更新  $dp[i][j]$  即可, 这样时间复杂度就减少到了  $O(KN \lg N)$ , 其实也是险过, 参见代码如下:

```

// 若我们仔细观察 dp[i-1][k-1] 和 dp[i][j-k], 可以发现前者是随着 k 递增, 后者是随着 k 递减, 且每次变化的值最多为 1,
// 所以只要存在某个 k 值使得二者相等, 那么就能得到最优解, 否则取最相近的两个 k 值做比较,
// 由于这种单调性, 我们可以在 [1, j] 范围内对 k 进行二分查找, 找到第一个使得 dp[i-1][k-1] 不小于 dp[i][j-k] 的 k 值, 然后用这个 k 值去更新 dp[i][j]
// 这样时间复杂度就减少到了 O(KN \lg N)
public int superEggDrop(int k, int n) { // O(KN \lg N)
    int [][] dp = new int [k+1][n+1];
    for (int i = 1; i <= n; i++) dp[1][i] = i; // 把只有一个鸡蛋的情况初始化为最大值
    for (int i = 2; i <= k; i++) {
        for (int j = 1; j <= n; j++) {
            dp[i][j] = j;
            int l = 1, r = j, m = 0;
            while (l < r) {
                m = (l + r) / 2;
                if (dp[i-1][m-1] < dp[i][j-m]) l = m + 1; // < 左边随 m 递增, < 右边随 m 递减, 这个性质在这里很重要, 所以可以二分查找
                else r = m;
            } // 这里查找的右边界也注意一下
            dp[i][j] = Math.min(dp[i][j], Math.max(dp[i-1][r-1], dp[i][j-r]) + 1); // 用这个鸡蛋测试了一次了
        }
    }
    return dp[k][n];
}

```

进一步来想, 对于固定的  $k$ ,  $dp[i][j-k]$  会随着  $j$  的增加而增加, 最优决策点也会随着  $j$  单调递增, 所以在每次移动  $j$  后, 从上一次的最优决策点的位置来继续向后查找最优点即可, 这样时间复杂度就优化到了  $O(KN)$ , 我们使用一个变量  $s$  表示当前的  $j$  值下的的最优决策点, 然后当  $j$  值改变了, 我们用一个 while 循环, 来找到下一个最优决策点  $s$ , 使得  $dp[i-1][s-1]$  不小于  $dp[i][j-s]$ , 参见代码如下:

```

public int superEggDrop(int k, int n) { // O(KN)
    int [][] dp = new int [k+1][n+1];
    for (int i = 1; i <= n; i++) dp[1][i] = i; // 把只有一个鸡蛋的情况初始化为最大值
    for (int i = 2; i <= k; i++) {
        int s = 1; // k
        for (int j = 1; j <= n; j++) { // n
            dp[i][j] = j;
            while (s < j && dp[i-1][s-1] < dp[i][j-s]) s++; // 因为单调性, s 一直向右滑动, 总共执行 n 次
            dp[i][j] = Math.min(dp[i][j], Math.max(dp[i-1][s-1], dp[i][j-s]) + 1);
        }
    }
    return dp[k][n];
}

```

其实我们还可以进一步优化时间复杂度到  $O(K \lg N)$ , 不过就比较难想到了, 需要将问题转化一下, 变成已知鸡蛋个数, 和操作次数, 求最多能测多少层楼的临界点。还是使用动态规划 Dynamic Programming 来做, 用一个二维 DP 数组, 其中  $dp[i][j]$  表示当有  $i$  次操作, 且有  $j$  个鸡蛋时能测出的最高的楼层数。再来考虑状态转移方程如何写, 由于  $dp[i][j]$  表示的是在第  $i$  次移动且使用第  $j$  个鸡蛋测试第  $dp[i-1][j-1]+1$  层, 因为上一个状态是第  $i-1$  次移动, 且用第  $j-1$  个鸡蛋。此时还是有两种情况:

鸡蛋碎掉: 说明至少可以测到的不会碎的层数就是  $dp[i-1][j-1]$ 。

鸡蛋没碎: 那这个鸡蛋可以继续利用, 此时我们还可以再向上查找  $dp[i-1][j]$  层。

那么加上当前层, 总共可以通过  $i$  次操作和  $j$  个鸡蛋查找的层数范围是  $[0, dp[i-1][j-1] + dp[i-1][j] + 1]$ , 这样就可以得到状态转移方程如下:

$dp[i][j] = dp[i-1][j-1] + dp[i-1][j] + 1$

当  $dp[i][K]$  正好小于  $N$  的时候,  $i$  就是我们要求的最小次数了, 参见代码如下:

```

public int superEggDrop(int k, int n) { // O(KlgN): 为什么说这里就是 O(klgN) 呢？因为 dp[m][k] 的增长呈指数级的？
    int [][] dp = new int [n+1][k+1]; // 在第 i 次移动且使用第 j 个鸡蛋测试第 dp[i-1][j-1]+1 层
    int m = 0; // 最小测试次数
    while (dp[m][k] < n) { // 当 dp[m][k] == n，退出循环的时候，m 就是要求的最小操作次数了
        ++m;
        for (int j = 1; j <= k; j++)
            dp[m][j] = dp[m-1][j-1] + dp[m-1][j] + 1; // 分上一次测的 第 j-1 个鸡蛋 碎了 和 没有碎 两种情况来更新当前值
    }
    return m;
}

```

我们可以进一步的优化空间，因为当前的操作次数值的更新只跟上一次操作次数有关，所以我们并不需要保存所有的次数，可以使用一个一维数组，其中  $dp[i]$  表示当前次数下使用  $i$  个鸡蛋可以测出的最高楼层。状态转移方程的推导思路还是跟上面一样，参见代码如下：

```

public int superEggDrop(int k, int n) { // O(KlgN): 空间压缩
    int [] dp = new int [k+1]; // dp[i] 表示当前次数下使用 i 个鸡蛋可以测出的最高楼层
    int ans = 0;
    for (; dp[k] < n; ans++) // 压缩代码到一行
        for (int i = k; i >= 1; i--) // 压缩空间后，因为是想要根据上一行 [ans-1] 时的 i-1/i 来更新当前结果，需要倒序遍历以免产生脏数据
            dp[i] = dp[i] + dp[i-1] + 1;
    return ans;
}

```

下面这种方法就非常的 tricky 了，居然推导出了使用  $k$  个鸡蛋，移动  $x$  次所能测的最大楼层数的通项公式，推导过程可以参见 [这个帖子]([https://leetcode.com/problems/super-egg-drop/discuss/181702/C%2B%2B-codeRuntime-O-msO\(1\)-spacewith-explosion.No-DPWhat-we-need-is-mathematical-thought!](https://leetcode.com/problems/super-egg-drop/discuss/181702/C%2B%2B-codeRuntime-O-msO(1)-spacewith-explosion.No-DPWhat-we-need-is-mathematical-thought!))，通项公式如下：

$$f(k,x) = x(x-1)\dots(x-k)/k! + \dots + x(x-1)(x-2)/3! + x(x-1)/2! + x$$

这数学功底也太好了吧，有了通向公式后，我们就可以通过二分搜索法 Binary Search 来快速查找满足题目的  $x$ 。这里其实是博主之前总结贴 [LeetCode Binary Search Summary 二分搜索法小结](<http://www.cnblogs.com/grandyang/p/6854825.html>) 中的第四类，用子函数当作判断关系，这里子函数就是用来实现上面的通向公式的，不过要判断，当累加和大于等于  $N$  的时候，就要把当的累加和返回，这样相当于进行了剪枝，因为在二分法中只需要知道其跟  $N$  的大小关系，并不 care 到底大了多少，这样快速定位  $x$  的方法运行速度貌似比上面的 DP 解法要快不少，但是这通项公式尼玛谁能容易的推导出来，只能膜拜叹服了，参见代码如下：

```

public int superEggDrop(int k, int n) { // O(KlgN): 数学推导找出了推理公式，速度最快
    int l = 1, r = n;
    while (l < r) { // 寻找右边界
        int m = l + (r - l) / 2;
        if (helper(m, k, n) < n) l = m + 1;
        else r = m;
    }
    return r;
}
int helper(int m, int k, int n) { // 能查找到的最大楼层数
    int ans = 0, r = 1;
    for (int i = 1; i <= k; i++) {
        r *= m - i + 1;
        r /= i;
        ans += r;
        if (ans >= n) break;
    }
    return ans;
}

```

### 1.3.40 1981. Minimize the Difference Between Target and Chosen Elements

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

```

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

```

```

        // }
        for (int k = sum.length(); (k = sum.previousSetBit(k-1)) >= 0; ) {
            newSum.set(k+mat[i][j]);
        }
    }
    sum = newSum;
}
int ans = 4900;
for (int k = sum.length(); (k = sum.previousSetBit(k-1)) >= 0;) {
    int diff = Math.abs(k - target);
    ans = Math.min(ans, diff);
}
return ans;
}

public int minimizeTheDifference(int[][] mat, int target) {
    int m = mat.length;
    int n = mat[0].length;
    int diff = Integer.MAX_VALUE, limit = 4900;
    int [] dp = new int[limit];
    for (int i = 0; i < n; i++) // 相当于是手工实现 java BitSet
        dp[mat[0][i]] = 1;
    for (int i = 1; i < m; i++) {
        int [] tmp = new int [limit];
        for (int v = limit-1; v >= 0; v--) {
            if (dp[v] == 0) continue;
            for (int j = 0; j < n; j++) {
                if (v + mat[i][j] < limit)
                    tmp[v+mat[i][j]] = 1;
            }
        }
        System.arraycopy(tmp, 0, dp, 0, dp.length);
    }
    for (int i = 0; i < limit; i++)
        if (dp[i] > 0) diff = Math.min(diff, Math.abs(i-target));
    return diff; // min difference
}
}

```

### 1.3.41 801. Minimum Swaps To Make Sequences Increasing - Hard

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

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

An array `arr` is strictly increasing if and only if  $\text{arr}^1 < \text{arr}^2 < \text{arr}^3 < \dots < \text{arr}[\text{arr.length} - 1]$ .

```

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

```

### 1.3.42 837. New 21 Game - Medium

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

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

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

Return the probability that Alice has  $n$  or fewer points.

Answers within 10<sup>-5</sup> of the actual answer are considered accepted.

```

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

```

### 1.3.43 1105. Filling Bookcase Shelves - Medium

You are given an array books where books[i] = [thickness<sub>i</sub>, height<sub>i</sub>] indicates the thickness and height of the *i*th book. You are also given an integer shelfWidth.

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

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

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

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

```

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

```

- 后来又出的 bug

```

public int minHeightShelves(int [][] b, int w) {
    int n = b.length, curLayerWidth = 0, max = 0;
    int[] dp = new int[n+1];
    Arrays.fill(dp, n+1);
    dp[0] = 0;
    for (int i = 1; i <= n; i++) { // 依次求摆放前 i 本书的最小高度
        dp[i] = dp[i-1] + b[i-1][1]; // 初始化为最大值：单列一行，再看之前的行有没有空隙可以放入之前的层？
        curLayerWidth = b[i-1][0];
        max = b[i-1][1];
        for (int j = i-1; j > 0 && curLayerWidth + b[j-1][0] <= w; j--) { // 将前 i-1 本书从第 i-1 本开始放在与 i 同一层，直到这一层摆满或者所有的书都摆好
            curLayerWidth += b[j-1][0];
            max = Math.max(max, b[j-1][1]);
            // dp[i] = Math.min(dp[i], dp[j] + max); // bug: dp[j] + max 是从第 j 本开始，包括第 j 本，这与之后的书放到最后一层所能形成的小高度
            dp[i] = Math.min(dp[i], dp[j-1] + max); // 所以，是用 j 之前所开成的最小高度，dp[j-1] 加上最后一层所能形成的最小高度
        }
    }
    return dp[n];
}

```

### 1.3.44 790. Domino and Tromino Tiling

You have two types of tiles: a  $2 \times 1$  domino shape and a tromino shape. You may rotate these shapes.

Given an integer  $n$ , return the number of ways to tile an  $2 \times n$  board. Since the answer may be very large, return it modulo  $10^9 + 7$ .

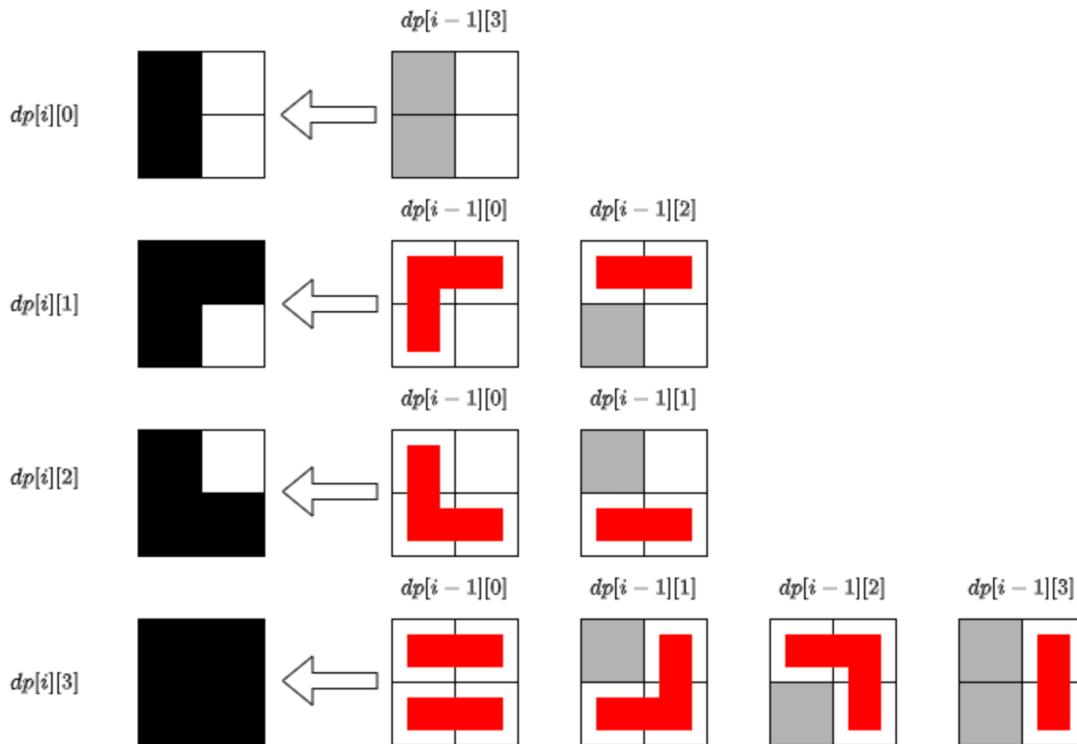
In a tiling, every square must be covered by a tile. Two tilings are different if and only if there are two 4-directionally adjacent cells on the board such that exactly one of the tilings has both squares occupied by a tile.

#### 1. 官方的：更为直接的图形动规题的分析方法

考虑这么一种平铺的方式：在第  $i$  列前面的正方形都被瓷砖覆盖，在第  $i$  列后面的正方形都没有被瓷砖覆盖（ $i$  从 1 开始计数）。那么第  $i$  列的正方形有四种被覆盖的情况：

- 一个正方形都没有被覆盖，记为状态 0；
- 只有上方的正方形被覆盖，记为状态 1；
- 只有下方的正方形被覆盖，记为状态 2；
- 上下两个正方形都被覆盖，记为状态 3。

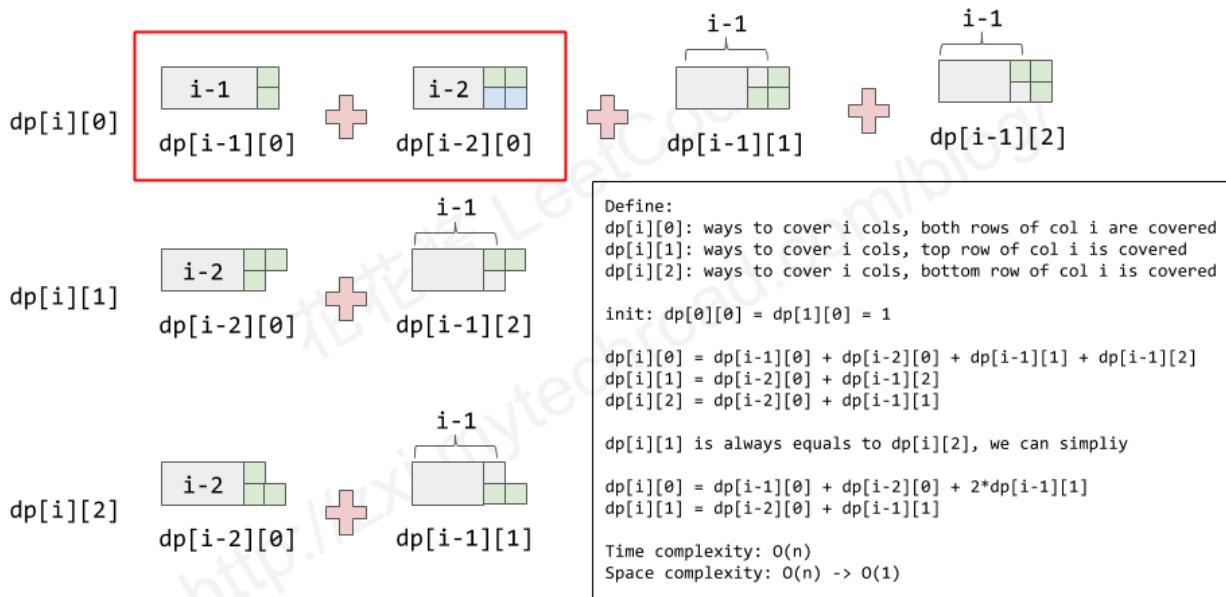
使用  $dp[i][s]$  表示平铺到第  $i$  列时，各个状态  $s$  对应的平铺方法数量。考虑第  $i - 1$  列和第  $i$  列正方形，它们之间的状态转移如下图（红色条表示新铺的瓷砖）：



初始时  $dp[0][0] = 0$ ,  $dp[0][1] = 0$ ,  $dp[0][2] = 0$ ,  $dp[0][3] = 1$ , 对应的状态转移方程 ( $i > 0$ ) 为：

```
static final int mod = (int)1e9 + 7;
public int numTilings(int n) {
    long [][] f = new long [n+1][4];
    f[0][3] = 1;
    for (int i = 1; i <= n; i++) {
        f[i][0] = f[i-1][3];
        f[i][1] = (f[i-1][0] + f[i-1][2]) % mod;
        f[i][2] = (f[i-1][0] + f[i-1][1]) % mod;
        f[i][3] = (f[i-1][0] + f[i-1][1] + f[i-1][2] + f[i-1][3]) % mod;
    }
    return (int)f[n][3];
}
```

#### 2. 解题思路与分析



```
public int numTilings(int n) {
    int mod = (int)1e9 + 7;
    int [][] dp = new int [n+1][2];
    dp[0][0] = 1;
    dp[1][0] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i][0] = (int)((dp[i-1][0] + dp[i-2][0]) % mod + (2 * dp[i-1][1]) % mod) % mod;
        dp[i][1] = (int)(dp[i-2][0] + dp[i-1][1]) % mod;
    }
    return dp[n][0];
}
```

### 1.3.45 1997. First Day Where You Have Been in All the Rooms - Medium

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

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

Assuming that on a day, you visit room  $i$ , if you have been in room  $i$  an odd number of times (including the current visit), on the next day you will visit a room with a lower or equal room number specified by  $\text{nextVisit}[i]$  where  $0 \leq \text{nextVisit}[i] \leq i$ ; if you have been in room  $i$  an even number of times (including the current visit), on the next day you will visit room  $(i + 1) \bmod n$ . Return the label of the first day where you have been in all the rooms. It can be shown that such a day exists. Since the answer may be very large, return it modulo  $10^9 + 7$ .

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

### 1.3.46 943. Find the Shortest Superstring - Hard

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

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

- 深搜 + 记忆数组 + 裁枝

```
public String shortestSuperstring(String [] sa) { // 回溯: 暴搜 + 剪枝, 但回溯仍然是最慢的方法
    n = sa.length;
    max = new int [n][n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++) {
            if (i == j) continue;
            for (int k = Math.min(sa[i].length(), sa[j].length()); k >= 1; k--) // 不想遍历所有, 找到一个有效最优解, 就剪枝中断
                if (sa[i].substring(sa[i].length()-k).equals(sa[j].substring(0, k))) {
                    max[i][j] = k; // sa[i] 的尾与 sa[j] 的头的 最长公共后缀 长度
                }
        }
}
```

```

        break;
    }
}

dp = new int [1 << n][n];
ans = new int [n]; // 最终答案: 最小长度的字符串下标位置
vis = new boolean [n];
dfs(new int [n], 0, 0, 0);
String s = sa[ans[0]];
for (int i = 1; i < n; i++) { // 当前字符串的前缀已经被上一个字符串的后缀 cover 了, 所以只取后没被覆盖住的后半部分
    int cmmLen = max[ans[i-1]][ans[i]];
    s += sa[ans[i]].substring(cmmLen);
}
return s;
}

int [][] dp;
int [][] max; // max common length between two strings
boolean [] vis;
int n, maxLen = Integer.MIN_VALUE; // BUG: has to be initialized
int [] ans;
private void dfs(int [] a, int idx, int sum, int state) {
    if (idx == n) {
        if (sum > maxLen) {
            maxLen = sum;
            // ans = Arrays.copyOf(a, n);
            ans = a.clone(); // 效果一样
        }
        return ;
    }
    for (int i = 0; i < n; i++) {
        if (vis[i]) continue;
        int mask = state | (1 << i);
        int curLen = sum + (idx == 0 ? 0 : max[a[idx-1]][i]);
        if (dp[mask][i] > 0 && dp[mask][i] >= curLen) continue; // == 的情况可以剪枝, 因为 dp[][] 本来就是记忆着各路径状态下的全局最优解, 不能更优就剪掉
        vis[i] = true;
        a[idx] = i;
        dp[mask][i] = curLen; // BUG: 需要这一行记忆化, 加速搜索与剪枝, 重中之重不可记忆, otherwise tle !!!
        dfs(a, idx+1, curLen, mask);
        vis[i] = false;
    }
}
}

```

- 动态规划

```

public String shortestSuperstring(String[] s) {
    int n = s.length;
    String [][] dp = new String [1 << n][n]; // 这个 dp 的设计还比较新颖奇特:
    int [][] max = new int [n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (i == j) continue;
            for (int k = Math.min(s[i].length(), s[j].length()); k >= 1; k--)
                if (s[i].substring(s[i].length()-k).equals(s[j].substring(0, k))) {
                    max[i][j] = k;
                    break;
                }
        }
    }
    for (int i = 0; i < n; i++) dp[1 << i][i] = s[i]; // 初始化: 每个字符串与自己的最长公共后缀串就是它本身
    for (int r = 1; r < 1 << n; r++)
        for (int i = 0; i < n; i++) {
            if (((r >> i) & 1) == 0) continue;
            for (int j = 0; j < n; j++) {
                if (i == j || (((r >> j) & 1) == 0)) continue; // 保证状态 r 是包含了字符串 i 和 j 的有效 state
                String cur = dp[r ^ (1 << j)][i] + s[j].substring(max[i][j]);
                if (dp[r][j] == null || dp[r][j].length() > cur.length()) // dp[i][j]: 这里比较字符串的长度操作起来就比较复杂一点儿
                    dp[r][j] = cur;
            }
        }
    int r = (1 << n) - 1;
    String ans = dp[r][0];
    for (int i = 1; i < n; i++)
        if (dp[r][i].length() < ans.length()) ans = dp[r][i];
    return ans;
}

```

## 1. 解题思路与分析

我们的算法包括三个部分:

预先计算出所有的 overlap(A[i], A[j]);

使用动态规划计算出所有的 dp(mask, i), 并记录每个状态从哪个状态转移得来, 记为 parent;

通过 parent 还原这个字符串。

```

public String shortestSuperstring(String[] A) {
    int N = A.length;
    int[][] overlaps = new int[N][N];
    for (int i = 0; i < N; ++i)

```

```

for (int j = 0; j < N; ++j) if (i != j) {
    int m = Math.min(A[i].length(), A[j].length());
    for (int k = m; k >= 0; --k)
        if (A[i].endsWith(A[j].substring(0, k))) {
            overlaps[i][j] = k;
            break;
        }
}
int[][] dp = new int[1<<N][N]; // dp[mask][i] = most overlap with mask, ending with ith element
int[][] parent = new int[1<<N][N];
for (int mask = 0; mask < (1<<N); ++mask) {
    Arrays.fill(parent[mask], -1);
    for (int bit = 0; bit < N; ++bit) if (((mask >> bit) & 1) > 0) {
        // Let's try to find dp[mask][bit]. Previously, we had a collection of items represented by pmask.
        int pmask = mask ^ (1 << bit);
        if (pmask == 0) continue;
        for (int i = 0; i < N; ++i) if (((pmask >> i) & 1) > 0) {
            // For each bit i in pmask, calculate the value if we ended with word i, then added word 'bit'.
            int val = dp[pmask][i] + overlaps[i][bit];
            if (val > dp[mask][bit]) {
                dp[mask][bit] = val;
                parent[mask][bit] = i;
            }
        }
    }
}
// # Answer will have length sum(len(A[i]) for i) - max(dp[-1])
// Reconstruct answer, first as a sequence 'perm' representing the indices of each word from left to right.
int[] perm = new int[N];
boolean[] vis = new boolean[N];
int t = 0;
int mask = (1 << N) - 1;
// p: the last element of perm (last word written left to right)
int p = 0;
for (int j = 0; j < N; ++j)
    if (dp[(1<<N) - 1][j] > dp[(1<<N) - 1][p])
        p = j;
// Follow parents down backwards path that retains maximum overlap
while (p != -1) {
    perm[t++] = p;
    vis[p] = true;
    int p2 = parent[mask][p];
    mask ^= 1 << p;
    p = p2;
}
// Reverse perm
for (int i = 0; i < t/2; ++i) {
    int v = perm[i];
    perm[i] = perm[t-1-i];
    perm[t-1-i] = v;
}
// Fill in remaining words not yet added
for (int i = 0; i < N; ++i)
    if (!vis[i]) perm[t++] = i;
// Reconstruct final answer given perm
StringBuilder ans = new StringBuilder(A[perm[0]]);
for (int i = 1; i < N; ++i) {
    int overlap = overlaps[perm[i-1]][perm[i]];
    ans.append(A[perm[i]].substring(overlap));
}
return ans.toString();
}

```

### 1.3.47 964. Least Operators to Express Number - Hard

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

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

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

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

接下来就要处理  $x$  小于 target 的情况了，此时由于不知道  $x$  到底比 target 小多少，若差距太大的话，肯定不能用加号，

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

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

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

注意， $2^k$  需要的符号是  $k$ ，最后因为第一个一定可以是正的，省一个符号。

看了花花酱的题解，感觉更像是 bfs。cost 小的点先扩展。这个再看一下

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

### 1.3.48 1955. Count Number of Special Subsequences - Hard 把这些个递推公式记住

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

For example, [0,1,2] and [0,0,1,1,1,2] are special. In contrast, [2,1,0],  $^2$ , and [0,1,2,0] are not special. Given an array  $nums$  (consisting of only integers 0, 1, and 2), return the number of different subsequences that are special. Since the answer may be very large, return it modulo  $10^9 + 7$ .

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

定义：

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

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

若  $j = nums[i]$  则需要分类计算：

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

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

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

最后答案为  $f[n - 1][2]$ 。

代码实现时, 可以把第一维压缩掉。

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

### 1.3.49 446. Arithmetic Slices II - Subsequence - Hard

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

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

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

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

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

好, 既然决定要用 DP 了, 那么首先就要确定  $dp$  数组的定义了, 刚开始我们可能会考虑使用一个一维的  $dp$  数组, 然后  $dp[i]$  定义为范围为  $[0, i]$  的子数组中等差数列的个数。定义的很简单, OK, 但是基于这种定义的状态转移方程却十分的难想。我们想对于  $(0, i)$  之间的任意位置  $j$ , 如何让  $dp[i]$  和  $dp[j]$  产生关联呢? 是不是只有  $A[i]$  和  $A[j]$  的差值  $diff$ , 跟  $A[j]$  之前等差数列的差值相同, 才会有关系, 所以差值  $diff$  是一个很重要的隐藏信息 Hidden Information, 我们必须要在  $dp$  的定义中考虑进去。所以一维  $dp$  数组是罩不住的, 必须升维, 但是用二维  $dp$  数组的话, 差值  $diff$  那一维的范围又是个问题, 数字的范围是整型数, 所以差值的范围也很大, 为了节省空间, 我们建立一个一维数组  $dp$ , 数组里的元素不是数字, 而是放一个  $HashMap$ , 建立等差数列的差值和当前位置之前差值相同的数字个数之间的映射。我们遍历数组中的所有数字, 对于当前遍历到的数字, 又从开头遍历到当前数字, 计算两个数字之差  $diff$ , 如果越界了不做任何处理, 如果没越界, 我们让  $dp[i]$  中  $diff$  的差值映射自增 1, 因为此时  $A[i]$  前面有相差为  $diff$  的  $A[j]$ , 所以映射值要

加 1。然后我们看  $dp[j]$  中是否有  $diff$  的映射，如果有的话，说明此时相差为  $diff$  的数字至少有三个了，已经能构成题目要求的等差数列了，将  $dp[j][diff]$  加入结果  $res$  中，然后再更新  $dp[i][diff]$ ，这样等遍历完数组， $res$  即为所求。

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

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

最终累计出来的结果是跟上面红色的数字相关，分别对应着如下的等差数列：

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

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

### 方法一：动态规划 + 哈希表

我们首先考虑至少有两个元素的等差子序列，下文将其称作弱等差子序列。

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

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

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

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

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

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

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

Refrain from using lambda expressions inside loops.

```
public int numberOfArithmeticSlices(int[] a) {
    int n = a.length, ans = 0;
    Map<Integer, Integer>[] dp = new HashMap[n];
    dp[0] = new HashMap<>();
    for (int i = 1; i < n; i++) {
        dp[i] = new HashMap<>();
        for (int j = 0; j < i; j++) {
            // for (int j = i-1; j >= 0; j--) { // BUG: 反向遍历不可以
            long diff = (long)a[i] - a[j]; // bug: (long)a[i]
            if (diff > Integer.MAX_VALUE || diff < Integer.MIN_VALUE) continue;
            int diff_ = (int)diff;
            dp[i].put(diff_, dp[i].getOrDefault(diff_, 0) + 1); // bug: 这里加的是 1, 不是 2 // 这里先更新上
            if (dp[j].containsKey(diff_)) {
                ans += dp[j].get(diff_); // 只自增 1 (而不是记为 2), 为的是方便这里统计结果
                dp[i].put(diff_, dp[i].get(diff_) + dp[j].get(diff_)); // 再加上之前累积的
            }
        }
    }
}
```

```

        }
    }
    return ans;
}

```

### 1.3.50 639. Decode Ways II - Hard

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

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

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

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

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

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

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

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

```

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

```

**2. 解题思路与分析**

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

如果 x==0, 且 y==0, 无法解码, 返回0;  
 如果只有x==0, 则y只能单独放在最后, 不能与x合并(不能以0开头), 此时有: dp[i] = dp[i-1];  
 如果只有y==0, 则y不能单独放置, 必须与x合并, 并且如果合并结果大于26, 返回0, 否则有: dp[i] = dp[i-2];  
 如果 xy<=26: 则y可以“单独”放在abcy的每个解码结果之后, 并且如果abcy以y单独结尾, 此时可以合并xy作为结尾, 而这种解码种数就是abcy的解码结果, 此时有: dp[i+1];  
 如果 xy>26: 此时y又不能与y合并, y只能单独放在dp[i]的每一种情况的最后, 此时有: dp[i+1] = dp[i]

```

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

```

```

        dp[i] = dp[i-2];
    }else if(n>26){
        dp[i] = dp[i-1];
    }else{
        dp[i] = dp[i-1]+dp[i-2];
    }
}
return dp[dp.length-1];
}

```

### 1.3.51 629. K Inverse Pairs Array - Hard

For an integer array nums, an inverse pair is a pair of integers [i, j] where  $0 \leq i < j < \text{nums.length}$  and  $\text{nums}[i] > \text{nums}[j]$ .

Given two integers n and k, return the number of different arrays consist of numbers from 1 to n such that there are exactly k inverse pairs. Since the answer can be huge, return it modulo  $10^9 + 7$ .

#### 1. 解题思路与分析

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

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

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

……

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

所以： $dp[n][k] = dp[n-1][k] + dp[n-1][k-1] + dp[n-1][k-2] + \dots + dp[n-1][k+n-1] + dp[n-1][k-n+1]$  // A  
 $dp[n][k+1] = dp[n-1][k+1] + dp[n-1][k] + dp[n-1][k-1] + dp[n-1][k-2] + \dots + dp[n-1][k+1-n+1]$  // B  
 $dp[n][k+1] - dp[n][k] = dp[n-1][k+1] - dp[n-1][k-n+1];$  // B-A  
 $dp[n][k+1] = dp[n][k] + dp[n-1][k+1] - dp[n-1][k-n+1];$  // 移项  
// 将 k+1 换回成 k, 可以得到：  
 $dp[n][k] = dp[n][k-1] + dp[n-1][k] - dp[n-1][k-n]$

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

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

```

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

```

#### 2. 解题思路与分析

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

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

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

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

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

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

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

### 1.3.52 1787. Make the XOR of All Segments Equal to Zero - Hard

You are given an array  $\text{nums}$  and an integer  $k$ . The XOR of a segment  $[\text{left}, \text{right}]$  where  $\text{left} \leq \text{right}$  is the XOR of all the elements with indices between  $\text{left}$  and  $\text{right}$ , inclusive:  $\text{nums}[\text{left}] \text{ XOR } \text{nums}[\text{left}+1] \text{ XOR } \dots \text{ XOR } \text{nums}[\text{right}]$ .

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

#### 1. 解题思路与分析

### 解题思路：

容易发现，最后得到的数组一定满足：

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

因此可以考虑将同一组中的数放在一起进行考虑。

首先，我们对每一组分别进行计数。

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

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

在转移时，我们有两种选择：

1. 将当前组的数全都改为同一个值，此时不管我们之前如何选择，都可以利用这一个值的选择得到任何一个异或值，那么我们自然应该选择之前的代价中最小的那个，此时的代价为  $size[i] + \min dp[val]$ ，新状态的异或值可以为  $[0, 2^D)$  中的任意一个值。这一转移需要进行  $K$  次。
2. 使用当前组中的某一个值  $val'$ 。此时我们需要枚举之前的异或值  $val$ （或枚举达到的目标值，二者是等价的），此时的代价为  $size[i] - cnt[i][val'] + dp[val]$ ，新状态的异或值为  $val \oplus val'$ 。这一转移最多需要进行  $N$  次。
  - 时间复杂度  $\mathcal{O}(2^D \cdot (K + N))$ 。其中  $D = 10$ 。
  - 空间复杂度  $\mathcal{O}(N + 2^D)$ 。

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

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

因此可将数组中的元素按上述规律，每间隔  $k$  个的数字为一组进行分组。

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

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

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

完成  $k$  个组的动态规划后， $dp^1$  就是所求的解。

```
public int minChanges(int[] a, int k) {
    int n = a.length;
    Map<Integer, Integer>[] group = new HashMap[k]; // 存储 k 个组、各组中各个数字数量
    int[] cnt = new int[k]; // k 元素片段里，每个下标对应的所有元素总个数 // 各组大小，它们会有可能不同吗？
    for (int i = 0; i < k; i++) {
        group[i] = new HashMap<>();
        for (int j = i; j < n; j += k) { // 将数组中的每个元素分布到其在 k 元素片段中下标位置所在的组里去，并统计重复出现次数
            group[i].put(a[j], group[i].getOrDefault(a[j], 0) + 1);
            cnt[i]++;
        }
    }
    int r = 1 << 10; // 题中 nums[i] < 2^10，为了遍历所有可能更改值，以取最小
    int[] dp = new int[r], curDp = new int[r]; // 当前组异或到对应数字时的更改次数
    Arrays.fill(dp, Integer.MAX_VALUE);
    dp[0] = 0;
    for (int i = 0; i < k; i++) { // 遍历 k 个元素的片段——中的每个元素，逐元素优化出全局最优解
        int minValue = Arrays.stream(dp).min().getAsInt(); // 累积到上一个元素的全局最优解
        Arrays.fill(curDp, minValue + cnt[i]); // 变为当前组中不存在数字的改变次数：之前的最小改变次数 + 当前组元素个数
        for (int j = 0; j < r; j++) {
            if (dp[j] == Integer.MAX_VALUE) continue;
            for (Map.Entry<Integer, Integer> en : group[i].entrySet()) {
                int num = en.getKey(), v = en.getValue(), xorNum = num ^ j;
                curDp[xorNum] = Math.min(curDp[xorNum], dp[j] + cnt[i] - v);
            }
        }
        dp = Arrays.copyOf(curDp, r); // 将遍历到当前组、累积较优解的 curDp 复制入全局最优解 dp 数组中
    }
}
```

```

    }
    return dp[0];
}

```

### 1.3.53 1735. Count Ways to Make Array With Product - Hard 乘积为 K 的质因子数排列组合的总个数：分解质因子

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

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

#### 1. 解题思路与分析

##### 题意分析

回顾题意：对于一个查询  $(n, k)$  而言，我们需要找出  $n$  个整数的一个排列，使得它们的乘积为  $k$ 。

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

$$k = p_1^{c_1} \cdot p_2^{c_2} \cdots \cdot p_m^{c_m}$$

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

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

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

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

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

$$\prod_{k=1}^{m} dp[n][c_k]$$

##### 算法细节

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

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

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

最后，我们就可以回答每个查询了。

```

// 在找出每个变量的上界之后，就可以在 O(maxn * maxc^2) 的时间内预处理 dp 数组。
// 随后，我们还要预先求出 [2, maxn] 之间的全部质数，复杂度为 O(maxn)。
public int[] waysToFillArray(int[][] q) {
    int n = q.length;
    int mod = (int)1e9 + 7;
    int maxC = 15; // 找出「加和」的最大值 cmax。根据前面的描述，cmax 即为质因数数量的最大值。
    int maxN = 0, maxK = 0; // 而由于质因数最小为 2, maxkmax 最大为 10^4 因此质因数数量不会超过 log_2 10^4 < 15
    for (int i = 0; i < n; i++) { // 需要遍历数组，找到所有 n, kn, k 的最大值 maxn, maxk
        maxN = Math.max(maxN, q[i][0]);
        maxK = Math.max(maxK, q[i][1]);
    }
}

```

```

}
long [][] dp = new long [maxN + 1][maxC + 1]; // dp[i][j] 代表给定一个正整数 jj, 满足长度为 ii, 且加和为 jj 的非负整数排列的数量。
for (int i = 1; i <= maxC; i++)
    dp[1][i] = 1;
for (int i = 1; i <= maxN; i++)
    dp[i][0] = 1;
for (int i = 2; i <= maxN; i++)
    for (int j = 1; j <= maxC; j++)
        for (int k = 0; k <= j; k++) { // 为了求解 dp[i][j], 我们考虑排列中的第一个整数, 它的取值范围为 [0,j][0,j],
            dp[i][j] += dp[i-1][j-k]; // 若它取值为 kk, 则余下 i-1 个整数的加和需要为 j-kjk, 对应的方案数目为 dp[i-1][jk]
            dp[i][j] %= mod;
        }
int [] isPrime = new int [maxK + 1]; // 分解乘积的质因子
Arrays.fill(isPrime, 1);
List<Integer> primes = new ArrayList<>();
for (int i = 2; i <= maxK; i++) {
    if (isPrime[i] == 1)
        primes.add(i);
    for (int j = i*2; j <= i*i && j <= maxK; j += i) // 最大乘积为 maxK 的数组, 分解出小的质因子了, 那么凡是小质因子的乘积倍数的数都不是质数
        isPrime[j] = 0;
}
int [] ans = new int [n];
for (int i = 0; i < n; i++) {
    int m = q[i][0], k = q[i][1];
    List<Integer> cs = new ArrayList<>(); // 乘积 k 的质因子表
    for (int p : primes) {
        if (p > k) break;
        int cnt = 0, left = k;
        while (left % p == 0) {
            left /= p;
            cnt++;
        }
        if (cnt > 0) cs.add(cnt); // 乘积 k 中各质因子的个数 (指数)
    }
    long res = 1;
    for (int c : cs) {
        res *= dp[m][c]; // 数组长度为 n, 数组和为质因子 c 的指数个数的所有可能的分布数合数, 各质因子之间个数之间相乘
        res %= mod;
    }
    ans[i] = (int)res;
}
return ans;
}

```

- 下面这种方法理解得还不是很透

```

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

```

```

        c[i][j] = (c[i-1][j-1] + c[i-1][j]) % mod;
    }
}
public long get(int n, int m) {
    return c[n][m];
}
}

```

### 1.3.54 1359. Count All Valid Pickup and Delivery Options - Hard

Given  $n$  orders, each order consist in pickup and delivery services.

Count all valid pickup/delivery possible sequences such that delivery( $i$ ) is always after of pickup( $i$ ).

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

#### 1. 解题思路与分析

就是总共有  $2N$  个位置，每次放一两个，还剩下多少个位置可以合理占用

```

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

```

### 1.3.55 1187. Make Array Strictly Increasing - Hard 需要重写

Given two integer arrays arr1 and arr2, return the minimum number of operations (possibly zero) needed to make arr1 strictly increasing.

In one operation, you can choose two indices  $0 \leq i < \text{arr1.length}$  and  $0 \leq j < \text{arr2.length}$  and do the assignment  $\text{arr1}[i] = \text{arr2}[j]$ .

If there is no way to make arr1 strictly increasing, return -1.

#### 1. 解题思路与分析

```

public int makeArrayIncreasing(int[] a, int[] b) {
    b = Arrays.stream(b).distinct().toArray();
    Arrays.sort(b);
    int m = a.length, n = b.length;
    int minCnt = 0, minValue = 0;
    Queue<int> q = new LinkedList<>();
    q.offer(new int[] {-1, 0}); // 初始化，假想第 0 位的前一位是-1
    for (int i = 0; i < m; i++) {
        minValue = Integer.MAX_VALUE;
        for (int size = q.size() - 1; size >= 0; size--) {
            int[] cur = q.poll(); // 取前一位的一个选择
            if (a[i] > cur[0]) minCnt = Math.min(minCnt, cur[1]); // 先不急将当前选择加入 queue 中，找到最小的再说
            minValue = binarySearchMin(b, cur[0]); // 查找一个比前一个数大的最小数
            if (minValue != -1) q.offer(new int[] {minValue, cur[1] + 1}); // 将这个最小数加入到 queue 中，操作次数在前一位的基础上加一
        }
        if (minCnt != Integer.MAX_VALUE) // 如果当前位置可以保持不变，将最小次数加入到 queue 中
            q.offer(new int[] {a[i], minCnt});
    }
    if (q.size() == 0) return -1; // 如果最后一位没有合法的选择方案，返回-1
    int ans = Integer.MAX_VALUE;
    while (!q.isEmpty()) ans = Math.min(ans, q.poll()[1]);
    return ans;
}
private int binarySearchMin(int[] arr, int v) {
    int l = 0, r = arr.length - 1, m = 0;
    while (l < r) { // l,r: [0, 1]
        m = l + (r - l) / 2;
        if (arr[m] <= v) l = m + 1;
        else r = m;
    }
    return arr[l] > v ? arr[l] : -1; // 所以这里要再判断一下
}

```

#### 2. 解题思路与分析

这里若是要替换后面的数字为较大的数字，那么就需要在 arr2 中找到比当前数字大的数字，为了让整个数组更容易的递增，那么这个较大数应该尽量越小越好，所以就是要找到第一个比当前数字大的数。为了更容易的在 arr2 中查找，而不是每次都遍历整个数组，需要给 arr2 排个序，然后用二分搜索来查找更高效一些，这里也可以将 arr2 放到一个 TreeSet 中，利用其自动排序的特点，之后再进行二分搜索就行了。这道题的正确解法是用动态规划 Dynamic Programming，这里的 dp 表达式比较难想，一般来说，dp 值都是定义为题目中要求的值，而这道题是个例外，这里的

$dp[i][j]$  表示对于数组中的前  $j$  个数字组成的子数组中，需要替换  $i$  次可以使得其变为严格递增，且第  $j$  个数字的最小值为  $dp[i][j]$ 。这里的  $dp$  值不是定义为替换次数，而是第  $j$  个数字的最小值（可能是替换后的值），因为要保证数组严格递增，最后一个数字的大小很重要，这是个关键信息，而这个数字的大小跟数组坐标之间没有啥必然联系，所以这个信息不太好放到  $dp$  数组的坐标中，而所求的替换次数跟数组长度是相关的，因为其不可能超过数组的总长度，最差的情况也就是将整个  $arr1$  数组都替换了（当然还需要考虑  $arr2$  的长度）。

接下来就来考虑状态转移方程怎么写，由于这里的  $j$  表示前  $j$  个数字，那么第  $j$  个数字实际上是  $arr1[j-1]$ ，若第  $j$  个数字大于  $dp[i][j-1]$ ，这里表示对于前  $j-1$  个数字，替换  $i$  次可以使得其严格递增，且第  $j-1$  个数字为  $dp[i][j-1]$ ，这样的话就不需要额外的替换操作，还是严格递增的，则  $dp[i][j]$  可以赋值为  $arr1[j-1]$ 。若此时  $i$  大于 0，说明之前已经进行过替换操作，则上一个操作状态是  $dp[i-1][j-1]$ ，当前操作是从  $arr2$  中选一个数字替换  $arr1$  的第  $j$  个数字，这里就要在  $arr2$  中选择第一个大于  $dp[i-1][j-1]$  的数字，若存在的话，就用这个数字来更新  $dp[i][j]$  的值。若某个时刻  $j$  等于  $n$  了，说明已经到  $arr1$  的末尾了，若此时  $dp[i][j]$  不等于  $INT\_MAX$ （初始值），说明是可以将整个  $arr1$  替换成严格递增的数组的，替换次数就是  $i$ ，直接返回即可。最终循环退出了，返回 -1，参见代码如下：

```
int makeArrayIncreasing(vector<int>& arr1, vector<int>& arr2) {
    int n = arr1.size();
    if (n == 1) return 0;
    set<int> st(arr2.begin(), arr2.end());
    vector<vector<int>> dp(n + 1, vector<int>(n + 1, INT_MAX));
    dp[0][0] = INT_MIN;
    for (int j = 1; j <= n; ++j) {
        for (int i = 0; i <= j; ++i) {
            if (arr1[j - 1] > dp[i][j - 1]) {
                dp[i][j] = arr1[j - 1];
            }
            if (i > 0) {
                auto it = st.upper_bound(dp[i - 1][j - 1]);
                if (it != st.end()) dp[i][j] = min(dp[i][j], *it);
            }
            if (j == n && dp[i][j] != INT_MAX) return i;
        }
    }
    return -1;
}
```

## 1.4 多维数个数、数种类数多一维 k 介人的 $dp[i][j][k]$

### 1.4.1 920. Number of Music Playlists - Hard

Your music player contains  $n$  different songs. You want to listen to goal songs (not necessarily different) during your trip. To avoid boredom, you will create a playlist so that:

Every song is played at least once. A song can only be played again only if  $k$  other songs have been played. Given  $n$ , goal, and  $k$ , return the number of possible playlists that you can create. Since the answer can be very large, return it modulo  $10^9 + 7$ .

#### 1. 解题思路与分析

当加入的是一首新歌，则表示之前的  $L-1$  首歌中有  $j-1$  首不同的歌曲，其所有的组合情况都可以加上这首新歌，那么当前其实有  $N-(j-1)$  首新歌可以选。

当加入的是一首重复的歌，则表示之前的  $L-1$  首歌中已经有了  $j$  首不同的歌，那么若没有  $K$  的限制，则当前有  $j$  首重复的歌可以选。但是现在有了  $K$  的限制，意思是两首重复歌中间必须要有  $K$  首其他的歌，则当前只有  $j-K$  首可以选。而当  $j < k$  时，其实这种情况是为 0 的。  
 $<="" li="">$

综上所述可以得到状态转移方程：

$$dp[i][j] = \sum_{k=0}^{j-1} dp[i-1][j-1] \times (N-(j-1))$$

$$dp[i][j] = \sum_{k=0}^{j-1} dp[i-1][j-1] \times (N-(j-1)) \quad (j \leq K)$$

```
public int numMusicPlaylists(int n, int goal, int k) {
    long mod = (int)1e9 + 7;
    long [][] dp = new long [goal+1][n+1]; // dp[i][j]: 播完 i 首用了 j 首不同的曲子, 分第 i 首播不播第 j 首两种情况 (i >= j for sure)
    for (int i = 1; i <= goal; i++)
        for (int j = 1; j <= n; j++) {
            if (i < j) dp[i][j] = 0; // 这行不能省略
            else if (i == 1 && j == 1) dp[i][j] = n; // 用 1 首歌放完 1 次, 共有 n 种不同的选择
            else if (i > 1 && j == 1) {
                if (k == 0) dp[i][j] = n; // 相当于没有任何外加限制条件
                // else dp[i][j] = 0; // 这行可略
            } else // 分两种情况: 第 i 首不播第 j 首歌 (那么可以从前面 j-k 首里面选择一首), 和第 i 首播第 j 首歌 (第 j 首就可以从不曾播放过的 n-(j-1) 首里选择一首)
                dp[i][j] = (dp[i-1][j] * Math.max(j-k, 0) + (j == 0 ? 0 : dp[i-1][j-1] * (n - (j-1)))) % mod;
        }
    return (int)dp[goal][n];
}
```

- 简化一下代码

```

static final int mod = (int)1e9 + 7;
public int numMusicPlaylists(int n, int goal, int k) {
    long [][] dp = new long [goal+1][n+1]; // dp[i][j]: 播完 i 首用了 j 首不同的曲子，分第 i 首播不播第 j 首两种情况 (i >= j for sure)
    dp[0][0] = 1;
    for (int i = 1; i <= goal; i++) {
        for (int j = 1; j <= n; j++) {
            dp[i][j] = (dp[i-1][j-1] * (n - (j-1))) % mod; // 第 j 首放新歌
            if (j > k) // 第 j 首放 (j-k) 之前的某首歌
                dp[i][j] = (dp[i][j] + dp[i-1][j] * (j-k)) % mod;
        }
    }
    return (int)dp[goal][n];
}

```

## 1.4.2 1866. Number of Ways to Rearrange Sticks With K Sticks Visible - Hard

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

For example, if the sticks are arranged [1,3,2,5,4], then the sticks with lengths 1, 3, and 5 are visible from the left. Given  $n$  and  $k$ , return the number of such arrangements. Since the answer may be large, return it modulo  $10^9 + 7$ .

```

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

```

- dfs + memo

```

public int rearrangeSticks(int n, int k) {
    dp = new long [n+1][k+1];
    return (int)dfs(n, k);
}
int mod = (int)1e9 + 7;
long [][] dp;
private long dfs(int n, int k) {
    if (n < k || k == 0) return 0;
    if (n == k) return 1;
    if (dp[n][k] != 0) return dp[n][k];
    // instead of iterating for every stick
    // we are just multiplying number of ways with (n - 1)
    return dp[n][k] = (dfs(n-1, k-1) + (n - 1) * dfs(n-1, k)) % mod;
}

```

## 1.4.3 1916. Count Ways to Build Rooms in an Ant Colony - Hard

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

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

Return the number of different orders you can build all the rooms in. Since the answer may be large, return it modulo  $10^9 + 7$ .

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

本题求解过程涉及较多前置知识点，包括排列组合、乘法逆元、快速乘方等

读者需要掌握如下进阶知识点才能解决本题：

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

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

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

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

这样将除法运算转化为乘法运算, 就可以在计算的过程中进行取模了。

乘法逆元的具体计算方法可以参考题解的最后一节「进阶知识点：乘法逆元」。

### 方法一：动态规划

#### 提示 1

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

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

#### 提示 2

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

题目中的要求与拓扑排序的要求是等价的。

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

## 思路与算法

我们可以使用动态规划的方法求出方案数。

设  $f[i]$  表示以节点  $i$  为根的子树的拓扑排序方案数，那么：

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

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

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

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

因此，我们就可以得到状态转移方程：

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

## 细节

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

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

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

```

static final int mod = (int)1e9 + 7;
public int waysToBuildRooms(int[] prevRoom) {
    int n = prevRoom.length;
    // 求阶乘数列及对应逆元
    this.fac = new int [n]; // fac[i]=i!
    this.inv = new int [n]; // inv[i]=i!^(-1)
    fac[0] = inv[0] = 1;
    for (int i = 1; i < n; i++) {
        fac[i] = (int)((long)fac[i-1] * i % mod); // 算阶层
        inv[i] = quickMultiply(fac[i], mod - 2); // 乘法逆元：费马小定理：(fac[i]^(-1))%mod = (fac[i]^(mod-2))%mod 转换成快速幂操作
    }
    for (int i = 0; i < n; i++) // 记录各个节点与子节点之间的边
        m.computeIfAbsent(prevRoom[i], z -> new ArrayList<>()).add(i);
    return dfs(0)[1]; // 动态规划得到总体顺序数量 x
}
Map<Integer, List<Integer>> m = new HashMap<>();
int [] fac, inv;
private int [] dfs(int idx) { // 返回以当前节点为根的子树节点个数 及 内部排列数
    if (!m.containsKey(idx)) return new int [] {1, 1}; // 子节点，节点个数及内部排列数均为 1
    int cnt = 1, ans = 1; // 子树的结点个数、内部排列数
    for (Integer next : m.get(idx)) {
        int [] cur = dfs(next); // 递归得到子节点对应树的节点个数和排列数
        cnt += cur[0];
        ans = (int)((long)ans * cur[1] % mod * inv[cur[0]] % mod);
    }
    ans = (int)((long)ans * fac[cnt-1] % mod);
    return new int [] {cnt, ans};
}
private int quickMultiply(int x, int y) { // 快速幂：快速计算 x^y 的乘方
    long ans = 1, base = x;
    while (y > 0) {
        if ((y & 1) == 1)
            ans = (ans * base) % mod; // 指数是奇数次，就先乘一次底数
        base = base * base % mod; // 指数剩偶数次了，就可以直接底数先平方，指数除以 2
    }
}

```

```

        y >= 1; // 指数除 2
    }
    return (int) ans;
}
}

```

#### 1.4.4 1504. Count Submatrices With All Ones - Medium

Given an  $m \times n$  binary matrix  $\text{mat}$ , return the number of submatrices that have all ones.

##### 1. 解题思路与分析

首先我们对矩阵进行数据初始化。即求出每一行以及每一列上的前缀和。

遍历矩阵每一个点（两层循环），并以该点最为起点（ $\text{row}, \text{col}$ ），向右下方向画矩形（两层循环，分别循环矩形的宽  $\text{width}$  和高  $\text{height}$ ），注意矩形范围不能越界。起始时  $\text{width}$  和  $\text{height}$  分别为 0，即当前点自身是一个矩形。

当  $\text{width}$  扩大一格后，实际上是增加了  $(\text{row}, \text{col}+\text{width})$  到  $(\text{row}+\text{height}, \text{col}+\text{width})$  这一部分的面积（宽为  $\text{width}$ ，高为  $\text{height}$ ），我们通过前缀和数组求出该区域和是否等于  $\text{height}$ ，如果等于，返回结果加一即可。

$\text{width}$  扩大一格的操作同理。

```

public int numSubmat(int[][] mat) {
    int m = mat.length, n = mat[0].length;
    int [][] row = new int [m][n]; // 每一行的前缀和
    int [][] col = new int [m][n]; // 每一列的前缀和
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            row[i][j] = (j == 0 ? 0 : row[i][j-1]) + mat[i][j];
    for (int j = 0; j < n; j++)
        for (int i = 0; i < m; i++)
            col[i][j] = (i == 0 ? 0 : col[i-1][j]) + mat[i][j];
    int ans = 0;
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            for (int r = 0; i+r < m; r++) // 以当前点为顶点，向下扩大一格，r = 0 起点是 0，当前格自身也是答案
                for (int c = 0; j+c < n; c++) { // 以当前点为顶点，向右扩大一格
                    int x = i + r, y = j + c;
                    // 数新扩张区域内每行每列区域内长度累加和都等于长度，即新增区域每格都是 1
                    if ((j == 0 && row[x][y] == c+1 || j > 0 && row[x][y] - row[x][j-1] == c+1)
                        && (i == 0 && col[x][y] == r+1 || i > 0 && col[x][y] - col[i-1][y] == r+1))
                        ans++;
                    else break;
                }
    return ans;
}
}

```

##### 2. 解题思路与分析：单调栈 todo

```

private int res = 0;
private int n;
public int numSubmat(int[][] mat) {
    this.n = mat[0].length;
    // dp[j] : the height (number of consecutive '1's) of column j
    int[] dp = new int[n];
    for (int i = 0; i < mat.length; i++) {
        // calculating (updating) heights
        for (int j = 0; j < n; j++)
            dp[j] = mat[i][j] == 1 ? dp[j] + 1 : 0;
        enumerateRowByMinHeight(dp);
    }
    return res;
}
public void enumerateRowByMinHeight(int[] dp) {
    // monotonic stack storing indices : for index p < q in stack, dp[p] < dp[q]
    Deque<Integer> stack = new LinkedList<Integer>();
    stack.offerLast(-1);
    for (int j = 0; j < n; j++) {
        while (stack.peekLast() != -1 && dp[stack.peekLast()] >= dp[j]) {
            int idx = stack.pollLast();
            res += dp[idx] * (idx - stack.peekLast()) * (j - idx);
        }
        stack.offerLast(j);
    }
    while (stack.peekLast() != -1) {
        int idx = stack.pollLast();
        res += dp[idx] * (idx - stack.peekLast()) * (n - idx);
    }
}

// Used two Arrays to store number of consecutive ones on the left, and number of consecutive ones above(up)
// In one  $m \times n$  loop we can count the number of order  $1 \times M$  rectangles where  $M$  belongs to  $[1, m-1]$ 
// and we can count rectangles of order  $M \times N$  each time where  $M > 1$ . in the k index loop.
public int numSubmat(int[][] mat) {
    int m = mat.length, n = mat[0].length;
    int [][] left = new int [m][n]; // 每一行的前缀和
    int [][] up = new int [m][n]; // 每一列的前缀和
}
}

```

```

int [][] abov = new int [m][n]; // 每一列的前缀和
int ans = 0;
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        if (mat[i][j] == 1) {
            left[i][j] = (j == 0 ? 0 : left[i][j-1]) + 1;
            ans += (j == 0 ? 1 : left[i][j]);
            abov[i][j] = (i == 0 ? 0 : abov[i-1][j]) + 1;
            if (i > 0) {
                int min = left[i][j];
                for (int k = 1; k < abov[i][j]; k++) {
                    min = Math.min(min, left[i-k][j]);
                    ans += min;
                }
            }
        }
    }
}
return ans;
}

// In the first pass through the matrix, we store the heights of 1s above a given i,j
// In the second pass, we go through each element that is nonzero, scan leftwards,
// adding the minimum of the heights encountered until we reach the beginning of the row or hit a zero.
public int numSubmat(int[][] mat) {
    int m = mat.length, n = mat[0].length;
    for (int j = 0; j < n; j++) {
        int colsum = 0;
        for (int i = 0; i < m; i++) {
            if (mat[i][j] == 0) colsum = 0;
            else colsum += mat[i][j];
            mat[i][j] = colsum;
        }
    }
    int tot = 0;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < mat[i].length; j++) {
            int k = j;
            int min = Integer.MAX_VALUE;
            while (k >= 0 && mat[i][k] != 0) {
                min = Math.min(min, mat[i][k]);
                tot += min;
                k--;
            }
        }
    }
    return tot;
}

```

## 1.5 BitMask 掩码相关的【这是第三次不会的】

### 1.5.1 1595. Minimum Cost to Connect Two Groups of Points

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.

- 主要是，它可能涉及的过程状态很多，要想全了

```

public int connectTwoGroups(List<List<Integer>> a) { // 看来这个题。。。1595 要好好多想几遍【爱表哥，爱生活!!!】
    int m = a.size(), n = a.get(0).size();
    int [][] r = new int [m][1 << n]; // 这里，就是准备，待用
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < (1 << n); j++) {
            int sum = 0;
            for (int k = 0; k < n; k++)
                if ((j & (1 << k)) > 0) sum += a.get(i).get(k);
            r[i][j] = sum;
        }
    }
    int [][] f = new int [m+1][1 << n];
    Arrays.stream(f).forEach(z -> Arrays.fill(z, Integer.MAX_VALUE));
    // f[0][0] = 0; // 【BUG:】 初始化得不对，这里一个点是不行的，得一行点...
    f[1] = r[0];
    for (int i = 1; i < m; i++) // 遍历每一行：左边的每个点，要么去更新已连的右边点减少消耗，要么去连一个新的点
        // for (int j = (1 << n)-1; j > 0; j--) { // 遍历每一种与右点配对现状的更新可能性
        for (int j = 1; j < (1 << n); j++) {
            // 【更新】右边现存连点中的每个状态下的每个点，将其换连成左边这个点【仍然不对】...
            for (int k = 0; k < n; k++) { // 遍历右边当前状态 mask 下，每个点，【与左边当前遍历点】的更新可能性【情况二分】

```

```

// if ((j & (1 << k)) > 0) // 当前点连了一个某个左边点: 这里遍历的状态, 仍然不够, 左边这个待连点, 仍然是可以与右边一片点, 几个很多个点一起连的...
// f[i][j | (1 << k)] = Math.min(f[i][j | (1 << k)], f[i-1][j] + a.get(i).get(k));
// f[i+1][j] = Math.min(f[i+1][j], f[i][j] ^ (1 << k) + a.get(i).get(k)); // 把右边 k 点, 换成连左边 i 点
// f[i+1][j | (1 << k)] = Math.min(f[i+1][j | (1 << k)], f[i][j] + a.get(i).get(k)); // 把右边 k 点, 换成连左边 i 点

// else // 【BUG:】出发点是对的, 但没想全, 就是左边这个点, 不是只能连右边一个点, 还可以是一片点? 所以有很多状态...
f[i+1][j | (1 << k)] = Math.min(f[i+1][j | (1 << k)], f[i][j] + a.get(i).get(k));
}

// 【更新:】右边还没有连的连点中的每个状态下的每个还没能连的点, 将其连接成左边这个点, 左边这个点可以对应右边很多个点连接
int rest = (1 << n)-1 - j; // 想要表示: 右连, 所有还不曾被连过的点, 所有可能的状态 rest 表示截至第 i 行还没被选过的列
for (int k = rest; k > 0; k = (k - 1) & rest) // 只遍历没选过的列的所有组合
f[i+1][j | k] = Math.min(f[i+1][j | k], f[i][j] + r[i][k]);
// f[i][j | k] = Math.min(f[i][j | k], f[i-1][j] + r[i][k]);
}

// return f[m-1][(1 << n) - 1];
return f[m][(1 << n) - 1];
}

```

## 1.6 【第三次】不会的

### 1.6.1 1187. Make Array Strictly Increasing: 【动规:】思路独特

Given two integer arrays arr1 and arr2, return the minimum number of operations (possibly zero) needed to make arr1 strictly increasing.

In one operation, you can choose two indices  $0 \leq i < \text{arr1.length}$  and  $0 \leq j < \text{arr2.length}$  and do the assignment  $\text{arr1}[i] = \text{arr2}[j]$ .

If there is no way to make arr1 strictly increasing, return -1.

此题为经典的「300. 最长递增子序列」问题的变形题目，我们可以参考类似的题目解法。首先我们思考一下，由于要求数组严格递增，因此数组中不可能存在相同的元素，对于数组  $arr_2$  来说，可以不需要考虑数组中的重复元素，可以预处理去除  $arr_2$  的重复元素，假设数组  $arr_1$  的长度为  $n$ ，数组  $arr_2$  的长度为  $m$ ，此时可以知道最多可以替换的次数为  $\min(n, m)$ 。如何才能定义动态规划的递推公式，这就需要进行思考。我们设  $dp[i][j]$  表示数组  $arr_1$  中的前  $i$  个元素进行了  $j$  次替换后组成严格递增子数组末尾元素的最小值。当我们遍历  $arr_1$  的第  $i$  个元素时，此时  $arr_1[i]$  要么进行替换，要么进行保留，实际可以分类进行讨论：

- 此时如果  $arr_1[i]$  需要进行保留，则  $arr_1[i]$  一定严格大于前  $i - 1$  个元素替换后组成的严格递增子数组最末尾的元素。假设前  $i - 1$  个元素经过了  $j$  次变换后得到的递增子数组的末尾元素的最小值为  $dp[i - 1][j]$ ，如果满足  $arr_1[i] > dp[i - 1][j]$ ，则此时  $arr_1[i]$  可以保留加入到该子数组中且构成的数组严格递增；
- 此时如果  $arr_1[i]$  需要进行替换，则替换后的元素一定严格大于前  $i - 1$  个元素替换后组成的严格递增子数组最末尾的元素。假设前  $i - 1$  个元素经过了  $j - 1$  次变换后得到的递增子数组的末尾元素的最小值为  $dp[i - 1][j - 1]$ ，此时我们从  $arr_2$  找到严格大于  $dp[i - 1][j - 1]$  的最小元素  $arr_2[k]$ ，则此时将  $arr_2[k]$  加入到该子数组中且构成数组严格递增；
- 综上可知，每个元素在替换时只有两种选择，要么选择保留当前元素  $arr_1$ ，要么从  $arr_2$  中选择一个满足条件的最小元素加入到数组中，最少替换方案一定包含在上述替换方法中。我们可以得到以下递推关系：

$$\begin{cases} dp[i][j] = \min(dp[i][j], arr_1[i]), & \text{if } arr_1[i] > dp[i - 1][j] \\ dp[i][j] = \min(dp[i][j], arr_2[k]), & \text{if } arr_2[k] > dp[i - 1][j - 1] \end{cases}$$

为了便于计算，我们将  $dp[i][j]$  的初始值都设为  $\infty$ ，为了便于计算在最开始加一个哨兵，此时令  $dp[0][0] = -1$  表示最小值。实际计算过程如下：

- 为了方便计算，需要对  $arr_2$  进行预处理，去掉其中的重复元素，为了快速找到数组  $arr_2$  中的最小元素，还需要对  $arr_2$  进行排序；
- 依次尝试计算前  $i$  个元素在满足  $j$  次替换时的最小元素：
  - 如果当前元素  $arr_1[i]$  大于  $dp[i][j - 1]$ ，此时可以尝试将  $arr_1[i]$  替换为  $dp[i][j]$ ，即此时  $dp[i][j] = \min(dp[i][j], arr_1[i])$ 。
  - 如果前  $i - 1$  个元素可以满足  $j - 1$  次替换后成为严格递增数组，即满足  $dp[i - 1][j - 1] \neq \infty$ ，可以尝试在第  $j$  次替换掉  $arr_1[i]$ ，此时根据贪心原则，利用二分查找可以快速的找到严格大于  $dp[i - 1][j - 1]$  的最小值进行替换即可。
- 设当前数组  $arr_1[i]$  的长度为  $n$ ，如果前  $n$  个元素满足  $j$  次替换后成为严格递增数组，此时我们找到最小的  $i$  反向即可。

```
public int makeArrayIncreasing(int[] a, int[] b) {
    Arrays.sort(b);
    int n = a.length, p = -1; m = b.length;
    li = new ArrayList<>(); // 尽量写这种简单的，免得自己容易出错
    for (int v : b) {
        if (v != p) li.add(v);
        p = v;
    }
    m = li.size();
    int [][] f = new int [n+1][Math.min(m, n) + 1]; // 最多时行 Math.min(m, b) 次交换
    Arrays.stream(f).forEach(z -> Arrays.fill(z, Integer.MAX_VALUE));
    f[0][0] = -1; // 哨兵: -1
    for (int i = 1; i <= n; i++)
        for (int j = 0; j <= Math.min(i, m); j++) { // 分两种情况来更新
            /* 如果当前元素大于序列的最后一个元素 */
            if (a[i-1] > f[i-1][j])
                f[i][j] = Math.min(f[i-1][j], arr1[i]);
            else
                f[i][j] = Math.min(f[i-1][j], arr2[li.indexOf(a[i-1])]);
        }
}
```

```

f[i][j] = a[i-1];
/* 查找严格大于 dp[i - 1][j - 1] 的最小元素 */
if (j > 0 && f[i-1][j-1] != Integer.MAX_VALUE) {
    int idx = binarySearch(j-1, f[i-1][j-1]);
    if (idx != m)
        f[i][j] = Math.min(f[i][j], li.get(idx));
}
if (i == n && f[i][j] != Integer.MAX_VALUE) return j;
}
return -1;
}
List<Integer> li;
int m;
int binarySearch(int l, int v) { // 找第一个 >= v 的下标
    int r = m; // 右边界: 不包括【l,r】: 感觉这里我还没有弄很透彻...
    while (l < r) {
        int mid = (l + r) / 2;
        if (li.get(mid) > v) r = mid;
        else l = mid + 1;
    }
    return l;
}
}

```

## 1.6.2 416. Partition Equal Subset Sum 同硬币和问题，感觉总容易忘记，有点儿提示才能回想起 来

Given an integer array nums, return true if you can partition the array into two subsets such that the sum of the elements in both subsets is equal or false otherwise.

### 1. 动规: 二维

创建二维数组  $dp$ , 包含  $n$  行  $target + 1$  列, 其中  $dp[i][j]$  表示从数组的  $[0, i]$  下标范围内选取若干个正整数 (可以是 0 个), 是否存在一种选取方案使得被选取的正整数的和等于  $j$ 。初始时,  $dp$  中的全部元素都是 false。

在定义状态之后, 需要考虑边界情况。以下两种情况都属于边界情况。

- 如果不选取任何正整数, 则被选取的正整数等于 0。因此对于所有  $0 \leq i < n$ , 都有  $dp[i][0] = \text{true}$ 。
- 当  $i == 0$  时, 只有一个正整数  $nums[0]$  可以被选取, 因此  $dp[0][nums[0]] = \text{true}$ 。

对于  $i > 0$  且  $j > 0$  的情况, 如何确定  $dp[i][j]$  的值? 需要分别考虑以下两种情况。

- 如果  $j \geq nums[i]$ , 则对于当前的数字  $nums[i]$ , 可以选取也可以不选取, 两种情况只要有一个为 true, 就有  $dp[i][j] = \text{true}$ 。
  - 如果不选取  $nums[i]$ , 则  $dp[i][j] = dp[i - 1][j]$ ;
  - 如果选取  $nums[i]$ , 则  $dp[i][j] = dp[i - 1][j - nums[i]]$ 。
- 如果  $j < nums[i]$ , 则在选取的数字的和等于  $j$  的情况下无法选取当前的数字  $nums[i]$ , 因此有  $dp[i][j] = dp[i - 1][j]$ 。

状态转移方程如下:

$$dp[i][j] = \begin{cases} dp[i - 1][j] \mid dp[i - 1][j - nums[i]], & j \geq nums[i] \\ dp[i - 1][j], & j < nums[i] \end{cases}$$

最终得到  $dp[n - 1][target]$  即为答案。

// 【动规:】同拼硬币什么之类的题目是一样的. 所以, 有了最基本的提醒之后, 我是可以自己把它想出来的...
 public boolean canPartition(int[] a) { // 好慢, 还没有上面的, 自己瓣出来的一维的快

```

int n = a.length;
if (n < 2) return false;
int sum = Arrays.stream(a).sum(), max = Arrays.stream(a).max().getAsInt(), target = sum / 2;
if (sum % 2 != 0 || max > sum / 2) return false;
Arrays.sort(a); // 子集: 之类的, 一般顺序不重要, 所以解题时, 可以排序, 来帮助自己解答问题
boolean[][] f = new boolean[n][target + 1];
Arrays.stream(f).forEach(z -> Arrays.fill(z, false));
for (int i = 0; i < n; i++) f[i][0] = true;
f[0][a[0]] = true;

```

```

for (int i = 1; i < n; i++) {
    int v = a[i];
    for (int j = 1; j <= target; j++) {
        if (j < v) // 因为这些现取得的值 j, 不能通过当前数得到, 只能取它们先前只能得到的结果
            f[i][j] = f[i-1][j];
        else // 这些取值, 就可以通过添加一个当前数 a[i] 取得
            f[i][j] = f[i-1][j] | f[i-1][j-v];
    }
}
return f[n-1][target];
}

```

## 2. 动规: 一维, 压缩空间

```

public boolean canPartition(int[] a) { // 这里至少也得降序排列一下呀。。。先试着分两组, 往什么数组和小的一组里去添加之类的...
    int n = a.length;
    if (n < 2) return false;
    int sum = Arrays.stream(a).sum(), max = Arrays.stream(a).max().getAsInt(), target = sum / 2;
    if (sum % 2 != 0 || max > sum / 2) return false;
    Arrays.sort(a); // 子集: 之类的, 一般顺序不重要, 所以解题时, 可以排序, 来帮助自己解答问题
    boolean[] f = new boolean[target + 1];
    Arrays.fill(f, false);
    f[0] = true;
    // f[a[0]] = true; // 一维时, 这个初始化也不必要了...
    // for (int i = 1; i < n; i++) {
    for (int i = 0; i < n; i++) {
        int v = a[i];
        for (int j = target; j >= v; j--)
            f[j] |= f[j-v];
    }
    return f[target];
}

```



# Chapter 2

## dfs 记忆化搜索

### 2.0.1 1977. Number of Ways to Separate Numbers - Hard

You wrote down many positive integers in a string called *num*. However, you realized that you forgot to add commas to separate the different numbers. You remember that the list of integers was non-decreasing and that no integer had leading zeros.

Return the number of possible lists of integers that you could have written down to get the string *num*. Since the answer may be large, return it modulo  $10^9 + 7$ .

1. 动态规划: 与最长公共前缀

#### 思路与算法

我们用  $f[i][j]$  表示对于字符串 *num* 的第  $0, 1, \dots, j$  个字符进行划分，并且最后一个数使用了第  $i, i+1, \dots, j$  个字符的方案数。为了叙述方便，我们用  $num(i, j)$  表示该数。

那么如何进行状态转移呢？我们可以基于如下的一个事实：

对于数  $a$  和  $b$ ，如果  $a$  的位数严格大于  $b$  的位数，那么  $a$  一定严格大于  $b$ 。

由于  $f[i][j]$  中的最后一个数的位数为  $j - i + 1$ ，那么上一个数的位数小于等于  $j - i$  即可进行转移。上一个数的结尾在位置  $i - 1$ ，那么其开始下标需要大于等于：

$$(i - 1) - (j - i) + 1 = 2i - j$$

对应的状态即为：

$$f[2i - j][i - 1], f[2i - j + 1, i - 1], \dots, f[i - 1][i - 1]$$

此外，我们还需要比较  $num(2i - j - 1, i - 1)$  和  $num(i, j)$  的值的大小关系，此时这两个数的位数都是  $j - i + 1$ 。如果前者小于等于后者，那么  $f[i][j]$  还可以从  $f[2i - j - 1][i - 1]$  转移而来。因此，状态转移方程为：

$$f[i][j] = \begin{cases} \sum_{k=2i-j}^{i-1} f[k][i-1], & num(2i-j-1, i-1) > num(i, j) \\ \sum_{k=2i-j-1}^{i-1} f[k][i-1], & num(2i-j-1, i-1) \leq num(i, j) \end{cases}$$

需要注意的是：为了防止状态转移方程显得过于复杂，我们在状态转移方程中：

没有考虑  $2ij$  和  $2ij1$  是否超出边界。但在实际的代码编写中，需要保证求和式中  $kk$  的最小值不能小于  $00$ ；

没有考虑  $num(i,j)$  是否包含前导零。如果  $num[i]=0$ ，那么  $f[i][j]=0$ 。特别地，如果  $num^1=0$ ，那么不会有任何满足要求的划分方案，直接返回  $00$  作为答案，无需进行动态规划。

动态规划的边界条件为  $f[..] = 1$  其余的状态的初始值均为 0。最终的答案即为所有  $f[..][n-1]f[..][n1]$  的和，其中  $n$  是字符串  $\text{num}$  的长度。

## 前缀和优化

即使我们不考虑如何快速地比较  $\text{num}(2i-j-1, i-1)$  和  $\text{num}(i, j)$  的大小关系，上述动态规划的时间复杂度也为  $O(n^3)$ ：即我们需要  $O(n^2)$  的时间枚举  $i$  和  $j$ ，还需要  $O(n)$  的时间枚举  $k$  计算对应项的和。

然而我们可以发现，这些和是「连续」的，因此我们可以使用前缀和进行优化。设  $\text{pre}[i][j]$  表示：

$$\text{pre}[i][j] = \sum_{k=0}^i f[i][j]$$

那么状态转移方程可以改写为：

$$f[i][j] = \begin{cases} \text{pre}[i-1][i-1] - \text{pre}[2i-j-1][i-1], & \text{num}(2i-j-1, i-1) > \text{num}(i, j) \\ \text{pre}[i-1][i-1] - \text{pre}[2i-j-2][i-1], & \text{num}(2i-j-1, i-1) \leq \text{num}(i, j) \end{cases}$$

只要在计算  $f$  的过程中维护  $\text{pre}$ ，就可以将动态规划的时间复杂度优化至  $O(n^2)$ 。

此外，我们也可以无需显式地使用前缀和数组：如果我们按照先枚举  $i$  再枚举  $j$  的顺序计算  $f[i][j]$ ，那么有：

$$f[i][j] = \sum_{k=2i-j}^{i-1} f[k][i-1]$$

这里我们不考虑  $\text{num}(2i-j-1, i-1)$  和  $\text{num}(i, j)$  的大小关系，即使前者小于等于后者，多出的  $f[2i-j-1][i-1]$  这一项也可以  $O(1)$  的时间累加进  $f[i][j]$ ，无需刻意前缀和进行优化。

当  $j \rightarrow j+1$  时：

$$f[i][j+1] = \sum_{k=2i-j-1}^{i-1} f[k][i-1]$$

可以发现， $f[i][j+1]$  只比  $f[i][j]$  多出了  $f[2i-j-1][i-1]$  这一项，因此在求得  $f[i][j]$  的前提下，我们需要  $O(1)$  的时间即可得到  $f[i][j+1]$ 。

## 快速比较两个数的大小关系

此时，我们只剩最后一步，也就是快速比较  $\text{num}(2i - j - 1, i - 1)$  和  $\text{num}(i, j)$  的大小关系了。这一步可以使用预处理巧妙地解决。

记  $\text{lcp}[i][j]$  表示在字符串  $\text{nums}$  中，以  $i$  开始的后缀与以  $j$  开始的后缀的「最长公共前缀」的长度。直观上看，它表示：

- $\text{num}(i, i + \text{lcp}[i][j] - 1) = \text{num}(j, j + \text{lcp}[i][j] - 1)$ ;
- $\text{num}[i + \text{lcp}[i][j]] \neq \text{num}[j + \text{lcp}[i][j]]$  或者其中某一下标超出边界。

$\text{lcp}[i][j]$  可以很方便地使用动态规划求出，即：

$$\text{lcp}[i][j] = \begin{cases} \text{lcp}[i+1][j+1] + 1, & \text{num}[i] = \text{num}[j] \\ 0, & \text{num}[i] \neq \text{num}[j] \end{cases}$$

当我们求出了  $\text{lcp}$  后，就可以方便地比较  $\text{num}$  中两个子串的大小关系了。对于  $\text{num}(2i - j - 1, i - 1)$  和  $\text{num}(i, j)$ ：

- 如果  $\text{lcp}[2i - j - 1][i] \geq j - i + 1$ ，那么  $\text{num}(2i - j - 1, i - 1)$  一定等于  $\text{num}(i, j)$ ;
- 如果  $\text{lcp}[2i - j - 1][i] < j - i + 1$ ，那么  $\text{num}(2i - j - 1, i - 1)$  和  $\text{num}(i, j)$  的大小关系，等同于  $\text{num}[2i - j - 1 + \text{lcp}[2i - j - 1][i]]$  与  $\text{num}[i + \text{lcp}[2i - j - 1][i]]$  的大小关系。

这样做的原因在于，两个长度相等的数的「数值」大小关系是等同于它们「字典序」的大小关系的。因此我们找出这两个数的最长公共前缀，再比较最长公共前缀的下一个字符的大小关系即可。

至此，我们就将动态规划的时间复杂度完全优化至  $O(n^2)$ ，也就可以通过本题了。

## 注解

$\text{lcp}$  来源于 Longest Common Prefix，即最长公共前缀。如果读者研究过算法竞赛，学习过「后缀数组」，那么上述的  $\text{lcp}$  是可以通过后缀数组 + ST 表在  $O(n \log n)$  的时间内预处理得到的。但这已经远远超出了面试和笔试的难度，因此这里不再深入解释。

```
static final int mod = (int)1e9 + 7;
public int number0fCombinations(String t) {
    int n = t.length();
    char[] s = t.toCharArray();
    if (s[0] == '0') return 0;
    int[][] lcp = new int[n][n]; // 求的是：以 s[i] 开始的后缀，与以 s[j] 开始的后缀，两字符串的最长公共前缀长度
    for (int i = n-1; i >= 0; i--) { // 预处理 lcp
        lcp[i][n-1] = s[i] == s[n-1] ? 1 : 0;
        for (int j = i+1; j < n-1; j++)
            lcp[i][j] = s[i] == s[j] ? lcp[i+1][j+1] + 1 : 0;
    }
    int[][] dp = new int[n][n];
    for (int i = 0; i < n; i++) dp[0][i] = 1; // dp[0][...] = 1
    for (int i = 1; i < n; i++) {
        if (s[i] == '0') continue; // 有前导零，无需转移
        int preSum = 0;
        for (int j = i; j < n; j++) {
            int length = j - i + 1; // s[i, j]
            dp[i][j] = preSum; // dp[i][j] = dp[i][j-1] + (one item) // 这里是 j 从 i 开始累加的 dp[...] 前缀和
            if (i - length >= 0) { // 使用 lcp 比较 s[2i-j-1, i-1] 与 s[i, j] 的大小关系
                if (lcp[i-length][i] >= length || s[i-length + lcp[i-length][i]] < s[i + lcp[i-length][i]])
                    dp[i][j] = (dp[i][j] + dp[i-length][i-1]) % mod;
                preSum = (preSum + dp[i-length][i-1]) % mod; // 更新前缀和，这里是 j 从 i 开始累加的 dp[...] 前缀和
            }
        }
    }
    int ans = 0;
    for (int i = 0; i < n; i++) // 最终答案即为所有 dp[...][n-1] 的和
        ans = (ans + dp[i][n-1]) % mod;
    return ans;
}
```

- 另一种写法

```

private void getLongestCommonPrefixLength() { // Pre compute Longest Common Prefix sequence for each index in the string
    for (int i = n-1; i >= 0; i--) // 从右向左遍历，计算最长公共前缀序列长度
        for (int j = n-1; j >= 0; j--)
            if (s[i] == s[j]) {
                if (i >= n-1 || j >= n-1) lcp[i][j] = 1;
                else lcp[i][j] = lcp[i+1][j+1] + 1;
            } else lcp[i][j] = 0;
    }
private boolean compare(int i, int j, int len) { // compare substring of same length for value,
    int commonLength = lcp[i][j]; // 返回以 i 开始长度为 len 的序列是否比以 j 开始长度为 len 的序列（数值）小
    if (commonLength >= len) return true;
    return s[i + commonLength] <= s[j + commonLength]; // <= ? 为什么不可以等于呢？
}
long mod = (int)1e9 + 7;
int [][] lcp;
char [] s;
int n;
public int numberOfCombinations(String t) {
    if (t.charAt(0) == '0') return 0;
    n = t.length();
    this.s = t.toCharArray();
    lcp = new int[n][n];
    int [][] f = new int [n][n];
    int [][] pre = new int [n][n]; // 从右向左的累加和
    getLongestCommonPrefixLength(); // 计算从右向左遍历的最长公共前缀（右边，其实是后缀）
    for (int i = 0; i < n; i++) {
        f[0][i] = 1;
        pre[0][i] = 1;
    }
    for (int j = 1; j < n; j++) { // 跟上面超内存的写法是反着走，这次是从左向右遍历，可是两种方法，为什么就有一个会超内存呢？
        for (int i = 1; i <= j; i++) {
            if (s[i] == '0') {
                f[i][j] = 0;
                // continue;
            } else {
                f[i][j] = pre[i-1][i-1];
                if (i - (j-i+1) >= 0) // 现在长度为 i-j+1 的数，前面是否存在一个同样长度的数，即前一个数的第一个位下标是否 >= 0
                    f[i][j] -= pre[2*i-j-1][i-1];
                if (i - (j-i+1) >= 0 && compare(i-(j-i+1), i, j-i+1)) {
                    f[i][j] = (int)((f[i][j] + pre[i-(j-i+1)][i-1]) % mod);
                    if (i - (j-i+1) - 1 >= 0)
                        f[i][j] -= pre[i-(j-i+1)-1][i-1];
                }
            }
            f[i][j] = (int)((f[i][j] + mod) % mod);
            pre[i][j] = (int)((pre[i-1][j] + f[i][j]) % mod);
        }
    }
    return pre[n-1][n-1];
}

```

(a) 解题思路与分析 todo: 其它思路，改天补上

## 2.0.2 87. Scramble String - Hard 非常经典：要韧熟于心

We can scramble a string  $s$  to get a string  $t$  using the following algorithm:

If the length of the string is 1, stop. If the length of the string is  $> 1$ , do the following: Split the string into two non-empty substrings at a random index, i.e., if the string is  $s$ , divide it to  $x$  and  $y$  where  $s = x + y$ . Randomly decide to swap the two substrings or to keep them in the same order. i.e., after this step,  $s$  may become  $s = x + y$  or  $s = y + x$ . Apply step 1 recursively on each of the two substrings  $x$  and  $y$ . Given two strings  $s_1$  and  $s_2$  of the same length, return true if  $s_2$  is a scrambled string of  $s_1$ , otherwise, return false.

1. 朴素解法 (TLE) 一个朴素的做法根据「扰乱字符串」的生成规则进行判断。

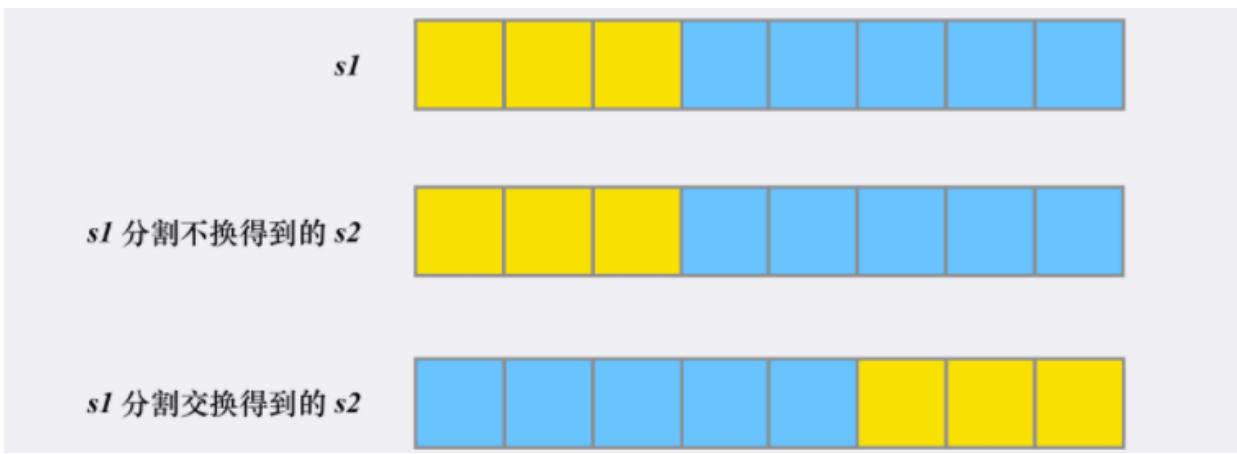
由于题目说了整个生成「扰乱字符串」的过程是通过「递归」来进行。

我们要实现 `isScramble` 函数的作用是判断  $s_1s_1$  是否可以生成出  $s_2s_2$ 。

这样判断的过程，同样我们可以使用「递归」来做：

假设  $s_1s_1$  的长度为  $n$ ，第一次分割的分割点为  $i$ ，那么  $s_1s_1$  会被分成  $[0, i][0, i]$  和  $[i, n][i, n]$  两部分。

同时由于生成「扰乱字符串」时，可以选交换也可以选不交换。因此我们的  $s_2s_2$  会有两种可能性：



因为对于某个确定的分割点， $s1s1$  固定分为两部分，分别为  $[0,i][0,i)$  &  $[i, n)[i,n)$ 。

而  $s2s2$  可能会有两种分割方式，分别  $[0,i][0,i)$  &  $[i,n)[i,n)$  和  $[0, n-i)[0,ni)$  &  $[n-i,n)[ni,n)$ 。

我们只需要递归调用 `isScramble` 检查  $s1s1$  的  $[0,i][0,i)$  &  $[i, n)[i,n)$  部分能否与「 $s2s2$  的  $[0,i][0,i)$  &  $[i,n)[i,n)$ 」或者「 $s2s2$  的  $[0, n-i)[0,ni)$  &  $[n-i,n)[ni,n)$ 」匹配即可。

同时，我们将「 $s1s1$  和  $s2s2$  相等」和「 $s1s1$  和  $s2s2$  词频不同」作为「递归」出口。

理解这套做法十分重要，后续的解法都是基于此解法演变过来。

```

private boolean idCheck(String ss, String tt) {
    int [] one = new int [26];
    int [] two = new int [26];
    char [] s = ss.toCharArray();
    char [] t = tt.toCharArray();
    for (int i = 0; i < s.length; i++)
        one[s[i] - 'a']++;
    for (int i = 0; i < t.length; i++)
        two[t[i] - 'a']++;
    for (int i = 0; i < 26; i++)
        if (one[i] != two[i])
            return false;
    return true;
}
public boolean isScramble(String s, String t) { // tle tle tle
    int n = s.length();
    if (n == 1) return s.charAt(0) == t.charAt(0);
    if (s.equals(t)) return true;
    if (!idCheck(s, t)) return false;
    for (int i = 1; i < n; i++) {
        System.out.println("\n i: " + i);
        String ls = s.substring(0, i), rs = s.substring(i);
        String ltone = t.substring(0, i), rttone = t.substring(i);
        String lttwo = t.substring(0, n-i), rtttwo = t.substring(n-i);
        if (isScramble(ls, ltone) && isScramble(rs, rttone)
            || isScramble(ls, rtttwo) && isScramble(rs, lttwo))
            return true;
    }
    return false;
}

```

时间复杂度:  $O(5^n)$

空间复杂度: 忽略递归与生成子串带来的空间开销，复杂度为  $O(1)$

## 2. 记忆化搜索 朴素解法卡在了 286 / 288 个样例。

我们考虑在朴素解法的基础上，增加「记忆化搜索」功能。

我们可以重新设计我们的「爆搜」逻辑：假设  $s1$  从  $i$  位置开始， $s2$  从  $j$  位置开始，后面的长度为  $len$  的字符串是否能形成「扰乱字符串」（互为翻转）。

那么在单次处理中，我们可分割的点的范围为  $[1, len)$ ，然后和「递归」一下，将  $s1$  分割出来的部分尝试去和  $s2$  的对应位置匹配。

同样的，我们将「入参对应的子串相等」和「入参对应的子串词频不同」作为「递归」出口。

```

private boolean idCheck(String ss, String tt) {
    int [] one = new int [26];
    int [] two = new int [26];
    char [] s = ss.toCharArray();
    char [] t = tt.toCharArray();
    for (int i = 0; i < s.length; i++)
        one[s[i] - 'a']++;

```

```

for (int i = 0; i < t.length; i++)
    two[t[i] - 'a']++;
for (int i = 0; i < 26; i++)
    if (one[i] != two[i])
        return false;
return true;
}
private int dfs(int i, int j, int k) { // k: length dp[i][j][len]: 这个 dp 的设计还是比较难想的!
    if (dp[i][j][k] != 0) return dp[i][j][k];
    String a = s.substring(i, i+k), b = t.substring(j, j+k);
    if (a.equals(b)) return dp[i][j][k] = 1;
    if (!idCheck(a, b)) return dp[i][j][k] = -1;
    for (int l = 1; l < k; l++) {
        if (dfs(i, j, l) == 1 && dfs(i+l, j+l, k-l) == 1)
            return dp[i][j][k] = 1;
        if (dfs(i, j+k-l, l) == 1 && dfs(i+l, j, k-l) == 1)
            return dp[i][j][k] = 1;
    }
    return dp[i][j][k] = -1;
}
int[][][] dp;
String s, t;
int n;
public boolean isScramble(String s, String t) {
    this.s = s;
    this.t = t;
    n = s.length();
    if (s.equals(t)) return true;
    if (!idCheck(s, t)) return false;
    dp = new int[n][n][n+1];
    return dfs(0, 0, s.length()) == 1;
}

```

3. 动态规划（区间 DP）当然，这道题也可以用动态规划 Dynamic Programming，根据以往的经验来说，根字符串有关的题十有八九可以用 DP 来做，那么难点就在于如何找出状态转移方程。

其实有了上述「记忆化搜索」方案之后，我们就已经可以直接忽略原问题，将其改成「动态规划」了。

根据「dfs 方法的几个可变入参」作为「状态定义的几个维度」，根据「dfs 方法的返回值」作为「具体的状态值」。

我们可以得到状态定义  $f[i][j][len]$ ：

$f[i][j][len]f[i][j][len]$  代表  $s_1s_1$  从  $i$  开始， $s_2s_2$  从  $j$  开始，后面长度为  $len$  的字符是否能形成「扰乱字符串」（互为翻转）。

状态转移方程其实就是翻译我们「记忆化搜索」中的 dfs 主要逻辑部分：

```

// 对应了「s1 的 [0,i] & [i,n]」匹配「s2 的 [0,i] & [i,n]」
if (dfs(i, j, k) && dfs(i + k, j + k, len - k)) {
    cache[i][j][len] = Y;
    return true;
}
// 对应了「s1 的 [0,i] & [i,n]」匹配「s2 的 [n-i,n] & [0,n-i]」
if (dfs(i, j + len - k, k) && dfs(i + k, j, len - k)) {
    cache[i][j][len] = Y;
    return true;
}

```

从状态定义上，我们就不难发现这是一个「区间 DP」问题，区间长度大的状态值可以由区间长度小的状态值递推而来。而且由于本身我们在「记忆化搜索」里面就是从小到大枚举  $len$ ，因此这里也需要先将  $len$  这层循环提前，确保我们转移  $f[i][j][len]$  时所需要的状态都已经被计算好。

这道题看起来是比较复杂的，如果用 brute force，每次做切割，然后递归求解，是一个非多项式的复杂度，一般来说这不是面试官想要的答案。

这其实是一道三维动态规划的题目，我们提出维护量  $res[i][j][n]$ ，其中  $i$  是  $s_1$  的起始字符， $j$  是  $s_2$  的起始字符，而  $n$  是当前的字符串长度， $res[i][j][len]$  表示的是以  $i$  和  $j$  分别为  $s_1$  和  $s_2$  起点的长度为  $len$  的字符串是不是互为 scramble。

有了维护量我们接下来看看递推式，也就是怎么根据历史信息来得到  $res[i][j][len]$ 。判断这个是不是满足，其实我们首先是把当前  $s_1[i \dots i+len-1]$  字符串劈一刀分成两部分，然后分两种情况：

第一种是左边和  $s_2[j \dots j+len-1]$  左边部分是不是 scramble，以及右边和  $s_2[j+1 \dots j+len-1]$  右边部分是不是 scramble；第二种情况是左边和  $s_2[j \dots j+len-1]$  右边部分是不是 scramble，以及右边和  $s_2[j+1 \dots j+len-1]$  左边部分是不是 scramble。如果以上两种情况有一种成立，说明  $s_1[i \dots i+len-1]$  和  $s_2[j \dots j+len-1]$  是 scramble 的。而对于判断这些左右部分是不是 scramble 我们是有历史信息的，因为长度小于  $n$  的所有情况我们都在前面求解过了（也就是长度是最外层循环）。

上面说的是劈一刀的情况，对于  $s_1[i \dots i+len-1]$  我们有  $len-1$  种劈法，在这些劈法中只要有一种成立，那么两个串就是 scramble 的。

总结起来递推式是  $\text{res}[i][j][\text{len}] = \text{||} (\text{res}[i][j][k] \&\& \text{res}[i+k][j+k][\text{len}-k]) \text{ || } \text{res}[i][j+\text{len}-k][k] \&\& \text{res}[i+k][j][\text{len}-k])$  对于所有  $1 <= k < \text{len}$ , 也就是对于所有  $\text{len}-1$  种劈法的结果求或运算。因为信息都是计算过的, 对于每种劈法只需要常量操作即可完成, 因此求解递推式是需要  $O(\text{len})$  (因为  $\text{len}-1$  种劈法)。

如此总时间复杂度因为是三维动态规划, 需要三层循环, 加上每一步需要线行时间求解递推式, 所以是  $O(n^4)$ 。虽然已经比较高了, 但是至少不是指数量级的, 动态规划还是有很大本事的, 空间复杂度是  $O(n^3)$ 。

时间复杂度:  $O(n^4)$

空间复杂度:  $O(n^3)$

```
public boolean isScramble(String s, String t) {
    int n = s.length();
    if (s.equals(t)) return true;
    boolean[][][] dp = new boolean[n][n][n+1];
    for (int len = 1; len <= n; len++)
        for (int i = 0; i+len <= n; i++)
            for (int j = 0; j+len <= n; j++) {
                if (len == 1) {
                    dp[i][j][len] = s.charAt(i) == t.charAt(j);
                    continue;
                }
                for (int k = 1; k < len; k++)
                    if (dp[i][j][k] && dp[i+k][j+k][len-k] || dp[i][j+len-k][k] && dp[i+k][j][len-k])
                        dp[i][j][len] = true;
            }
    return dp[0][0][n];
}
```

### 2.0.3 2065. Maximum Path Quality of a Graph - Hard 记忆化搜索 + 重复遍历

There is an undirected graph with  $n$  nodes numbered from 0 to  $n - 1$  (inclusive). You are given a 0-indexed integer array values where  $\text{values}[i]$  is the value of the  $i$ th node. You are also given a 0-indexed 2D integer array edges, where each  $\text{edges}[j] = [uj, vj, time_j]$  indicates that there is an undirected edge between the nodes  $uj$  and  $vj$ , and it takes  $time_j$  seconds to travel between the two nodes. Finally, you are given an integer  $\text{maxTime}$ .

A valid path in the graph is any path that starts at node 0, ends at node 0, and takes at most  $\text{maxTime}$  seconds to complete. You may visit the same node multiple times. The quality of a valid path is the sum of the values of the unique nodes visited in the path (each node's value is added at most once to the sum).

Return the maximum quality of a valid path.

Note: There are at most four edges connected to each node.

#### 1. 解题思路与分析

A straightforward idea is to try all paths from node 0 and calculate the max path quality for paths also end with node 0.

One **optimization** we can add is: once we cannot return back to node 0, we stop. The `min_time` required from any node to node 0 can be pre-computed using Dijkstra algorithm.

**Time complexity:**  $O(4^{10})$

**Note the constraints:**  $10 \leq \text{time}_j$ ,  $\text{maxTime} \leq 100$  and There are at most four edges connected to each node..

It means the max levels of dfs search is 10, and at each level we have maximum of 4 neighbouring nodes to try.

So the time complexity is:  $O(4^{10})$ .

```
private int[] dijkstra() {
    int[] ans = new int[n];
    Arrays.fill(ans, Integer.MAX_VALUE);
    ans[0] = 0;
    // boolean[] vis = new boolean[n]; // 因为可以重复遍历, 要允许它重复遍历
    // vis[idx] = true; // 因为可以重复遍历, 要允许它重复遍历
    // Queue<int> q = new LinkedList<int>();
    Queue<int> q = new PriorityQueue<int>((a, b) ->a[1] - b[1]);
    q.offer(new int[] {0, 0});
    while (!q.isEmpty()) {
        int[] cur = q.poll();
        if (cur[1] > ans[cur[0]]) continue;
        for (int[] nei : adj.get(cur[0]))
            if (nei[0] == cur[0]) continue; // 因为可以重复遍历, 要允许它重复遍历
            if (nei[1] + cur[1] < ans[nei[0]]) {
                ans[nei[0]] = nei[1] + cur[1];
                // if (!vis[nei[0]]) {
                q.offer(new int[] {nei[0], ans[nei[0]]});
                // vis[nei[0]] = true;
            }
    }
    return ans;
}
```

```

private void dfs(int idx, int avaTime, int[] t, int[] v, Set<Integer> vis) {
    if (idx == 0) {
        int cur = 0;
        for (Integer node : vis)
            cur += v[node];
        ans = Math.max(ans, cur);
    }
    for (int[] nei : adj.get(idx))
        if (t[nei[0]] + nei[1] <= avaTime) { // boolean added = vis.add(nei[0]);
            dfs(nei[0], avaTime - nei[1], t, v, vis);
            if (added)
                vis.remove(nei[0]);
        }
}
int[] time;
List<List<int[]>> adj = new ArrayList<>();
int n, ans = 0;
public int maximalPathQuality(int[] values, int[][] edges, int maxTime) {
    n = values.length;
    for (int i = 0; i < n; i++)
        adj.add(new ArrayList<>());
    for (int[] e : edges) {
        adj.get(e[0]).add(new int[] {e[1], e[2]});
        adj.get(e[1]).add(new int[] {e[0], e[2]});
    }
    time = dijkstra();
    Set<Integer> si = new HashSet<>();
    si.add(0);
    dfs(0, maxTime, time, values, si);
    return ans;
}

```

## 2.0.4 1397. Find All Good Strings - Hard 记忆化搜索

Given the strings  $s_1$  and  $s_2$  of size  $n$  and the string  $evil$ , return the number of good strings.

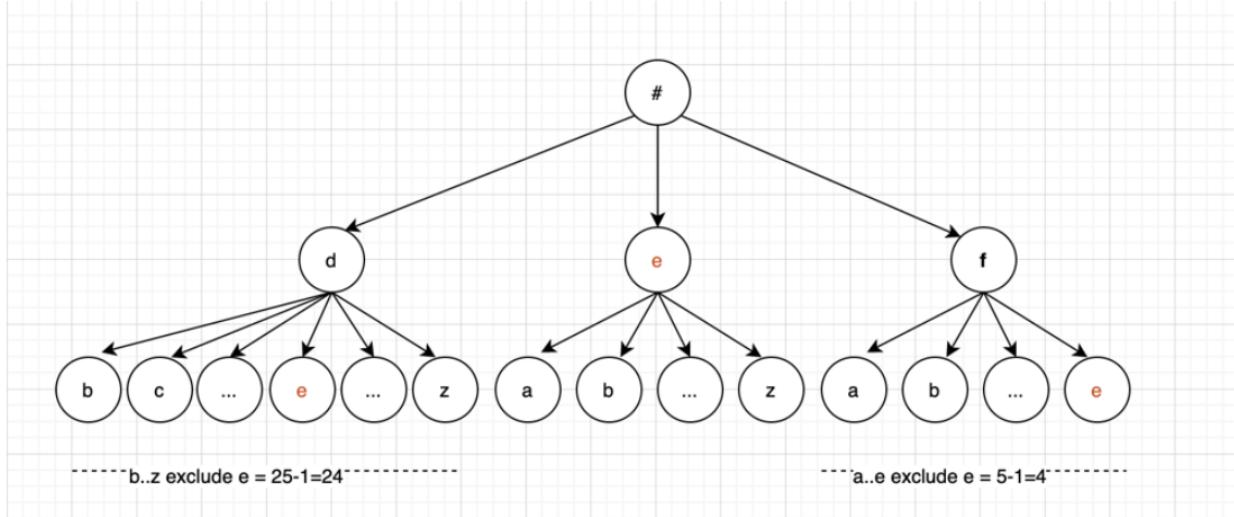
A good string has size  $n$ , it is alphabetically greater than or equal to  $s_1$ , it is alphabetically smaller than or equal to  $s_2$ , and it does not contain the string  $evil$  as a substring. Since the answer can be a huge number, return this modulo  $10^9 + 7$ .

### 1. 解题思路与分析: 记忆化搜索

**Input:**  $n = 2, s_1 = "db", s_2 = "fe", evil = "e"$

**Output:**  $24+4=28$

**Explanation:**



- Complexity

Time:  $O(n \cdot m \cdot 2^2 \cdot 26)$ , where  $n \leq 500$ ,  $m \leq 50$  is length of  $evil$

Space:  $O(m \cdot n \cdot 2^2 \cdot 2)$

```

private int[] computeLongestPrefixSuffix (char[] s) { // 这个要再理解一下: 重中之重
    int n = s.length;
    int[] lps = new int[n];
    for (int i = 1, j = 0; i < n; i++) {
        while (j > 0 && s[i] != s[j]) j = lps[j-1]; // 转向它 j 的前一位字符 (在 j-1 下标) 所指向的匹配位置 lps[j-1]
        if (s[i] == s[j]) lps[i] = ++j; // 同时增加两个的下标
    }
}

```

```

    }
    return lps;
}
private int getKey(int i, int j, boolean l, boolean r) { // bits occupied: i 9, j 6, l 1, r 1
    // 9 bits store n ( $2^9=512$ ), 6 bits for m ( $2^6=64$ ), 1 bit for b1, 1 bit for b2
    return (i << 8) | (j << 2) | ((l ? 1 : 0) << 1) | (r ? 1 : 0); // 这是一个压缩空间存 key 的聪明技巧
}
private int dfs(int n, int i, int evilMatched, boolean leftBound, boolean rightBound) {
    if (evilMatched == e.length) return 0; // matched evil string, no good
    if (i == n) return 1; // DIDN'T match evil string, great
    int key = getKey(i, evilMatched, leftBound, rightBound); // state: represented by <= 17 bits integer
    if (dp[key] > 0) return dp[key];
    char from = leftBound ? s[i] : 'a';
    char to = rightBound ? t[i] : 'z';
    int ans = 0;
    for (char c = from; c <= to; c++) {
        int j = evilMatched; // j means the next match between current string (end at char `c`) and `evil` string
        while (j > 0 && e[j] != c) j = lps[j-1]; // 向左回溯寻找 match 字符 c 的上一个位置 ?
        if (c == e[j]) j++;
        ans += dfs(n, i+1, j, leftBound && (c == from), rightBound && (c == to));
        ans %= mod;
    }
    return dp[key] = ans;
}
int mod = (int)1e9 + 7;
char[] s, t, e;
int[] dp, lps;
public int findGoodStrings(int n, String s1, String s2, String evil) {
    dp = new int[1 << 17]; // Need total 17 bits, according to data limits
    s = s1.toCharArray();
    t = s2.toCharArray();
    e = evil.toCharArray();
    lps = computeLongestPrefixSuffix(e);
    return dfs(n, 0, 0, true, true);
}
}

```

## 2. 动态规划: 数位 DP + KMP todo: 改天把这个补上

- <https://leetcode-cn.com/problems/find-all-good-strings/solution/shu-wei-dp-kmp-by-qodjf/>
- <https://www.cnblogs.com/wenruo/p/12616985.html>
- <https://www.codeleading.com/article/42703213478/>
- <https://leetcode-cn.com/problems/find-all-good-strings/solution/shu-wei-dp-kmpqian-zhui-shu-zu-jav/>
- <https://leetcode-cn.com/problems/find-all-good-strings/solution/kmpshang-de-dpc-by-zhu-mang-4/>

之前做的题大部分是关于数字的数位 dp, 而现在要的就是字符串的数位 dp。

设  $d p [p o s][s t a t s][b o u n d]$   $dp[pos][stats][bound]$  为数位 dp 的数组, 其中  $p o s$  表示第 pos 个位置的字符总共有多少个,  $s t a t s$  表示的是匹配  $e v i l$  的状态, 即能够匹配到  $e v i l$  数组的位置。 $b o u n d$  表示此时能够选择字符的范围, 即当前字符选择的时候是否有限制。

对于  $b o u n d$  我们用四个数字表示

0 00	表示此时的字符选择是没有限制的, 即可以选择的范围为 $a   z$
1 11	表示此时的字符选择是有下限的, 所以选择的范围是 $s 1 [ p o s ]   z$
2 22	表示此时的字符选择是有上限的, 所以可以选择的范围是 $a   s 2 [ p o s ]$
3 33	表示此时的字符既有上限又有下限。这种情况只有当 $s 1 [ p o s ]   s 2 [ p o s ]$

而对于  $s t a t s$  表示匹配  $e v i l$  的状态, 由于  $e v i l$  的长度最长为 50, 所以可以生成字符串  $e v i l$  的  $n e x t$  数组。那么当匹配不成立的时候, 就可以直接进行跳转。

设一个记忆数组  $mem[e\_pos][n\_char]$  表示当匹配  $e v i l$  的位置为  $e\_pos$  时, 下一个字符为  $n\_char$  时, 可以跳转的位置, 因为在整个搜索的过程中, 可能需要多次调用这个数组, 而这个数组大小为  $mem \backslash footnote\{DEFINITION NOT FOUND.\} \backslash textsuperscript{,}\}$ , <sup>1</sup>  $\{DEFINITION NOT FOUND.\}$ , 因此没必要每次都计算。

对于如何生成  $next$  的数组, 小伙伴们可以去搜索与 KMP 算法相关的博客查看。

## 2.0.5 2060. Check if an Original String Exists Given Two Encoded Strings - Hard dfs 记忆化搜索

An original string, consisting of lowercase English letters, can be encoded by the following steps:

Arbitrarily split it into a sequence of some number of non-empty substrings. Arbitrarily choose some elements (possibly none) of the sequence, and replace each with its length (as a numeric string). Concatenate the sequence as the encoded string. For example, one way to encode an original string "abcdefghijklmnp" might be:

<sup>1</sup>{

Split it as a sequence: ["ab", "cdefghijklmn", "o", "p"]. Choose the second and third elements to be replaced by their lengths, respectively. The sequence becomes ["ab", "12", "1", "p"]. Concatenate the elements of the sequence to get the encoded string: "ab121p". Given two encoded strings s1 and s2, consisting of lowercase English letters and digits 1-9 (inclusive), return true if there exists an original string that could be encoded as both s1 and s2. Otherwise, return false.

Note: The test cases are generated such that the number of consecutive digits in s1 and s2 does not exceed 3.

### 1. 解题思路与分析- (这里需要再好好总结一下)

- solution is straight forward we have 2 pointer in each string

```

1.consider the easy case, they all character, we compare s1.charAt(i) == s2.charAt(j)
2.digit case, we get a number from s1, we can calculate the number s1 has, (descripton said less than 1000),
   we can pass this value compare with number from s2 name it diff
3.character case if we still has remaing diff to spend passed from our parents,
   so we can use one dollar a day, one diff one position dfs(i + 1, j, diff - 1)
4.terminating condition, if both reach the end and diff == 0

public boolean possiblyEquals(String ss, String tt) {
    m = ss.length(); N = 1000;
    n = tt.length();
    s = ss.toCharArray();
    t = tt.toCharArray();
    dp = new Boolean[m+1][n+1][2001]; // dp[i][j][diff] means if s1[i:] truncated by <diff> characters if diff > 0
    // and s2[j:] truncated by <-diff> characters if diff < 0 are equal
}
Boolean[][][] dp;
char[] s, t;
int m, n, N;
private boolean dfs(int i, int j, int k) { // k: dif
    if (i == m && j == n) return k == 0;
    if (dp[i][j][k+N] != null) return dp[i][j][k+N];
    if (i < m && j < n && k == 0 && s[i] == t[j] && dfs(i+1, j+1, 0)) // Literal matching on s1[i] and s2[j]
        return dp[i][j][N] = true;
    if (i < m && !Character.isDigit(s[i]) && k > 0 && dfs(i+1, j, k-1)) // Literal matching on s1[i]
        return dp[i][j][k+N] = true;
    if (j < n && !Character.isDigit(t[j]) && k < 0 && dfs(i, j+1, k+1)) // Literal matching on s2[j]
        return dp[i][j][k+N] = true;
    for (int x = i, val = 0; x < m && Character.isDigit(s[x]); x++) { // Wildcard matching on s1[i]
        val = val * 10 + s[x] - '0';
        if (dfs(x+1, j, k-val)) return dp[i][j][k+N] = true;
    }
    for (int x = j, val = 0; x < n && Character.isDigit(t[x]); x++) { // Wildcard matching on s2[j]
        val = val * 10 + t[x] - '0';
        if (dfs(i, x+1, k+val)) return dp[i][j][k+N] = true;
    }
    return dp[i][j][k+N] = false;
}

```

## 2.0.6 638. Shopping Offers - Medium 记忆化搜索 or 背包动态规划

In LeetCode Store, there are n items to sell. Each item has a price. However, there are some special offers, and a special offer consists of one or more different kinds of items with a sale price.

You are given an integer array price where price[i] is the price of the ith item, and an integer array needs where needs[i] is the number of pieces of the ith item you want to buy.

You are also given an array special where special[i] is of size n + 1 where special[i][j] is the number of pieces of the jth item in the ith offer and special[i][n] (i.e., the last integer in the array) is the price of the ith offer.

Return the lowest price you have to pay for exactly certain items as given, where you could make optimal use of the special offers. You are not allowed to buy more items than you want, even if that would lower the overall price. You could use any of the special offers as many times as you want.

### 1. 解题思路与分析: 记忆化搜索 List 作 key

```

// 在 java 里, List 的哈希方式, 是将其哈希成开头加个 1 后的 31 进制整数, 例如对于列表 [ 1 , 2 , 3 ], java 会将其哈希成 31 进制下的 1123, 也就是哈希成
// 1*31^3 + 1 * 32^2 + 2 * 31 + 3 = 30817, 而 List 里判断是否 equals, 是逐个比较列表里的值, 如果值全相等就返回 true。
// 所以在记忆化的时候, 可以直接把 key 设为是 List 类型的。
public int shoppingOffers(List<Integer> p, List<List<Integer>> of, List<Integer> need) {
    n = p.size();
    return dfs(p, of, need);
}
Map<List<Integer>, Integer> dp = new HashMap<>();
int n;
private int dfs(List<Integer> p, List<List<Integer>> of, List<Integer> need) {
    if (dp.containsKey(need)) return dp.get(need); // 首先检查记忆
    int ans = 0;
    for (int i = 0; i < n; i++)
        ans += p.get(i) * need.get(i);
    if (ans == 0) return ans; // ? 这里就不需要记忆了?
    for (List<Integer> cur : of) {

```

```

        if (cur.get(cur.size()-1) >= ans) continue; // >
    List<Integer> newNeed = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        if (cur.get(i) > need.get(i)) break;
        newNeed.add(need.get(i) - cur.get(i));
    }
    if (newNeed.size() == need.size()) // 即当前礼包合法
        ans = Math.min(ans, cur.get(cur.size()-1) + dfs(p, of, newNeed));
}
dp.put(need, ans);
return ans;
}

```

## 2. 解题思路与分析: 背包动态规划 todo bug fix

当然此题也可以用背包的递推公式来做记忆化搜索，并且使用状态压缩记录每个物品有多少个。由于最多有 6 66 种物品，每个物品最多 6 66 个，所以可以用一个  $6 \times 3 \times 36$  位二进制数来表示每个物品有多少个。

```

public int shoppingOffers(List<Integer> price, List<List<Integer>> special, List<Integer> needs) { // todo: rest 6 test cases bug to be fixed
    n = price.size();
    int[][] dp = new int[special.size() + 1][1 << (n * 3)]; // 开一个记忆化数组, dp[i][s] 表示如果只考虑前 i 个套餐的话, 要买到 s 这个状态, 至少需要多少
    System.out.println("(1 << n*3): " + (1 << n*3));
    for (int [] row : dp) Arrays.fill(row, -1); // 先初始化为-1
    int target = 0; // 求一下 needs 代表的状态, 这里 target 的最低 3 位表示的是下标是 0 的商品要买多少个, 以此类推
    for (int i = needs.size() - 1; i >= 0; i--)
        target = (target << 3) + needs.get(i);
    System.out.println("Integer.toBinaryString(target): " + Integer.toBinaryString(target));
    return dfs(special.size(), target, special, price, dp);
}
int n;
// 返回的是, 如果只考虑前 count 个套餐的话, 要达到 state 这个状态的最小花费 (当然单买也是考虑的)
private int dfs(int count, int state, List<List<Integer>> special, List<Integer> price, int[][] dp) {
    // System.out.println("state: " + state);
    // if (state >= (1 << n*3)) return 0;
    if (dp[count][state] != -1) return dp[count][state]; // 如果之前已经算出来过, 则直接返回
    if (count == 0) { // 如果一个套餐都不考虑, 那就是全单买
        dp[count][state] = 0;
        // 求一下每个商品买多少个, 然后累加一下花费
        for (int i = 0; i < n; i++) {
            int c = state >> i * 3 & 7;
            dp[count][state] += c * price.get(i);
        }
        return dp[count][state];
    }
    dp[count][state] = dfs(count - 1, state, special, price, dp); // 考虑不选第 count 个套餐的情况
    List<Integer> sp = special.get(count - 1); // 考虑选第 count 个套餐的情况
    int nextState = 0; // 存一下考虑完当前套餐后的需求状态
    for (int i = n - 1; i >= 0; i--) { // 逆序遍历是为了方便 nextState 的计算
        int c = state >> i * 3 & 7;
        // System.out.println("c: " + c);
        if (c < sp.get(i)) { // 小了, 说明当前套餐是不能选的, 标记为-1 并退出循环
            nextState = -1;
            break;
        }
        // System.out.println("c - sp.get(i): " + (c - sp.get(i)));
        nextState = (nextState << 3) + c - sp.get(i);
        // System.out.println("Integer.toBinaryString(nextState): " + Integer.toBinaryString(nextState));
    }
    if (nextState != -1) // 如果当前套餐能选, 再算一下选了当前套餐的情况下最小花费
        dp[count][state] = Math.min(dp[count][state], sp.get(n) + dfs(count, nextState, special, price, dp));
    return dp[count][state];
}

```

## 2.0.7 474. Ones and Zeroes - Medium

You are given an array of binary strings strs and two integers m and n.

Return the size of the largest subset of strs such that there are at most m 0's and n 1's in the subset.

A set x is a subset of a set y if all elements of x are also elements of y.

### 1. 解题思路与分析

```

public int findMaxForm(String[] strs, int m, int p) { // m: 0 p: 1 dfs 记忆化搜索
    n = strs.length;
    int [] one = new int [n]; // cnt 1s
    int [] two = new int [n]; // cnt 0s
    int cntOne = 0, cntZero = 0;
    for (int i = 0; i < n; i++) {
        String cur = strs[i];
        cntOne = 0;
        cntZero = 0;
        for (char c : cur.toCharArray()) {
            if (c == '0') cntZero++;
            else cntOne++;
        }
        one[i] = cntOne;
    }
}

```

```

        two[i] = cntZero;
    }
    List<String> l = new ArrayList<>();
    cnt = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        if (one[i] > p || two[i] > m) continue;
        l.add(strs[i]);
        cnt.add(new int [] {two[i], one[i]});
    }
    dp = new int [l.size()][m+1][p+1];
    return dfs(l, 0, m, p);
}
int [][] dp;
List<int []> cnt;
int n;
private int dfs(List<String> l, int idx, int i, int j) {
    if (idx >= l.size() || i < 0 || j < 0) return 0;
    if (i == 0 && j == 0) return 0;
    if (dp[idx][i][j] > 0) return dp[idx][i][j];
    int x = i - cnt.get(idx)[0], y = j - cnt.get(idx)[1];
    int ans = x >= 0 && y >= 0 ? 1 : 0;
    return dp[idx][i][j] = Math.max(ans + dfs(l, idx+1, x, y), dfs(l, idx+1, i, j));
}

```

## 2. 解题思路与分析: DP

DP 的写法要熟悉起来

```

public int findMaxForm(String[] strs, int m, int n) {
    int [][] dp = new int [m+1][n+1];
    dp[0][0] = 0;
    for (String s : strs) {
        int one = 0, zero = 0;
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '0') ++ zero;
            else ++one;
        }
        for (int i = m; i >= zero; i--)
            for (int j = n; j >= one; j--)
                dp[i][j] = Math.max(dp[i][j], dp[i-zero][j-one] + 1);
    }
    return dp[m][n];
}

```

## 2.0.8 1900. The Earliest and Latest Rounds Where Players Compete - Hard

There is a tournament where  $n$  players are participating. The players are standing in a single row and are numbered from 1 to  $n$  based on their initial standing position (player 1 is the first player in the row, player 2 is the second player in the row, etc.).

The tournament consists of multiple rounds (starting from round number 1). In each round, the  $i$ th player from the front of the row competes against the  $i$ th player from the end of the row, and the winner advances to the next round. When the number of players is odd for the current round, the player in the middle automatically advances to the next round.

For example, if the row consists of players 1, 2, 4, 6, 7 Player 1 competes against player 7. Player 2 competes against player 6. Player 4 automatically advances to the next round. After each round is over, the winners are lined back up in the row based on the original ordering assigned to them initially (ascending order).

The players numbered `firstPlayer` and `secondPlayer` are the best in the tournament. They can win against any other player before they compete against each other. If any two other players compete against each other, either of them might win, and thus you may choose the outcome of this round.

Given the integers  $n$ , `firstPlayer`, and `secondPlayer`, return an integer array containing two values, the earliest possible round number and the latest possible round number in which these two players will compete against each other, respectively.

### 1. 解题思路与分析: 分析本质不同的站位情况 + 记忆化搜索

本题思维难度较大。其中的有些技巧可能在其它的题目中很少出现。

读者在第一次阅读本题解时, 可以多去思考「怎么做」, 而尽量不要去思考「为什么要这么做」。

#### (a) 思路与算法

我们可以用  $F(n, f, s)F(n,f,s)$  表示还剩余  $n$  个人, 并且两名最佳运动员分别是一排中从左往右数的第  $f$  和  $s$  名运动员时, 他们比拼的最早回合数。

同理, 我们用  $G(n, f, s)G(n,f,s)$  表示他们比拼的最晚回合数。

那么如何进行状态转移呢?

只考虑本质不同的站位情况

如果我们单纯地用  $F(n, f, s)F(n, f, s)$  来进行状态转移，会使得设计出的算法和编写出的代码都相当复杂。例如我们需要考虑 ff 是在左侧（即从前往后数）、中间（即轮空）还是右侧（即从后往前数），对于 ss 也需要考虑那么多情况，这样状态转移方程就相当麻烦。

我们可以考虑分析出本质不同的站位情况，得到下面的表格：

	s 在左侧	s 在中间	s 在右侧
f 在左侧	保持不变	保持不变	保持不变
f 在中间	等价于「f 在左侧， s 在中间」	不存在这种情况	等价于「f 在左侧， s 在中间」
f 在右侧	等价于「f 在左侧， s 在右侧」	等价于「f 在左侧， s 在中间」	等价于「f 在左侧， s 在左侧」

其正确性在于：

- $F(n, f, s) = F(n, s, f)$  恒成立。即我们交换两名最佳运动员的位置，结果不会发生变化；
- $F(n, f, s) = F(n, n+1-s, n+1-f)$  恒成立。因为我们会让从前往后数的第  $i$  名运动员与从后往前数的第  $i$  名运动员进行比拼，那么我们将所有的运动员看成一个整体，整体翻转一下，结果同样不会发生变化。

我们使用这两条变换规则，就可以保证在  $F(n, f, s)F(n, f, s)$  中，ff 一定小于 ss，那么 ff 一定在左侧，而 ss 可以在左侧、中间或者右侧。这样我们就将原本的 88 种情况减少到了 33 种情况。

对于  $G(n, f, s)G(n, f, s)$ ，其做法是完全相同的。

### 状态转移方程的设计

既然我们知道了  $f$  一定在左侧，那么我们就可以根据  $s$  在左侧、中间还是右侧，分别设计状态转移方程了。

f-1	f	s-f-1	s	
-----	---	-------	---	--

如果  $s$  在左侧，如上图所示：

- $f$  左侧有  $f - 1$  名运动员，它们会与右侧对应的运动员进行比拼，因此剩下  $[0, f - 1]$  名运动员；
- $f$  与  $s$  中间有  $s - f - 1$  名运动员，它们会与右侧对应的运动员进行比拼，因此剩下  $[0, s - f - 1]$  名运动员。

如果  $f - 1$  名运动员中剩下了  $i$  名，而  $s - f - 1$  名运动员中剩下了  $j$  名，那么在下一回合中，两名最佳运动员分别位于位置  $i + 1$  和位置  $i + j + 2$ ，而剩余的运动员总数为  $\lfloor \frac{n+1}{2} \rfloor$ ，其中  $\lfloor x \rfloor$  表示对  $x$  向下取整。因此我们可以得到状态转移方程：

$$F(n, f, s) = \left( \min_{i \in [0, f-1], j \in [0, s-f-1]} F\left(\lfloor \frac{n+1}{2} \rfloor, i+1, i+j+2\right) \right) + 1$$

f-1	f	s-f-1	s	
-----	---	-------	---	--

如果  $s$  在中间，如上图所示，我们可以发现状态转移方程与  $s$  在左侧的情况是完全相同的。

如果  $s$  在右侧，那么情况会较为麻烦，会有三种情况：

- 最简单的情况就是  $f$  和  $s$  恰好比拼，即  $f + s = n + 1$ ，那么  $F(n, f, s) = 1$ ；
- 此外，设这一回合与  $s$  比拼的是  $s' = n + 1 - s$ ，那么  $f < s'$  是一种情况， $f > s'$  是另一种情况。

f-1	f	s'-f-1	s'	?	?	s	
-----	---	--------	----	---	---	---	--

然而我们可以知道，根据类似上一节「本质不同的站位情况」的分析，我们将  $f$  变为  $n + 1 - s$ ， $s$  变为  $n + 1 - f$ ，这样  $f$  仍然小于  $s$ ，并且  $f$  也小于  $s'$  了。因此我们只需要考虑  $f < s'$  的情况，如上图所示：

- $f$  左侧有  $f - 1$  名运动员，它们会与右侧对应的运动员进行比拼，因此剩下  $[0, f - 1]$  名运动员；
- $f$  与  $s'$  中间有  $s' - f - 1$  名运动员，它们会与右侧对应的运动员进行比拼，因此剩下  $[0, s' - f - 1]$  名运动员；
- $s'$  一定会输给  $s$ ；
- $s'$  与  $s$  中间有  $n - 2s'$  名运动员。如果  $n - 2s'$  是偶数，那么他们两两之间比拼，剩下  $\frac{n - 2s'}{2}$  名运动员；如果  $n - 2s'$  是奇数，那么其中一人轮空，剩余两两之间比拼，剩下  $\frac{n - 2s' + 1}{2}$  名运动员。因此，无论  $n - 2s'$  是奇数还是偶数， $s'$  与  $s$  中间一定会有  $\lfloor \frac{n - 2s' + 1}{2} \rfloor$  名运动员。

如果  $f - 1$  名运动员中剩下了  $i$  名，而  $s' - f - 1$  名运动员中剩下了  $j$  名，那么在下一回合中，两名最佳运动员分别位于位置  $i + 1$  和位置  $i + j + \lfloor \frac{n - 2s' + 1}{2} \rfloor + 2$ 。因此我们可以得到状态转移方程：

$$F(n, f, s) = \left( \min_{i \in [0, f-1], j \in [0, s'-f-1]} F\left(\lfloor \frac{n+1}{2} \rfloor, i+1, i+j+\lfloor \frac{n-2s'+1}{2} \rfloor+2\right) \right) + 1$$

这样我们就得到了所有关于  $F$  的状态转移方程。而关于  $G$  的状态转移方程，我们只需要把所有的 min 改为 max 即可。

#### (b) 细节

在「本质不同的站位情况」一节中，我们提到了两种变换规则。那么我们具体应当在  $n, f, s$  满足什么关系（而不是抽象的「左侧」「中间」「右侧」）时使用其中的哪些规则呢？

这里有很多种设计方法，我们介绍一种较为简单的，题解代码中使用的方法：

首先我们使用自顶向下的记忆化搜索代替动态规划进行状态转移，这样写更加简洁直观，并且无需考虑状态的求值顺序；

记忆化搜索的入口为  $F(n, \text{firstPlayer}, \text{secondPlayer})$ 。我们在开始记忆化搜索之前，先通过变换规则  $F(n, f, s) = F(n, s, f)$  使得  $\text{firstPlayer}$  一定小于  $\text{secondPlayer}$ ，这样一来，由于另一条变换规则  $F(n, f, s) = F(n, n+1-s, n+1-f)$  不会改变  $ff$  与  $ss$  间的大小关系，因此在接下来的记忆化搜索中， $f < s$  是恒成立的，我们也就无需使用变换规则  $F(n, f, s) = F(n, s, f)$  了；

在之前表格中，我们需要变换的情况有 55 种，分别是：「 $ff$  在中间， $ss$  在左侧」「 $ff$  在中间， $ss$  在右侧」「 $ff$  在右侧， $ss$  在左侧」「 $ff$  在右侧， $ss$  在中间」「 $ff$  在右侧， $ss$  在右侧」。由于我们已经保证了  $f < s$  恒成立，因此这 55 种情况中只剩下 22 种是需要处理的，即：「 $ff$  在中间， $ss$  在右侧」和「 $ff$  在右侧， $ss$  在右侧」。此外，我们在「状态转移方程的设计」一节中还发现了一种需要处理的情况，即「 $ff$  在左侧， $ss$  在右侧，并且  $f > s' = n+1-s$ 」。

那么这 33 种情况是否可以统一呢？对于最后一种情况，我们有  $f+s > n+1$ ，而「 $ff$  在中间， $ss$  在右侧」和「 $ff$  在右侧， $ss$  在右侧」也恰好满足  $f+s > n+1$ ，并且所有不需要变换的情况都不满足  $f+s > n+1$ 。因此我们只需要在  $f+s > n+1$  时，使用一次变换规则  $F(n, f, s) = F(n, n+1-s, n+1-f)$  就行了。

```
// 3 种情况是否可以统一呢？
// 对于最后一种情况，我们有 f+s > n+1
// 而「ff 在中间，ss 在右侧」和「ff 在右侧，ss 在右侧」也恰好满足 f+s > n+1
// 并且所有不需要变换的情况都不满足 f+s > n+1
// 因此我们只需要在 f+s > n+1 时，使用一次变换规则 F(n, f, s) = F(n, n+1-s, n+1-f) 就行
public int [] earliestAndLatest(int n, int firstPlayer, int secondPlayer) {
    min = new int [n+1][n+1][n+1];
    max = new int [n+1][n+1][n+1];
    if (firstPlayer < secondPlayer) return dfs(n, firstPlayer, secondPlayer);
    return dfs(n, secondPlayer, firstPlayer);
}
int [][][], min, max;
private int [] dfs(int n, int f, int s) { // f: firstPlayer, // s: secondPlayer
    if (min[n][f][s] > 0) return new int [] {min[n][f][s], max[n][f][s]};
    if (f + s == n+1) return new int [] {1, 1}; // 刚好第一轮就可以碰上
    if (f + s > n + 1) { // 三种特殊情况的变换：
        int [] res = dfs(n, n+1 - s, n+1 - f);
        min[n][f][s] = res[0];
        max[n][f][s] = res[1];
        return res;
    }
    int glbMin = Integer.MAX_VALUE, glbMax = 0;
    int half = (n+1) >> 1;
    if (s <= half) // // 在左侧或者中间
        for (int i = 0; i < f; i++) // f 左侧人数
            for (int j = 0; j < s-f; j++) { // f 和 s 之间的人数
                int [] res = dfs(half, i+1, i+j+2);
                glbMin = Math.min(glbMin, res[0]);
                glbMax = Math.max(glbMax, res[1]);
            }
    else { // s 在右侧
        int s_prime = n + 1 - s;
        int mid = (n - 2 * s_prime + 1) / 2;
        for (int i = 0; i < f; i++)
            for (int j = 0; j < s_prime - f; j++) { // s': n+1-s
                int [] res = dfs(half, i+1, i+j+2 + mid);
                // int [] res = dfs(half, i+1, i+j+2+(s*2-n-1)/2);
                glbMin = Math.min(glbMin, res[0]);
                glbMax = Math.max(glbMax, res[1]);
            }
    }
    min[n][f][s] = glbMin + 1;
    max[n][f][s] = glbMax + 1;
    return new int [] {glbMin + 1, glbMax + 1};
}
```

## 2.0.9 488. Zuma Game - Hard

You are playing a variation of the game Zuma.

In this variation of Zuma, there is a single row of colored balls on a board, where each ball can be colored red 'R', yellow 'Y', blue 'B', green 'G', or white 'W'. You also have several colored balls in your hand.

Your goal is to clear all of the balls from the board. On each turn:

Pick any ball from your hand and insert it in between two balls in the row or on either end of the row. If there is a group of three or more consecutive balls of the same color, remove the group of balls from the board. If this removal causes more groups of three or more of the same color to form, then continue removing each group until there are none left. If there are no more balls on the board, then you win the game. Repeat this process until you either win or do not have any more balls in your hand. Given a string board, representing the row of balls on the board, and a string hand, representing the balls in your hand, return the minimum number of balls you have to insert to clear all the balls from the board. If you cannot clear all the balls from the board using the balls in your hand, return -1.

### 1. 解题思路与分析: bfs 广度优先搜索 todo

```
public int findMinStep(String board, String hand) {
    char[] arr = hand.toCharArray();
    Arrays.sort(arr);
```

```

hand = new String(arr);
// 初始化用队列维护的状态队列：其中的三个元素分别为桌面球状态、手中球状态和回合数
Queue<State> queue = new ArrayDeque<State>();
queue.offer(new State(board, hand, 0));
// 初始化用哈希集合维护的已访问过的状态
Set<String> visited = new HashSet<String>();
visited.add(board + " " + hand);
while (!queue.isEmpty()) {
    State state = queue.poll();
    String curBoard = state.board;
    String curHand = state.hand;
    int step = state.step;
    for (int i = 0; i <= curBoard.length(); ++i) {
        for (int j = 0; j < curHand.length(); ++j) {
            // 第 1 个剪枝条件：当前球的颜色和上一个球的颜色相同
            if (j > 0 && curHand.charAt(j) == curHand.charAt(j - 1))
                continue;
            // 第 2 个剪枝条件：只在连续相同颜色的球的开头位置插入新球
            if (i > 0 && curBoard.charAt(i - 1) == curHand.charAt(j))
                continue;
            // 第 3 个剪枝条件：只在以下两种情况放置新球
            boolean choose = false;
            // - 第 1 种情况：当前球颜色与后面的球的颜色相同
            if (i < curBoard.length() && curBoard.charAt(i) == curHand.charAt(j))
                choose = true;
            // - 第 2 种情况：当前后颜色相同且与当前颜色不同时候放置球
            if (i > 0 && i < curBoard.length() && curBoard.charAt(i - 1) == curBoard.charAt(i) && curBoard.charAt(i - 1) != curHand.charAt(j))
                choose = true;
            if (choose) {
                String newBoard = clean(curBoard.substring(0, i) + curHand.charAt(j) + curBoard.substring(i));
                String newHand = curHand.substring(0, j) + curHand.substring(j + 1);
                if (newBoard.length() == 0) return step + 1;
                String str = newBoard + " " + newHand;
                if (visited.add(str))
                    queue.offer(new State(newBoard, newHand, step + 1));
            }
        }
    }
}
return -1;
}

public String clean(String s) {
    StringBuffer sb = new StringBuffer();
    Deque<Character> letterStack = new ArrayDeque<Character>();
    Deque<Integer> countStack = new ArrayDeque<Integer>();
    for (int i = 0; i < s.length(); ++i) {
        char c = s.charAt(i);
        while (!letterStack.isEmpty() && c != letterStack.peek() && countStack.peek() >= 3) {
            letterStack.pop();
            countStack.pop();
        }
        if (letterStack.isEmpty() || c != letterStack.peek()) {
            letterStack.push(c);
            countStack.push(1);
        } else countStack.push(countStack.pop() + 1);
    }
    if (!countStack.isEmpty() && countStack.peek() >= 3) {
        letterStack.pop();
        countStack.pop();
    }
    while (!letterStack.isEmpty()) {
        char letter = letterStack.pop();
        int count = countStack.pop();
        for (int i = 0; i < count; ++i)
            sb.append(letter);
    }
    sb.reverse();
    return sb.toString();
}

class State {
    String board;
    String hand;
    int step;
    public State(String board, String hand, int step) {
        this.board = board;
        this.hand = hand;
        this.step = step;
    }
}

```

## 2. 解题思路与分析: dfs 记忆化搜索

```

Map<String, Integer> dp = new HashMap<String, Integer>();
public int findMinStep(String b, String h) {
    char [] s = h.toCharArray();
    Arrays.sort(s);
    h = new String(s);
    int ans = dfs(b, h);
    return ans <= 5 ? ans : -1;
}

```

```

}

private int dfs(String b, String h) {
    if (b.length() == 0) return 0;
    String key = b + "_" + h;
    if (dp.containsKey(key)) return dp.get(key);
    int ans = 6;
    for (int j = 0; j < h.length(); j++) {
        if (j > 0 && h.charAt(j) == h.charAt(j - 1)) continue; // 第 1 个剪枝条件：当前球的颜色和上一个球的颜色相同
        for (int i = 0; i <= b.length(); ++i) {
            if (i > 0 && b.charAt(i - 1) == h.charAt(j)) continue; // 第 2 个剪枝条件：只在连续相同颜色的球的开头位置插入新球
            boolean choose = false; // 第 3 个剪枝条件：只在以下两种情况放置新球
            // - 第 1 种情况：当前球颜色与后面的球的颜色相同
            if (i < b.length() && b.charAt(i) == h.charAt(j))
                choose = true;
            // - 第 2 种情况：当前后颜色相同且与当前颜色不同时候放置球（在两个相同着色的中间放置不同着色的球？）
            if (i > 0 && i < b.length() && b.charAt(i - 1) == b.charAt(i) && b.charAt(i - 1) != h.charAt(j))
                choose = true;
            if (choose) {
                String newB = clean(b.substring(0, i) + h.charAt(j) + b.substring(i));
                String newH = h.substring(0, j) + h.substring(j + 1);
                ans = Math.min(ans, dfs(newB, newH) + 1);
            }
        }
    }
    dp.put(key, ans);
    return dp.get(key);
}

public String clean(String t) {
    Deque<Character> charSt = new ArrayDeque<Character>();
    Deque<Integer> cntSt = new ArrayDeque<Integer>();
    StringBuffer sb = new StringBuffer();
    char [] s = t.toCharArray();
    for (int i = 0; i < s.length; ++i) {
        char c = s[i];
        while (!charSt.isEmpty() && c != charSt.peek() && cntSt.peek() >= 3) { // 把能消除的先消除掉
            charSt.pop();
            cntSt.pop();
        }
        if (charSt.isEmpty() || c != charSt.peek()) { // 要么栈空了，要么字符不同，入栈 (2 个)
            charSt.push(c);
            cntSt.push(1); // 记数为 1
        } else // 栈顶有相同的字符，但计数不够 3，入栈，加数
            cntSt.push(cntSt.pop() + 1);
    }
    if (!cntSt.isEmpty() && cntSt.peek() >= 3) {
        charSt.pop();
        cntSt.pop();
    }
    while (!charSt.isEmpty()) {
        char letter = charSt.pop();
        int count = cntSt.pop();
        sb.append(String.valueOf(letter).repeat(count));
        // for (int i = 0; i < count; ++i) sb.append(letter);
    }
    sb.reverse();
    return sb.toString();
}

class St {
    String b, h;
    int cnt;
    public St(String bd, String hd, int cnt) {
        this.b = bd;
        this.h = hd;
        this.cnt = cnt;
    }
}
}

```



# Chapter 3

## Segment Tree 与 Binary Index Tree 线段树与 树状数组

线段树 (segment tree)，顾名思义，是用来存放给定区间 (segment, or interval) 内对应信息的一种数据结构。与树状数组 (binary indexed tree) 相似，线段树也用来处理数组相应的区间查询 (range query) 和元素更新 (update) 操作。与树状数组不同的是，线段树不止可以适用于区间求和的查询，也可以进行区间最大值，区间最小值 (Range Minimum/Maximum Query problem) 或者区间异或值的查询。

对于树状数组，线段树进行更新 (update) 的操作为  $O(\log n)$ ，进行区间查询 (range query) 的操作也为  $O(\log n)$ 。

### 3.0.1 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.

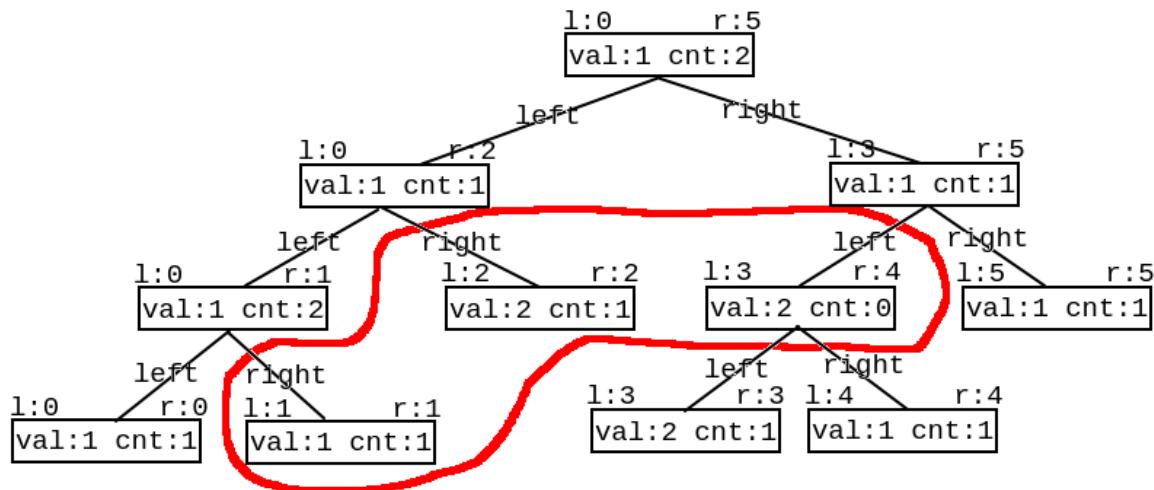
#### 1. 解题思路与分析

- <https://www.cnblogs.com/slowbirdoflsh/p/11381565.html> 思路比较清晰

数组: [1, 1, 2, 2, 1, 1]

0 1 2 3 4 5

使用线段树查询 query(1, 4, 3)



```
Map<Integer, List<Integer>> idx; // idx 存储数组出现元素种类 以及该元素下标索引
Node root; // 线段树的根节点
```

```

int key = 0, cnt = 0; // key 所查找的区域众数; count 所查找的区域众数出现次数,
public MajorityChecker(int[] a) {
    idx = new HashMap<>(); // idx 存储数组出现元素种类 以及该元素下标索引
    for (int i = 0; i < a.length; i++)
        idx.computeIfAbsent(a[i], z -> new ArrayList<>()).add(i);
    root = buildTree(a, 0, a.length-1);
}
public int query(int left, int right, int threshold) {
    key = 0; cnt = 0; // 初始化 所查询众数 key 及辅助判断的计数 cnt
    searchTree(root, left, right); // 查询线段树
    // 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
    if (key == 0 || idx.get(key).size() < threshold) return -1;
    // 上界 索序数组中 第一个大于 right 的下标
    int r = upper_bound(idx.get(key), right);
    // 下界 索序数组中 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;
}
int upper_bound(List<Integer> list, int v) { // 排序数组中 第一个大于 tar 的下标
    int l = 0, r = list.size();
    while (l < r) {
        int mid = l + (r - l) / 2;
        if (list.get(mid) <= v) l = mid + 1;
        else r = mid;
    }
    return l;
}
int lower_bound(List<Integer> list, int v) { // 排序数组中 第一个大于等于 tar 的下标
    int l = 0, r = list.size();
    while (l < r) {
        int mid = l + (r - l) / 2;
        if (list.get(mid) < v) l = mid + 1;
        else r = mid;
    }
    return l;
}
void searchTree(Node root, int l, int r) {
    if (root == null || l > r) return ;
    if (root.l > r || root.r < l) return ;
    if (root.l >= l && root.r <= r) { // 当查询边界被节点边界覆盖, 该节点就是查询区域
        if (key == root.v) cnt += root.cnt;
        else if (cnt <= root.cnt) {
            key = root.v;
            cnt = root.cnt - cnt;
        } else cnt = cnt - root.cnt;
        return ;
    }
    int mid = (root.l + root.r) / 2; // 这两个查询条件再好好想想 !!!!!!!!!!!!!!!
    if (l <= mid) // root.l <= l <= mid 左节点也可以是查询区域
        searchTree(root.left, l, r);
    if (r >= mid+1) // mid+1 <= r <= root.r 右节点也可以是查询区域
        searchTree(root.right, l, r);
}
Node buildTree(int [] a, int l, int r) {
    if (l > r) return null;
    Node root = new Node(l, r); // 初始一个线段树的根节点
    if (l == r) { // 叶子节点
        root.v = a[l];
        root.cnt = 1;
        return root;
    }
    int mid = (l + r) / 2;
    root.left = buildTree(a, l, mid);
    root.right = buildTree(a, mid+1, r);
    makeRoot(root); // 整合父节点
    return root;
}
void makeRoot(Node r) { // 整合父节点
    if (r == null) return ;
    if (r.left != null) { // 如果该节点有左子节点 该节点的值"先"等于左子节点
        r.v = r.left.v;
        r.cnt = r.left.cnt;
    }
    if (r.right != null) { // 如果该节点还有右子节点 融合父节点和子节点
        if (r.v == r.right.v)
            r.cnt = r.cnt + r.right.cnt;
        else {
            if (r.cnt >= r.right.cnt)
                r.cnt = r.cnt - r.right.cnt;
            else {
                r.v = r.right.v;
                r.cnt = r.right.cnt - r.cnt;
            }
        }
    }
}
class Node {

```

```

    int l, r, v, cnt;
    Node left, right;
    public Node(int l, int r) {
        this.l = l; this.r = r;
        v = 0; cnt = 0;
        left = null; right = null;
    }
}
}

```

### 3.0.2 1825. Finding MK Average - Hard

You are given two integers, m and k, and a stream of integers. You are tasked to implement a data structure that calculates the MKAverage for the stream.

The MKAverage can be calculated using these steps:

If the number of the elements in the stream is less than m you should consider the MKAverage to be -1. Otherwise, copy the last m elements of the stream to a separate container. Remove the smallest k elements and the largest k elements from the container. Calculate the average value for the rest of the elements rounded down to the nearest integer. Implement the MKAverage class:

MKAverage(int m, int k) Initializes the MKAverage object with an empty stream and the two integers m and k.  
void addElement(int num) Inserts a new element num into the stream.  
int calculateMKAverage() Calculates and returns the MKAverage for the current stream rounded down to the nearest integer.

```

// 根据题意需要找到前 k 大的数，又需要求区间和，就自然想到线段树。写起来较不容易出错。
// 维护 2 个线段树数组，一个记录数的个数，一个记录区间值。
// 注意一般线段树中 [s, e] 指固定的区间，这里类似线段数求第 k 小的数，所以 [s, e] 指第 s 小的值到第 e 小的值的区间。
Deque<Integer> q = new ArrayDeque<>(); // 始终维护 m 个数
int [] cnt; // 每个元素出现的次数
long [] sum; // 累积和
int m, k, n = 100000, N = n * 4 + 1; // 线段树所占用的空间为数组的四倍大小
public MKAverage(int m, int k) {
    cnt = new int [N];
    sum = new long [N];
    this.m = m;
    this.k = k;
}
public void addElement(int num) {
    if (q.size() == m) {
        int v = q.pollFirst();
        insert(1, 0, n, v, -1); // 当删除掉一个元素的时候，需要更新线段树中的和
    }
    insert(1, 0, n, num, 1);
    q.offerLast(num);
}
public int calculateMKAverage() {
    if (q.size() < m) return -1;
    int bgn = k + 1, end = m - k; // idx: 1 - based
    return (int)(query(1, 0, n, bgn, end) / (m - 2 * k));
}
void insert(int idx, int l, int r, int v, long d) { // d:
    cnt[idx] += d;
    sum[idx] += d * v;
    if (l == r) return ;
    int m = l + (r - l) / 2;
    if (v <= m)
        insert(idx << 1, l, m, v, d); // 向左子树查询
    else insert(idx << 1 | 1, m+1, r, v, d); // 向右子树查询
}
long query(int idx, int l, int r, int bgn, int end) { // 线段中第 bgn 个到第 end 个
    if (l == r) { // 起始和结束最多出现 2 次此情况 ?
        int c = end - bgn + 1;
        return (long)c * l; //
    } else if (cnt[idx] == end - bgn + 1)
        return sum[idx];
    else {
        int m = l + (r - l) / 2;
        int cl = cnt[idx << 1]; // left child cnt
        // int cr = cnt[idx << 1 | 1]; // left child cnt
        if (cl >= end) // 搜索 左子树
            return query(idx << 1, l, m, bgn, end);
        else if (cl >= bgn) // 搜索 左右子树
            return query(idx << 1, l, m, bgn, cl) + query(idx << 1 | 1, m+1, r, 1, end - cl);
        else // cl < bgn, 搜索 右子树
            return query(idx << 1 | 1, m+1, r, bgn - cl, end - cl);
    }
}

```

#### 1. 解题思路与分析: 三个 TreeMap, 自定义 TreeMap

```

CusTreeMap [] ms;
Deque<Integer> q;
int m, k, n;
public MKAverage(int m, int k) {

```

```

this.m = m;
this.k = k;
q = new ArrayDeque<>();
if (m - 2 * k > 0) {
    n = 3;
    ms = new CusTreeMap[n];
    ms[1] = new CusTreeMap(m - 2 * k);
} else {
    n = 2;
    ms = new CusTreeMap[n];
}
ms[0] = new CusTreeMap(k);
ms[n-1] = new CusTreeMap(k);
}
// 删除 num, 结果总是使 mapList 的小、中、大三个 treemap 依次填充。(先保证最小的 TreeMap 填充、再保证中间的 TreeMap 填充、最后是最大的填充)
private void removeElement(int num) {
    boolean removed = false;
    for (int i = 0; i < n; i++) {
        if (!removed)
            removed = ms[i].remove(num);
        else { // 将后现一两个图中的最小元素向第一个图中挪动一个数值
            Integer minK = ms[i].pollFirst();
            if (minK == null) break;
            ms[i-1].add(minK);
        }
    }
}
public void addElement(int num) {
    if (q.size() == m) {
        int v = q.pollFirst();
        removeElement(v);
    }
    q.offerLast(num);
    Integer vToAdd = num;
    for (int i = 0; i < n && vToAdd != null; i++)
        vToAdd = ms[i].add(vToAdd); // 记得这里返回的是：如果图中已有 k 个元素，扔出来的最大键
}
public int calculateMKAverage() {
    if (q.size() < m || n < 3) return -1;
    return ms[1].avg();
}
class CusTreeMap {
    TreeMap<Integer, Integer> m;
    final int capacity;
    int size, sum;
    public CusTreeMap(int capacity) {
        m = new TreeMap<>();
        this.capacity = capacity;
    }
    public boolean remove(int key) {
        if (m.containsKey(key)) {
            m.put(key, m.get(key)-1);
            if (m.get(key) == 0) m.remove(key);
            sum -= key;
            size--;
            return true;
        }
        return false;
    }
    public Integer pollFirst() { // return key
        if (m.size() > 0) {
            int k = m.firstKey();
            // m.remove(k); // BUG: 你也不能用原始的 TreeMap.remove(), 因为它会移走所有的重复 (如果这个元素存在重复的话)
            remove(k); // !!!
            return k; // 这里没有自动更新和
            // return m.firstKey(); // BUG: 这里并没有真正移走这个元素，只是返回了第个元素的键
        }
        return null;
    }
    public Integer add(int key) { // 返回的是删除掉元素的键
        m.put(key, m.getOrDefault(key, 0) + 1); // 这里新填入的元素是否是最后一个元素，关系不大
        size++;
        sum += key;
        if (size > capacity) {
            int last = m.lastKey();
            m.put(last, m.get(last)-1);
            if (m.get(last) == 0) m.remove(last);
            sum -= last;
            size--;
            return last;
        }
        return null;
    }
    public int avg() {
        return sum / size;
    }
}

```

## 2. 解题思路与分析: 树状数组

- 数状数组的解法：另外第一次看到别人二分 + 树状数组也能求前 k 大的值。

```

// We can have a queue to maintain m elements
// Use two Fenwick tree, 1 for count and 1 for prefix sum
// Do 2 times binary search for the first k elements and the last k elements by using the count from our first fenwick tree
// We can get the sum by subtracting the sum of first k elements and sum of last k element by using our second fenwick tree
Queue<Integer> q = new LinkedList<>();
Fenwick fone, ftwo;
int [] cnt = new int [100010];
long sum = 0;
int m,k;
public MKAverage(int m, int k) {
    this.m = m;
    this.k = k;
    long A [] = new long [100010];
    long B [] = new long [100010];
    fone = new Fenwick(A);
    ftwo = new Fenwick(B);
}
public void addElement(int num) {
    q.add(num);
    sum += num;
    fone.update(num, 1);
    ftwo.update(num, num);
    cnt[num]++;
}
public int calculateMKAverage() {
    if (q.size() < m) return -1;
    while (q.size() > m) {
        int cur = q.poll();
        cnt[cur]--;
        sum -= cur;
        fone.update(cur, -1);
        ftwo.update(cur, -cur);
    }
    // binary search for the first k (there may be duplicated)
    int l = 0, r = cnt.length-1;
    int i = -1, j = -1; // pos1, pos2
    while (l <= r) { // 二分查找总计数
        int m = (r + l) / 2;
        long count = fone.sumRange(0, m);
        if (count >= k) {
            i = m;
            r = m - 1;
        } else l = m+1;
    }
    // binary search for the last k (there may be duplicated)
    l = 0;
    r = cnt.length-1;
    while (l <= r) {
        int m = l + (r-l)/2;
        long count = fone.sumRange(m, cnt.length-1);
        if (count >= k) {
            j = m;
            l = m + 1;
        } else r = m-1;
    }
    long sum1 = ftwo.sumRange(0, i);
    long sum2 = ftwo.sumRange(j, cnt.length-1);
    long cnt1 = fone.sumRange(0, i);
    long cnt2 = fone.sumRange(j, cnt.length-1);
    if (cnt1 > k)
        sum1 -= i*(cnt1-k);
    if (cnt2 > k)
        sum2 -= j*(cnt2-k);
    long remain = sum - sum1 - sum2; // 总和，减去两边最小最大各 K 个数的和
    return (int)(remain / (m-2*k));
}
class Fenwick {
    long tree []; //1-index based
    long A [];
    long arr[];
    public Fenwick(long [] A) {
        this.A = A;
        arr = new long [A.length];
        tree = new long [A.length + 1];
    }
    public void update(int i, int v) {
        arr[i] += v;
        i++;
        while (i < tree.length) {
            tree[i] += v;
            i += (i & -i); // 这是的原理细节再回去复习一下
        }
    }
    public long sumRange(int i, int j) {
        return pre(j+1)-pre(i);
    }
    public long pre(int i) {
}

```

```

        long sum = 0;
        while (i > 0) {
            sum += tree[i];
            i -= (i & -i);
        }
        return sum;
    }
}

```

- 其它比较有兴趣以的 BST 二叉树的解法，改天补起来

### 3.0.3 315. Count of Smaller Numbers After Self - Hard

You are given an integer array nums and you have to return a new counts array. The counts array has the property where counts[i] is the number of smaller elements to the right of nums[i].

#### 1. 解题思路与分析: 二分查找的插入排序

```

public List<Integer> countSmaller(int[] a) { // O(NlogN) 插入排序
    int n = a.length;
    List<Integer> ans = new ArrayList<>();
    List<Integer> list = new ArrayList<>(); // 新建一个 list, 用于排序
    int [] tmp = new int [n]; // 为了提高效率, 新建一个数组型的返回结果
    for (int i = n-1; i >= 0; i--) {
        int v = a[i]; // 将当前数字插入到新建 list 中, 使用二分查找找到插入位置
        int l = 0, r = list.size()-1; // l: left; r: right 从排好序的 list 中二分查找正确的插入位置
        while (l <= r) {
            int m = l + (r - l) / 2;
            if (v <= list.get(m)) r = m-1;
            else l = m + 1;
        }
        list.add(l, v); // 将当前数字插入到相应位置, 保证 list 升序排列
        tmp[i] = l; // 当前位置前所有数字均小于当前数字, 将个数加入返回结果
    }
    for (Integer v : tmp) ans.add(v);
    return ans;
}

```

#### 2. 解题思路与分析: 数状数组

- 官方题解:<https://leetcode-cn.com/problems/count-of-smaller-numbers-after-self/solution/ji-suan-you-hua-shu-zu/>

```

private int[] c;
private int[] a; // 离散化、去重复 后的数组
public List<Integer> countSmaller(int[] nums) {
    List<Integer> ans = new ArrayList<Integer>();
    discretization(nums);
    init(nums.length + 5);
    for (int i = nums.length - 1; i >= 0; --i) {
        int id = getId(nums[i]);
        ans.add(query(id - 1));
        update(id);
    }
    Collections.reverse(ans);
    return ans;
}
private void init(int length) {
    c = new int[length];
    Arrays.fill(c, 0);
}
private int lowBit(int x) {
    return x & (-x);
}
private void update(int pos) {
    while (pos < c.length) {
        c[pos] += 1;
        pos += lowBit(pos);
    }
}
private int query(int pos) {
    int ret = 0;
    while (pos > 0) {
        ret += c[pos];
        pos -= lowBit(pos);
    }
    return ret;
}
private void discretization(int[] nums) { // 离散化、去重复 ?
    Set<Integer> set = new HashSet<Integer>(Arrays.stream(nums).boxed().collect(Collectors.toList()));
    int size = set.size();
    a = new int[size];
    int index = 0;
    for (int num : set) a[index++] = num;
    Arrays.sort(a);
}

```

```

private int getId(int x) {
    return Arrays.binarySearch(a, x) + 1; //
}

```

### 3. 解题思路与分析: 归并排序 todo 补上

## 3.0.4 327. Count of Range Sum - Hard 重点这几个题要好好再理解消化几遍

Given an integer array nums and two integers lower and upper, return the number of range sums that lie in [lower, upper] inclusive.

Range sum S(i, j) is defined as the sum of the elements in nums between indices i and j inclusive, where  $i \leq j$ .

### 1. 解题思路与分析: 分治法

在累积和的基础上, 我们计算有多少个区间的大小落在 [lower, upper] 之间, 一个朴素的算法就是枚举各个区间, 其时间复杂度是  $O(n^2)$ 。一个更好的方法是利用分治法来处理, 即利用归并排序算法将数组分成左右两边, 在合并左右数组之前, 对于左边数组中的每一个元素, 在右边数组找到一个范围, 使得在这个范围中的元素与左边元素构成的区间和落在 [lower, upper] 之间, 即在右边数组中找到两个边界, 设为 m, n, 其中 m 是在右边数组中第一个使得  $\text{sum}[m] - \text{sum}[i] \geqslant \text{lower}$  的位置, 而 n 是第一个使得  $\text{sum}[n] - \text{sum}[i] > \text{upper}$  的位置, 这样  $n-m$  就是与左边元素 i 所构成的位于 [lower, upper] 范围的区间个数。因为左右两边都是已经有序的, 这样就可以避免不必要的比较 (这也是为什么我们能将时间复杂度从  $O(n^2)$  降低到  $O(n\log n)$  的秘诀所在)。

```

public int countRangeSum(int[] a, int lower, int upper) { // 这个 merge sort 的思维很奇特: 二分, O(NlogN)
    int n = a.length;
    long[] sum = new long[n+1];
    for (int i = 1; i <= n; i++)
        sum[i] = sum[i-1] + a[i-1];
    return mergeAnalyse(sum, 0, n+1, lower, upper);
}

int mergeAnalyse(long[] a, int l, int r, int lo, int hi) { // l, r: 寻找 [l, r] 范围内和为 [lower, upper] 的片段的个数
    if (r - l <= 1) return 0;
    int mid = l + (r - l) / 2;
    int m = mid, n = mid, ans = 0;
    ans = mergeAnalyse(a, l, mid, lo, hi) + mergeAnalyse(a, mid, r, lo, hi);
    for (int i = l; i < mid; i++) { // 遍历 [l, r] 的半段长度: pivot 右移, 滑动窗口, 寻找合法窗口 // 通过遍历寻找当前范围内符合要求的个数,
        while (m < r && a[m] - a[i] < lo) m++; // 左端点右移, 直到找到合法 ( $\text{sum} \geqslant \text{lo}$ ) 的解: m 合法
        while (n < r && a[n] - a[i] <= hi) n++; // 右端点右移, 直到右端点右移至不再合法 ( $\text{sum} > \text{hi}$ ), n 不合法
        ans += n - m; // 对于 [l, r] 范围内的当前 i 来说, 满足要求的总个数为 n - m
    }
    Arrays.sort(a, l, r); // 将 [l, r] 片段排序
    return ans;
}

```

### 2. 解题思路与分析: 线段树

- 官方题解:<https://leetcode-cn.com/problems/count-of-range-sum/solution/qu-jian-he-de-ge-shu-by-leetcode/>

```

public int countRangeSum(int[] a, int lo, int hi) {
    int n = a.length;
    long[] preSum = new long[n + 1];
    for (int i = 1; i <= n; i++)
        preSum[i] = preSum[i-1] + a[i-1];
    Set<Long> allNumbers = new TreeSet<Long>();
    for (long x : preSum) {
        allNumbers.add(x);
        allNumbers.add(x - lo);
        allNumbers.add(x - hi);
    }
    // 利用哈希表进行离散化
    Map<Long, Integer> values = new HashMap<Long, Integer>();
    int idx = 0;
    for (long x : allNumbers) {
        values.put(x, idx);
        idx++;
    }
    SegNode root = build(0, values.size() - 1);
    int ret = 0;
    for (long x : preSum) {
        int left = values.get(x - hi), right = values.get(x - lo);
        ret += count(root, left, right);
        insert(root, values.get(x));
    }
    return ret;
}

public SegNode build(int left, int right) {
    SegNode node = new SegNode(left, right);
    if (left == right) {
        return node;
    }
    int mid = (left + right) / 2;
    node.lchild = build(left, mid);
    node.rchild = build(mid + 1, right);
    return node;
}

int count(SegNode node, int left, int right) {
    if (node == null) return 0;
    if (left > right) return 0;
    if (left <= node.left && right >= node.right) return node.sum;
    return count(node.lchild, left, right) + count(node.rchild, left, right);
}

void insert(SegNode node, int index) {
    if (node == null) return;
    if (index < node.left) insert(node.lchild, index);
    else if (index > node.right) insert(node.rchild, index);
    else node.sum++;
}

```

```

        node.rchild = build(mid + 1, right);
        return node;
    }
    public int count(SegNode root, int left, int right) {
        if (left > root.hi || right < root.lo)
            return 0;
        if (left <= root.lo && root.hi <= right)
            return root.add;
        return count(root.lchild, left, right) + count(root.rchild, left, right);
    }
    public void insert(SegNode root, int val) {
        root.add++;
        if (root.lo == root.hi)
            return;
        int mid = (root.lo + root.hi) / 2;
        if (val <= mid)
            insert(root.lchild, val);
        else insert(root.rchild, val);
    }
    class SegNode {
        int lo, hi, add;
        SegNode lchild, rchild;
        public SegNode(int left, int right) {
            lo = left;
            hi = right;
            add = 0;
            lchild = null;
            rchild = null;
        }
    }
}

```

### 3. 解题思路与分析: 动态增加节点的线段树 todo

### 4. 解题思路与分析: 树状数组

```

public int countRangeSum(int[] a, int lower, int upper) { // 树状数组
    int n = a.length;
    long[] preSum = new long[a.length + 1];
    for (int i = 1; i <= n; i++)
        preSum[i] = preSum[i-1] + a[i-1];
    Set<Long> allNumbers = new TreeSet<Long>();
    for (long x : preSum) {
        allNumbers.add(x);
        allNumbers.add(x - lower);
        allNumbers.add(x - upper);
    }
    // 利用哈希表进行离散化
    Map<Long, Integer> values = new HashMap<Long, Integer>();
    int idx = 0;
    for (long x: allNumbers) {
        values.put(x, idx);
        idx++;
    }
    int ret = 0;
    BIT bit = new BIT(values.size());
    for (int i = 0; i < preSum.length; i++) {
        int left = values.get(preSum[i] - upper), right = values.get(preSum[i] - lower);
        ret += bit.query(right + 1) - bit.query(left);
        bit.update(values.get(preSum[i]) + 1, 1);
    }
    return ret;
}
class BIT {
    int [] tree;
    int n;
    public BIT(int n) {
        this.n = n;
        this.tree = new int[n + 1];
    }
    public int lowbit(int x) {
        return x & (-x);
    }
    public void update(int x, int d) {
        while (x <= n) {
            tree[x] += d;
            x += lowbit(x);
        }
    }
    public int query(int x) {
        int ans = 0;
        while (x != 0) {
            ans += tree[x];
            x -= lowbit(x);
        }
        return ans;
    }
}

```

## 5. 解题思路与分析: 平衡二叉搜索树 思路与算法

考虑一棵平衡二叉搜索树。若其节点数量为  $N$ , 则深度为  $O(\log N)$ 。二叉搜索树能够在  $O(\log N)$  的时间内, 对任意给定的值  $\text{val}$ , 查询树中所有小于或等于该值的数量。

因此, 我们可以从左到右扫描前缀和数组。对于  $\text{preSum}[j]$  而言, 首先进行两次查询, 得到区间  $[\text{preSum}[j]\text{-upper}], \text{preSum}[j]\text{-lower}]$  内的整数数量; 随后再将  $\text{preSum}[j]$  插入到平衡树中。

平衡二叉搜索树有多种不同的实现, 最经典的为 AVL 树与红黑树。此外, 在算法竞赛中, 还包括 Treap、SBT 等数据结构。

下面给出基于 Treap 的实现。

```
public int countRangeSum(int[] nums, int lower, int upper) {
    long sum = 0;
    long[] preSum = new long[nums.length + 1];
    for (int i = 0; i < nums.length; ++i) {
        sum += nums[i];
        preSum[i + 1] = sum;
    }
    BalancedTree treap = new BalancedTree();
    int ret = 0;
    for (long x : preSum) {
        long numLeft = treap.lowerBound(x - upper);
        int rankLeft = (numLeft == Long.MAX_VALUE ? (int) (treap.getSize() + 1) : treap.rank(numLeft)[0]);
        long numRight = treap.upperBound(x - lower);
        int rankRight = (numRight == Long.MAX_VALUE ? (int) treap.getSize() : treap.rank(numRight)[0] - 1);
        ret += rankRight - rankLeft + 1;
        treap.insert(x);
    }
    return ret;
}
class BalancedTree {
    private class BalancedNode {
        long val;
        long seed;
        int count;
        int size;
        BalancedNode left;
        BalancedNode right;
        BalancedNode(long val, long seed) {
            this.val = val;
            this.seed = seed;
            this.count = 1;
            this.size = 1;
            this.left = null;
            this.right = null;
        }
        BalancedNode leftRotate() {
            int prevSize = size;
            int currSize = (left != null ? left.size : 0) + (right.left != null ? right.left.size : 0) + count;
            BalancedNode root = right;
            right = root.left;
            root.left = this;
            root.size = prevSize;
            size = currSize;
            return root;
        }
        BalancedNode rightRotate() {
            int prevSize = size;
            int currSize = (right != null ? right.size : 0) + (left.right != null ? left.right.size : 0) + count;
            BalancedNode root = left;
            left = root.right;
            root.right = this;
            root.size = prevSize;
            size = currSize;
            return root;
        }
    }
    private BalancedNode root;
    private int size;
    private Random rand;
    public BalancedTree() {
        this.root = null;
        this.size = 0;
        this.rand = new Random();
    }
    public long getSize() {
        return size;
    }
    public void insert(long x) {
        ++size;
        root = insert(root, x);
    }
    public long lowerBound(long x) {
        BalancedNode node = root;
        long ans = Long.MAX_VALUE;
        while (node != null) {
            if (node.val > x) {
                node = node.left;
            } else {
                ans = node.val;
                node = node.right;
            }
        }
        return ans;
    }
}
```

```

        if (x == node.val) return x;
        if (x < node.val) {
            ans = node.val;
            node = node.left;
        } else node = node.right;
    }
    return ans;
}
public long upperBound(long x) {
    BalancedNode node = root;
    long ans = Long.MAX_VALUE;
    while (node != null) {
        if (x < node.val) {
            ans = node.val;
            node = node.left;
        } else node = node.right;
    }
    return ans;
}
public int[] rank(long x) {
    BalancedNode node = root;
    int ans = 0;
    while (node != null) {
        if (x < node.val) {
            node = node.left;
        } else {
            ans += (node.left != null ? node.left.size : 0) + node.count;
            if (x == node.val) {
                return new int[]{ans - node.count + 1, ans};
            }
            node = node.right;
        }
    }
    return new int[]{Integer.MIN_VALUE, Integer.MAX_VALUE};
}
private BalancedNode insert(BalancedNode node, long x) {
    if (node == null)
        return new BalancedNode(x, rand.nextInt());
    ++node.size;
    if (x < node.val) {
        node.left = insert(node.left, x);
        if (node.left.seed > node.seed)
            node = node.rightRotate();
    } else if (x > node.val) {
        node.right = insert(node.right, x);
        if (node.right.seed > node.seed)
            node = node.leftRotate();
    } else
        ++node.count;
    return node;
}
}

```

### 3.0.5 699. Falling Squares - Hard

There are several squares being dropped onto the X-axis of a 2D plane.

You are given a 2D integer array positions where  $\text{positions}[i] = [\text{left}_i, \text{sideLength}_i]$  represents the  $i$ th square with a side length of  $\text{sideLength}_i$  that is dropped with its left edge aligned with X-coordinate  $\text{left}_i$ .

Each square is dropped one at a time from a height above any landed squares. It then falls downward (negative Y direction) until it either lands on the top side of another square or on the X-axis. A square brushing the left/right side of another square does not count as landing on it. Once it lands, it freezes in place and cannot be moved.

After each square is dropped, you must record the height of the current tallest stack of squares.

Return an integer array ans where  $\text{ans}[i]$  represents the height described above after dropping the  $i$ th square.

- 解题思路与分析:  $O(N^2)$  本能土办法 方块的大小不是固定的, 有可能很大, 但是不管方块再大, 只要有一点点部分搭在其他方块上面, 整个方块都会在上面, 并不会掉下来, 让我们求每落下一个方块后的最大高度。我们知道返回的是每落下一个方块后当前场景中的最大高度, 那么返回的数组的长度就应该和落下方块的个数相同。所以我们可以建立一个 heights 数组, 其中  $\text{heights}[i]$  表示第  $i$  块方块落下后所在的高度, 那么第  $i$  块方块落下后场景的最大高度就是  $[0, i]$  区间内的最大值。那么我们在求出 heights 数组后, 只要不停返回  $[0, i]$  区间内的最大值即可。继续来看, 这道题的难点就是方块重叠的情况, 我们先来想, 如果各个方块不重叠, 那么  $\text{heights}[i]$  的高度就是每个方块自身的高度。一旦重叠了, 就得在已有的基础上再加上自身的高度。那么我们可以采用 brute force 的思想, 对于每一个下落的方块, 我们都去看和后面将要落下的方块有没有重叠, 有的话, 和后面将要落下的方块的位置相比较, 取二者中较大值为后面要落下的方块位置高度  $\text{heights}[j]$ 。判读两个方块是否重叠的方法是如果方块 2 的左边界小于方块 1 的右边界, 并且方块 2 点右边界大于方块 1 点左边界。就拿题目中的例子 1 来举例吧, 第一个下落的方块的范围是  $[1, 3]$ , 长度为 2, 则  $\text{heights}^1=2$ , 然后我们看其和第二个方块  $[2, 5]$  是否重叠, 发现是重叠的, 则  $\text{heights}^2$  更新为 2, 再看第三个方块  $[6, 7]$ , 不重叠, 不更新。然后第二个方块落下, 此时累加高度, 则  $\text{heights}^2=5$ , 再看第三个方块, 不重叠, 不更新。然后第三个方块落下,  $\text{heights}^3=1$ 。此时我们 heights 数组更新好了, 然后我们开始从头遍历, 维护一个当前最大值 curMax, 每次将  $[0, i]$  中最大值加入结果 res 即可,

```

public List<Integer> fallingSquares(int[][] a) {
    List<Integer> ans = new ArrayList<>();
    int n = a.length, max = 0;
    int [] hi = new int [n]; // 表示第 i 块方块落下后所在的高度
    for (int i = 0; i < n; i++) {
        int h = a[i][1], l = a[i][0], r = a[i][0] + h;
        hi[i] += h;
        for (int j = i+1; j < n; j++) {
            int ll = a[j][0], rr = ll + a[j][1];
            // [[6,1],[9,2],[2,4]] 因为不能保证是从左往右延 x 轴顺序掉落，所以加上 l < rr 也很重要 确保不管左右边有交叠
            if (ll < r && rr > l) // 保证 j 在 i 的右边，并且有重叠区域
                hi[j] = Math.max(hi[j], hi[i]);
        }
        max = Math.max(max, hi[i]);
        ans.add(max);
    }
    return ans;
}

```

## 2. 解题思路与分析：线段树 + 离散化

想象  $x$  轴是地面，如果某个方块掉落的过程中遇到了之前的某个方块（擦边而过不算），则该方块会叠到上面。现在给定一个长  $n$  数组  $A$ ， $A[i]$  存了第  $i$  个掉落的方块的信息，其中  $A[i][0]$  表示它的左下角的  $x$  坐标， $A[i][1]$  表示它的边长。要求返回一个长  $n$  数组  $B$ ，使得  $B[i]$  表示在  $A[i]$  掉落之后，当前所有方块的最高点的  $y$  坐标。

思路是线段树 + 离散化。可以将  $x$  坐标离散化，这样可以节省存储空间（离散化的过程其实就是将一个数组  $d$  排序后去重，然后将每个数映射到它的下标。这样在线段树建树的时候，就只需维护  $[0, l_d - 1]$  这个区间的信息就行了，这会极大减少线段树的空间消耗，也从而会减少要做的操作的时间消耗）。具体来说，给定一个将要下落的方块，比如该方块的左端点的  $x$  坐标和右端点的  $x$  坐标分别是  $a$  和  $b$ ，边长是  $c$ ，那么我们需要实现两个操作，第一是查询  $(a, b)$  里的最大值  $M$ （注意这里查询的是开区间  $(a, b)$  的最大值，因为下落的方块擦着另一个方块的边的话，是不会叠上去的），另一个是将  $[a, b]$  里所有值都变成  $M + c$ 。本质上是要求一个数据结构可以查询区间最大值，以及将区间修改为某一值，这可以用线段树 + 懒标记来做到。在离散化之后，为了使得区间  $(a, b)$  非空（注意这里  $a$  和  $b$  都是离散化之后的值，此时  $(a, b) = [a + 1, b - 1]$ ），我们可以在离散化的时候将方块的中点也加入一起做离散化，但是这会导致中点变成非整数，这里将原坐标乘以 2 就行了。

给定一个数学里的二维平面直角坐标系  $xOy$ ，想象有若干正方形方块掉落，题目保证正方形方块的左下端点的坐标是正整数，也保证方块边长是正整数。想象  $x$  轴是地面，如果某个方块掉落的过程中遇到了之前的某个方块（擦边而过不算），则该方块会叠到上面。现在给定一个长  $n$  数组  $A$ ， $A[i]$  存了第  $i$  个掉落的方块的信息，其中  $A[i][0]$  表示它的左下角的  $x$  坐标， $A[i][1]$  表示它的边长。要求返回一个长  $n$  数组  $B$ ，使得  $B[i]$  表示在  $A[i]$  掉落之后，当前所有方块的最高点的  $y$  坐标。

思路是线段树 + 离散化。可以将  $x$  坐标离散化，这样可以节省存储空间（离散化的过程其实就是将一个数组  $d$  排序后去重，然后将每个数映射到它的下标。这样在线段树建树的时候，就只需维护  $[0, l_d - 1]$  这个区间的信息就行了，这会极大减少线段树的空间消耗，也从而会减少要做的操作的时间消耗）。具体来说，给定一个将要下落的方块，比如该方块的左端点的  $x$  坐标和右端点的  $x$  坐标分别是  $a$  和  $b$ ，边长是  $c$ ，那么我们需要实现两个操作，第一是查询  $(a, b)$  里的最大值  $M$ （注意这里查询的是开区间  $(a, b)$  的最大值，因为下落的方块擦着另一个方块的边的话，是不会叠上去的），另一个是将  $[a, b]$  里所有值都变成  $M + c$ 。本质上是要求一个数据结构可以查询区间最大值，以及将区间修改为某一值，这可以用线段树 + 懒标记来做到。在离散化之后，为了使得区间  $(a, b)$  非空（注意这里  $a$  和  $b$  都是离散化之后的值，此时  $(a, b) = [a + 1, b - 1]$ ），我们可以在离散化的时候将方块的中点也加入一起做离散化，但是这会导致中点变成非整数，这里将原坐标乘以 2 就行了。代码如

```

public List<Integer> fallingSquares(int[][] a) { // 需要对数据进行离散化处理，离散化的目的是为了线段树处理起来方便；离散的是 x 轴的横坐标
    List<Integer> x = new ArrayList<>();
    for (int [] v : a) {
        int i = v[0], j = i + v[1];
        x.add(i * 2);
        x.add(j * 2);
        x.add(i + j);
    }
    x = getUniques(x);
    MaxSeg maxSeg = new MaxSeg(x.size());
    List<Integer> ans = new ArrayList<>();
    for (int [] v : a) {
        int i = v[0], j = i + v[1];
        i = getIdxInList(i * 2, x);
        j = getIdxInList(j * 2, x);
        int h = maxSeg.query(1, i+1, j-1);
        maxSeg.update(i * 2, h);
        ans.add(h);
    }
    return ans;
}

```

```

        maxSeg.update(l, i, j, h + v[1]);
        ans.add(maxSeg.query());
    }
    return ans;
}
int getIdxInList(int v, List<Integer> list) { // 找到 x 在离散化之后的值是多少，其实就是要 xs 里 x 的下标，可以二分来找到
    int l = 0, r = list.size()-1;
    while (l < r) {
        int m = l + (r - l) / 2;
        if (list.get(m) >= v) r = m;
        else l = m + 1;
    }
    return l;
}
List<Integer> getUniques(List<Integer> l) {
    l.sort(Integer::compareTo);
    int j = 0; // 返回结果链表的下标 idx
    for (int i = 0; i < l.size(); i++) {
        if (i == 0 || l.get(j-1) != l.get(i))
            l.set(j++, l.get(i));
    }
    return l.subList(0, j);
}
class MaxSeg { // 实现一下带懒标记的线段树：这棵树好强大
    class Node { // v 是 [l, r] 区间最大值，lazy 是懒标记
        int l, r, v, lazy;
        public Node(int l, int r) {
            this.l = l;
            this.r = r;
        }
    }
    Node [] tree;
    public MaxSeg(int n) {
        tree = new Node[n << 2]; // n * 2 * 2
        buildTree(1, 0, n-1); // 下标从 1 开始 自顶向下
    }
    void buildTree(int i, int l, int r) {
        tree[i] = new Node(l, r);
        if (l == r) return;
        int m = l + r >> 1; // (l + r) / 2
        buildTree(i << 1, l, m);
        buildTree(i << 1 | 1, m+1, r);
    }
    void pushUp(int i) { // 自底向上：自左、右叶子节点向顶更新最大值，取左右节点的最大值
        tree[i].v = Math.max(tree[i << 1].v, tree[i << 1 | 1].v);
    }
    void pushDown(int i) { // 懒标记向底、叶子方向推进一层
        int c = tree[i].lazy;
        if (c != 0) { // 打有懒标记
            tree[i].lazy = 0;
            tree[i << 1].v = tree[i << 1 | 1].v = c;
            tree[i << 1].lazy = tree[i << 1 | 1].lazy = c;
        }
    }
    void update(int i, int l, int r, int c) { // 自顶向下传递懒标记，再自底向上更新父节点的值：取左右子节点的最大值
        if (l <= tree[i].l && tree[i].r <= r) { // 任务不需要下发，可以用懒标记懒住
            tree[i].v = tree[i].lazy = c; // 这里 tree[i].v = tree[i].lazy = c : c 是想要更新到的新值 v，用它来更新懒标记和 v 值
            return ;
        }
        pushDown(i); // 任务不得不下发，则先下发给两个孩子
        int m = tree[i].l + tree[i].r >> 1;
        if (l <= m) update(i << 1, l, r, c); // 回归调用，下传更新至左右子节点
        if (m + 1 <= r) update(i << 1 | 1, l, r, c);
        pushUp(i); // 孩子完成了任务，再修改自己的值
    }
    int query(int i, int l, int r) {
        if (l <= tree[i].l && r >= tree[i].r) return tree[i].v;
        pushDown(i);
        int ans = 0, m = tree[i].l + tree[i].r >> 1;
        if (l <= m) ans = Math.max(ans, query(i << 1, l, r));
        if (m + 1 <= r) ans = Math.max(ans, query(i << 1 | 1, l, r));
        return ans;
    }
    int query() {
        return tree[1].v;
    }
}

```

### 3. 解题思路与分析: 超简洁版的线段树，效率奇高

- <http://www.noobyard.com/article/p-sxwzvpgp-nz.html>

- 去找一下原文件中的优化步骤

```

private class Node { // 描述方块以及高度
    int l, r, h, maxR;
    Node left, right;
    public Node(int l, int r, int h, int maxR) {
        this.l = l;
    }
}

```

```

        this.r = r;
        this.h = h;
        this.maxR = maxR;
        this.left = null;
        this.right = null;
    }
}

public List<Integer> fallingSquares(int[][] positions) {
    List<Integer> res = new ArrayList<>(); // 建立返回值
    Node root = null; // 根节点， 默认为零
    int maxH = 0; // 目前最高的高度
    for (int[] position : positions) {
        int l = position[0]; // 左横坐标
        int r = position[0] + position[1]; // 右横坐标
        int e = position[1]; // 边长
        int curH = query(root, l, r); // 目前区间的最高的高度
        root = insert(root, l, r, curH + e);
        maxH = Math.max(maxH, curH + e);
        res.add(maxH);
    }
    return res;
}

private Node insert(Node root, int l, int r, int h) {
    if (root == null) return new Node(l, r, h, r);
    if (l <= root.l)
        root.left = insert(root.left, l, r, h);
    else
        root.right = insert(root.right, l, r, h);
    root.maxR = Math.max(r, root.maxR); // 最终目标是仅仅须要根节点更新 maxR
    return root; // 返回根节点
}

private int query(Node root, int l, int r) {
    // 新节点的左边界大于等于目前的 maxR 的话，直接获得 0，不须要遍历了
    if (root == null || l >= root.maxR) return 0;
    int curH = 0; // 高度
    if (!(r < root.l || root.r <= l)) // 是否跟这个节点相交
        curH = root.h;
    // 剪枝
    curH = Math.max(curH, query(root.left, l, r));
    if (r > root.l)
        curH = Math.max(curH, query(root.right, l, r));
    return curH;
}
}

```

### 3.0.6 1483. Kth Ancestor of a Tree Node - Hard 倍增法 binary lifting

You are given a tree with  $n$  nodes numbered from 0 to  $n - 1$  in the form of a parent array  $\text{parent}$  where  $\text{parent}[i]$  is the parent of  $i$ th node. The root of the tree is node 0. Find the  $k$ th ancestor of a given node.

The  $k$ th ancestor of a tree node is the  $k$ th node in the path from that node to the root node.

Implement the `TreeAncestor` class:

`TreeAncestor(int n, int[] parent)` Initializes the object with the number of nodes in the tree and the parent array.  
`int getKthAncestor(int node, int k)` return the  $k$ th ancestor of the given node  $node$ . If there is no such ancestor, return -1.

#### 1. 解题思路与分析: 倍增 binary lifting

给定一个 $n$ 个节点的树，每个节点编号 $0 \sim n - 1$ ，另外给出每个节点的父亲节点是谁，以数组 $p$ 给出。要求设计一个数据结构可以应答这样的询问，每次询问给出一个节点编号 $u$ 和一个正整数 $k$ ，问 $u$ 的第 $k$ 个祖先是谁（即 $u$ 向上走 $k$ 步是谁）。如果该祖先不存在则返回-1。

可以用倍增法。设 $f[u][k]$ 是节点 $u$ 向上走 $2^k$ 步能走到的节点，那么有

$$f[u][k] = f[f[u][k-1]][k-1]$$

初始条件 $f[u][0] = p[u]$ ，从而可以递推出 $f$ 数组。因为一共有 $n$ 个节点，树最高就是 $n$ ，也就是说向上最多能走 $n - 1$ 步，我们找到最小的 $2^k \geq n - 1$ ，把 $f$ 的第二维开 $k = \lfloor \log_2 n - 1 \rfloor + 1$ 这么长，以保证询问都能被正确应答。接下来考虑如何应答询问，可以用类似快速幂的思想，比方说 $k$ 的二进制表示是 $1101_2$ ，那么 $1101_2 = 2^0 + 2^2 + 2^3$ ，那么可以先求 $u_1 = f[u][0]$ ，这样就跳了 $2^0$ 步，再接着求 $u_2 = f[u_1][2]$ ，这样又跳了 $2^2$ 步，再接着求 $f[u_2][3]$ ，这样就将 $k$ 步全跳完了。中途如果发现要跳的幂次大于了 $f$ 的第二维能取的最大值，或者跳到了-1，则直接返回-1。代码如下：

- 预处理时间复杂度  $O(n\log n)$ ，每次询问时间  $O(\log n)$ ，空间  $O(n\log n)$ 。

```

private int [][] p;
private int log;
public TreeAncestor(int n, int[] parent) {
    log = (int) (Math.log(n - 1) / Math.log(2)) + 1;
    p = new int[n][log];
    for (int i = 0; i < parent.length; i++) // 初始化 p 数组
        p[i][0] = parent[i];
    for (int i = 1; i < log; i++) // 按公式递推 p 数组
        for (int j = 0; j < n; j++)
            if (p[j][i-1] != -1)
                p[j][i] = p[p[j][i-1]][i-1];
            else p[j][i] = -1;
}
public int getKthAncestor(int node, int k) {
    int pow = 0;
    while (k > 0) {
        if (pow >= log || node == -1) return -1;
        if ((k & 1) == 1)
            node = p[node][pow];
        k >>= 1;
        pow++;
    }
    return node;
}

```

## 2. 解题思路与分析

```

Map<Integer, List<Integer>> adj;
int [][] par;
public TreeAncestor(int n, int[] parent) {
    par = new int [n][30]; // 30 , 16: 不能证它是一棵很平衡的二叉树
    adj = new HashMap<>();
    for (int i = 0; i < n; i++) {
        Arrays.fill(par[i], -1);
        adj.put(i, new ArrayList<>());
    }
    for (int i = 0; i < parent.length; i++) {
        if (parent[i] != -1) {
            adj.get(parent[i]).add(i); // 自顶向下: 父 --> 子节点
            par[i][0] = parent[i]; // 每个子节点的第一个父节点 ( $2^0 = 1$ ), 即为父节点 // 自底向上: 子节点:  $2^0$  父节点、 $2^1$  节点、 $2^2$  节点
        }
    }
    dfs(0);
}
public int getKthAncestor(int node, int k) {
    for (int i = 0; k > 0; i++, k >>= 1) // k /= 2
        if ((k & 1) == 1) {
            node = par[node][i];
            if (node < 0) return -1;
        }
    return node;
}
private void dfs(int idx) { // 自顶向下: 从父节点遍历子节点
    for (int i = 1; par[idx][i-1] >= 0; i++) // 穷追塑源: 一直找到整棵树的根节点: 0
        par[idx][i] = par[par[idx][i-1]][i-1]; // 这里多想想
    for (int next : adj.get(idx))
        dfs(next);
}

```

### 3.0.7 236 二叉树的最近公共祖先

### 3.0.8 1505. Minimum Possible Integer After at Most K Adjacent Swaps On Digits - Hard BIT 树状数组

You are given a string num representing the digits of a very large integer and an integer k. You are allowed to swap any two adjacent digits of the integer at most k times.

Return the minimum integer you can obtain also as a string.

## 1. 解题思路与分析

```

public String minInteger(String t, int k) {
    int n = t.length();
    t = " " + t;
    char [] s = t.toCharArray();
    ArrayDeque<Integer> [] q = new ArrayDeque [10];
    for (int i = 1; i <= n; i++) {
        int j = s[i] - '0';
        if (q[j] == null) q[j] = new ArrayDeque<>();
        q[j].offerLast(i);
    }
    BIT bit = new BIT(n);
    StringBuilder sb = new StringBuilder();
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j < 10; j++) { // 从小数值往大数值遍历
            if (q[j] == null || q[j].isEmpty()) continue;

```

```

        int top = q[j].peekFirst(), pos = top + bit.sum(top); // pos 是最优解的位置，最优解的位置是原来的位置加上偏移量
        if (pos - i <= k) {
            k -= pos - i;
            sb.append(j);
            q[j].pollFirst();
            bit.add(1, 1); // 更新 [1, t] 这段的值每个加 1，即向右偏移 1 位。为什么要从 1 开始更新：假装每次都移动到最前端，方便计算？
            bit.add(top, -1);
            break;
        }
    }
    return sb.toString();
}
class BIT { // 开一个树状数组类，维护每个位置的字符的向右的偏移量 ? 向左偏移量
    private int n;
    private int[] a;
    public BIT(int n) {
        this.n = n;
        this.a = new int[n+1];
    }
    public void add(int idx, int v) { // 只有发生偏移，才移动某段区间的值
        while (idx <= n) {
            a[idx] += v;
            idx += lowbit(idx);
        }
    }
    public int sum(int idx) { // 得到以 i 为下标 1-based 的所有子、叶子节点的和，也就是 [1, idx] 的和，1-based
        int ans = 0;
        while (idx > 0) {
            ans += a[idx];
            idx -= lowbit(idx);
        }
        return ans;
    }
    int lowbit(int x) {
        return x & -x;
    }
}

```



# Chapter 4

## 字符串

### 4.1 总结：几种【字符串算法】关键字

- Rabin-Karp Rolling Hash 算法、AC 自动机

### 4.2 KMP 算法

#### 4.2.1 1392. Longest Happy Prefix - Hard : Use Longest Prefix Suffix (KMP-table) or String Hashing.

A string is called a happy prefix if it is a non-empty prefix which is also a suffix (excluding itself).

Given a string s, return the longest happy prefix of s. Return an empty string "" if no such prefix exists.

##### 1. 解题思路与分析: KMP 算法

给定一个字符串，其长度为  $k$  的前缀和长度为  $k$  后缀有可能相等，问最大的  $k$  对应的那个前缀是什么。

题目本质上是求 KMP 算法中的 next 数组 (KMP 有个原始版本和优化版本，这里指的是原始版本。对于原始版本， $n[i]$  表示  $s[0:i-1]$  中最长相等前后缀的长度，规定  $n[0] = -1$ )。

我们考虑  $n[i]$  和  $n[i-1]$  的递推关系。

首先  $n[1] = 0$ ，我们是可以确定的。对于  $n[i-1]$ ，假设  $n[i-1] = t$ ，意味着  $s[0:t-1] = s[i-t-1:i-2]$ ：

1、若  $s[i-1] = s[t]$ ，那么就有  $s[0:t] = s[i-t-1:i-1]$ ，所以  $n[i] \geq t+1$ 。但如果  $n[i] > t+1$  的话，就会导致  $s[0:t+1] = s[i-t-2:i-1]$ ，从而有  $s[0:t] = s[i-t-2:i-2]$ ，这与  $n[i-1] = t$  矛盾了，所以有  $n[i] = t+1$ 。

2、若  $s[i-1] \neq s[t]$ ，根据 next 数组的定义， $s[i-1]$  要继续和  $s[n[t]]$  进行比较，如果不等，则继续和  $s[n[n[t]]]$  进行比较，如此这样下去，直到与  $s[i-1] = s[n^k[t]]$  相等为止，相等后， $n[i] = n^k[t] + 1$ ；若一直都不等，则  $n[i] = n[0] + 1 = -1 + 1 = 0$ ，也是对的。证明与 1 类似。代码如下：

```
public String longestPrefix(String ss) {
    int n = ss.length();
    char [] s = ss.toCharArray();
    int [] lps = new int [n];
    for (int i = 1, j = 0; i < n; i++) {
        while (j > 0 && s[i] != s[j])
            j = lps[j-1];
        if (s[i] == s[j])
            lps[i] = ++j;
    }
    return ss.substring(0, lps[n-1]);
}
```

##### 2. rolling hash

Time complexity: O(n) / worst case: O(n^2)

Space complexity: O(1)

```
public String longestPrefix(String s) { // 容易出错, KMP 写起来比较简单
    int n = s.length(), hashPre = 0, hashSuf = 0;
    int left = 0, right = n-1, pow = 1, maxLen = 0;
    String res = "";
    while (left < n-1) {
        hashPre = hashPre * 31 + s.charAt(left);
```

```

        hashSuf = hashSuf + s.charAt(right)*pow;
        if (hashPre == hashSuf) maxLen = left + 1;
        left++;
        right--;
        pow *= 31;
    }
    return maxLen == 0 ? "" : s.substring(0, maxLen);
}

```

## 4.2.2 214. Shortest Palindrome - Hard KMP 算法

You are given a string  $s$ . You can convert  $s$  to a palindrome by adding characters in front of it.

Return the shortest palindrome you can find by performing this transformation.

### 1. 解题思路与分析

```

public String shortestPalindrome(String s) {
    int n = s.length();
    int base = 131, mod = 1000000007;
    int left = 0, right = 0, mul = 1;
    int best = -1;
    for (int i = 0; i < n; ++i) {
        left = (int) ((long) left * base + s.charAt(i)) % mod;
        right = (int) ((right + (long) mul * s.charAt(i)) % mod);
        if (left == right) best = i;
        mul = (int) ((long) mul * base % mod);
    }
    String add = (best == n - 1 ? "" : s.substring(best + 1));
    StringBuffer ans = new StringBuffer(add).reverse();
    ans.append(s);
    return ans.toString();
}

```

### 2. 解题思路与分析

```

public String shortestPalindrome(String t) {
    // if (t.chars().distinct().count() == 1) return t;
    // if (isPalindrome(t)) return t; // 这种提前的判断也会耽误时间
    int n = t.length(), j = 0;
    char[] s = t.toCharArray();
    for (int i = n-1; i >= 0; i--) // 精髓就在这两行
        if (s[i] == s[j]) j++; // 
    if (j == n) return t;
    String suf = t.substring(j);
    System.out.println("suf: " + suf);
    return new StringBuilder(suf).reverse().toString() + shortestPalindrome(t.substring(0, j)) + suf;
}

```

### 3. 解题思路与分析: KMP 算法

```

public String shortestPalindrome(String t) { // 这个方法比上面的一个方法要慢
    String s = new StringBuilder(t).reverse().toString();
    s = t + "#" + s; // 原串 + "#" + 反转串
    int n = t.length(), m = s.length(), j = 0;
    int[] p = new int[m];
    for (int i = 1; i < m; i++) {
        int k = p[i-1];
        while (k > 0 && s.charAt(k) != s.charAt(i)) k = p[k-1];
        k += (s.charAt(i) == s.charAt(k) ? 1 : 0);
        p[i] = k;
    }
    return s.substring(n+1, 2 * n + 1 - p[m-1]) + t;
}

```

- 官方发布的写法

```

public String shortestPalindrome(String s) {
    int n = s.length();
    int[] fail = new int[n];
    Arrays.fill(fail, -1);
    for (int i = 1; i < n; ++i) {
        int j = fail[i - 1];
        while (j != -1 && s.charAt(j + 1) != s.charAt(i))
            j = fail[j];
        if (s.charAt(j + 1) == s.charAt(i))
            fail[i] = j + 1;
    }
    int best = -1;
    for (int i = n - 1; i >= 0; --i) {
        while (best != -1 && s.charAt(best + 1) != s.charAt(i))
            best = fail[best];
        if (s.charAt(best + 1) == s.charAt(i))
            ++best;
    }
}

```

```

String add = (best == n - 1 ? "" : s.substring(best + 1));
StringBuffer ans = new StringBuffer(add).reverse();
ans.append(s);
return ans.toString();
}

```

### 4.2.3 1910. Remove All Occurrences of a Substring - Medium 可用 KMP 算法

Given two strings  $s$  and  $part$ , perform the following operation on  $s$  until all occurrences of the substring  $part$  are removed:

Find the leftmost occurrence of the substring  $part$  and remove it from  $s$ . Return  $s$  after removing all occurrences of  $part$ .

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

```

public String removeOccurrences(String s, String part) {
    if (!s.contains(part)) return s;
    int n = s.length();
    int m = part.length();
    while (s.contains(part)) {
        int idx = s.indexOf(part);
        s = s.substring(0, idx) + (idx+m-1 == n-1 ? "" : s.substring(idx+m));
    }
    return s;
}

```

- 有人用了 KMP: 字符串匹配可以用 KMP 算法, 由于  $p$  始终不变, 可以先算一下  $p$  的 next 数组, 然后每次从  $s$  中找  $p$  的第一次出现, 删去之, 再重复进行这个过程

```

private int[] buildNext(String s) { // 找与每个位置字符不同的下一个字母的 idx
    int[] next = new int[s.length()];
    for (int i = 0, j = next[0] = -1; i < s.length()-1; ) {
        if (j == -1 || s.charAt(i) == s.charAt(j)) {
            i++;
            j++;
            next[i] = s.charAt(i) != s.charAt(j) ? j : next[j];
        } else j = next[j];
    }
    return next;
}
private int kmp(String s, String p, int[] next) { // 像是夹生饭, 半生不熟的
    for (int i = 0, j = 0; i < s.length(); ) {
        if (j == -1 || s.charAt(i) == p.charAt(j)) {
            i++;
            j++;
        } else j = next[j];
        if (j == p.length()) return i-j;
    }
    return -1;
}
public String removeOccurrences(String s, String part) {
    int[] next = buildNext(part);
    int idx = -1;
    while ((idx = kmp(s, part, next)) != -1)
        s = s.substring(0, idx) + s.substring(idx + part.length());
    return s;
}

```

### 4.2.4 1316. Distinct Echo Substrings - Hard

Return the number of distinct non-empty substrings of  $text$  that can be written as the concatenation of some string with itself (i.e. it can be written as  $a + a$  where  $a$  is some string).

- 解题思路与分析: 枚举我们在  $text$  中枚举位置  $i$  和  $j$ , 若字符串  $text[i:j]$  和  $text[j:j*2-i]$  相等, 那么字符串  $text[i:j*2-i]$  就是一个满足条件的子串, 其中  $text[x:y]$  表示字符串  $text$  中以位置  $x$  开始, 位置  $y$  结束并且不包含位置  $y$  的子串。

由于题目需要求出不同的子串数目, 因此我们还需要使用哈希集合 (HashSet) 对所有满足条件的子串进行去重操作。

```

public int distinctEchoSubstrings(String t) {
    int n = t.length(), j = 0;
    Set<String> ss = new HashSet<>();
    for (int d = 1; d <= n/2; d++) {
        for (int i = 0; i+d <= n-d; i++) {
            if (t.substring(i, i+d).equals(t.substring(i+d, i+d+d)))
                ss.add(t.substring(i, i+d));
        }
    }
    return ss.size();
}

```

## 复杂度分析

- 时间复杂度:  $O(N^3)$ , 其中  $N$  是字符串 `text` 的长度。我们需要两重循环枚举位置 `i` 和 `j`, 时间复杂度为  $O(N^2)$ , 而在比较字符串 `text[i:j]` 和 `text[j:j*2-i]` 时, 最坏的时间复杂度为  $O(N)$ , 并且将字符串加入哈希集合中的时间复杂度也为  $O(N)$ , 因此总时间复杂度为  $O(N^3)$ 。但由于本题的测试数据较弱, 它可以在规定时间内通过所有的测试数据。
- 空间复杂度:  $O(N^2)$  或  $O(N^3)$ , 在最坏情况下, 哈希集合中会存储  $O(N^2)$  个字符串, 如果我们在哈希集合中存储的是字符串视图 (例如 C++ 中的 `std::string_view`) , 那么每个字符串视图使用的空间为  $O(1)$ , 总空间复杂度为  $O(N^2)$ ; 如果我们在哈希集合中存储的是字符串本身 (例如 C++ 中的 `std::string` 和 Python3 中的 `str` ) , 那么每个字符串使用的空间为  $O(N)$ , 总空间复杂度为  $O(N^3)$ 。

## 2. 解题思路与分析: 滚动哈希 + 前缀和

本方法需要一些关于「滚动哈希」或「Rabin-Karp 算法」的预备知识, 其核心是将字符串看成一个  $k$  进制的整数, 其中  $k$  是字符串中可能出现的字符种类, 本题中字符串只包含小写字母, 即  $k = 26$  (也可以取比  $k$  大的整数, 一般来说可以取一个质数, 例如 29 或 31)。这样做的好处是绕开了字符串操作, 将字符串看成整数进行比较, 并可以在常数时间内将字符串加入哈希集合中。

关于「滚动哈希」或「Rabin-Karp 算法」的知识, 可以参考 1044. 最长重复子串的官方题解或使用搜索引擎, 这里对算法本身的流程不再赘述。

### 使用滚动哈希

我们仍然在 `text` 中枚举位置 `i` 和 `j`，并判断字符串 `text[i:j]` 和 `text[j:j*2-i]` 是否相等。但与方法一不同的是，我们比较这两个字符串的哈希值，而并非字符串本身。如果仅使用 Rabin-Karp 算法计算这两个字符串的哈希值，我们仍然需要使用  $O(N)$  的时间，与直接比较字符串的时间复杂度相同。那么我们如何在更短的时间内计算出字符串的哈希值呢？

我们可以使用类似前缀和的方法。记 `prefix[i]` 表示字符串 `text[0:i]` 的哈希值。特别地，`prefix[0]` 的值为 `0`，为空串的哈希值。我们可以在  $O(N)$  的时间内计算出数组 `prefix` 中的所有元素，即：

$$\text{prefix}[i] = \text{prefix}[i - 1] * k + \text{text}[i]$$

当我们得到了前缀和数组 `prefix` 之后，如何计算任意字符串 `text[i:j]` 的哈希值呢？观察 `prefix[i]` 和 `prefix[j]` 这两项，它们的表达式为：

$$\begin{aligned}\text{prefix}[i] &= \text{text}[0] * k^{i-1} + \text{text}[1] * k^{i-2} + \cdots + \text{text}[i-1] \\ \text{prefix}[j] &= \text{text}[0] * k^{j-1} + \text{text}[1] * k^{j-2} + \cdots + \text{text}[j-1]\end{aligned}$$

如果将 `prefix[i]` 乘以  $k^{j-i}$ ，那么等式两侧会变为：

$$k^{j-i} * \text{prefix}[i] = \text{text}[0] * k^{j-1} + \text{text}[1] * k^{j-2} + \cdots + \text{text}[i-1] * k^{j-i}$$

将上式与 `prefix[j]` 的表达式相减，得到：

$$\text{prefix}[j] - k^{j-i} * \text{prefix}[i] = \text{text}[i] * k^{j-i-1} + \cdots + \text{text}[j-1]$$

与 `text[i:j]` 的哈希值相等，因此我们可以在  $O(1)$  的时间计算出任意字符串 `text[i:j]` 的哈希值。在此之前，我们还需要预处理计算出所有 `k` 的次幂，不然在进行乘法操作时，仍然需要超过  $O(1)$  的时间。

注意：上面的所有等式都省略了取模操作（如果你不知道为什么要取模，那么你对「Rabin-Karp 算法」的预备知识还没有掌握透彻），但在实际的代码编写中不能忽略，并且在取模的意义下，做减法时会产生负数。

通用的写法如下所示，它同时考虑了取模和负数（在 C++ 等语言中，还需要注意乘法可能产生的溢出）：

$$\text{hash\_value}(\text{text}[i:j]) = (\text{prefix}[j] - k^{j-i} * \text{prefix}[i] \% \text{mod} + \text{mod}) \% \text{mod}$$

回到我们原来的问题，当我们用字符串 `text[i:j]` 和 `text[j:j*2-i]` 的哈希值代替其本身判断是否相等后，如果两者相等，字符串 `text[i:j*2-i]` 就是一个满足条件的子串。但我们还需要进行去重操作，才能得到最终的答案。

在「判断」这一步中，由于我们只对两个字符串进行比较，因此引入哈希冲突（在下方的注意事项中也有提及）的概率极小。然而在「去重」这一步中，最坏情况下字符串的数量为  $O(N^2)$ ，大量的字符串造成哈希冲突的概率极大。为了减少字符串的数量以降低冲突的概率，我们可以使用 `N` 个哈希集合，分别存放不同长度的字符串，即第 `m` 个哈希集合存放长度为 `m + 1` 的字符串的哈希值。这样每个哈希集合只对某一固定长度的字符串进行去重操作，并且其中最多只有 `N` 个字符串，冲突概率非常低。

```
static final int mod = (int)1e9 + 7;
public int distinctEchoSubstrings(String t) {
    int n = t.length(), ans = 0;
    char[] s = t.toCharArray();
    int base = 31;
    int[] pre = new int[n+1], mul = new int[n+1];
    mul[0] = 1;
    for (int i = 1; i <= n; i++) {
        pre[i] = (int)((long)pre[i-1] * base + s[i-1]) % mod;
        mul[i] = (int)((long)mul[i-1] * base % mod);
    }
    Set<Integer>[] vis = new HashSet[n];
    for (int i = 0; i < n; i++) vis[i] = new HashSet<>();
```

```

    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            int l = j - i;
            if (j + l <= n) {
                int hash_left = gethash(pre, mul, i, j-1);
                if (!vis[l-1].contains(hash_left) && hash_left == gethash(pre, mul, j, j+l-1)) {
                    ++ans;
                    vis[l-1].add(hash_left);
                }
            }
        }
    }
    return ans;
}
private int gethash(int [] pre, int [] mul, int l, int r) {
    return (int)((pre[r+l] - (long)pre[l] * mul[r-l+1] % mod + mod) % mod);
}

```

- 注意事项

由于 Rabin-Karp 算法会将字符串对应的整数值进行取模，那么：

如果字符串 S1 和 S2 对应的整数值 I1 和 I2 不相等，那么 S1 和 S2 一定不相等；

如果字符串 S1 和 S2 对应的整数值 I1 和 I2 相等，并不代表 S1 和 S2 一定相等；

这与实际应用中使用的哈希算法也是一致的，即先判断两个实例的哈希值是否相等，再判断它们本质上是否相等。而在竞赛题目中，由于数据量较少，几乎不会产生哈希冲突，因此我们可以直接用 I1 和 I2 的相等代替 S1 和 S2 的相等，减少时间复杂度。但需要牢记在实际应用中，这样做是不严谨的。

#### 4.2.5 467. Unique Substrings in Wraparound String - Medium Rabin-Karp Rolling Hash 算法

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 findSubstringInWraparoundString(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;
}

```

#### 4.2.6 1044. Longest Duplicate Substring - Hard

Given a string s, consider all duplicated substrings: (contiguous) substrings of s that occur 2 or more times. The occurrences may overlap.

Return any duplicated substring that has the longest possible length. If s does not have a duplicated substring, the answer is "".

```

public static int base = 26; // 256
public static int mod = (1 << 31) - 1;
public static boolean match(String str1, String str2) {
    assert str1.length() == str2.length();
    for (int i = 0; i < str1.length(); i++)
        if (str1.charAt(i) != str2.charAt(i))

```

```

        return false;
    return true;
}

private String search(String s, int v) { // v: substring length
    int n = s.length();
    long hash = 0L, mp = 1L; // to avoid overflow, long long long
    Map<Long, List<Integer>> map = new HashMap<>();
    for (int j = 0; j < v; j++) {
        hash = ((hash * base) + s.charAt(j)) % mod;
        if (j >= 1)
            mp = mp * base % mod; // 先乘好准备好了，准备着备用
    }
    map.computeIfAbsent(hash, k->new ArrayList<>()).add(0);
    for (int i = 1; i+v <= n; i++) {
        hash = ((hash - s.charAt(i-1) * mp % mod + mod) % mod * base % mod + s.charAt(i+v-1)) % mod; // mod
        if (map.containsKey(hash))
            for (int idx : map.get(hash))
                if (match(s.substring(i, i+v), s.substring(idx, idx+v)))
                    return s.substring(i, i+v);
        map.computeIfAbsent(hash, k->new ArrayList<>()).add(i);
    }
    return null;
}

public String longestDupSubstring(String s) {
    int n = s.length();
    int l = 0, r = n;
    String res = "";
    while (l <= r) {
        int m = l + (r-l) / 2;
        String tmp = search(s, m);
        if (tmp == null) r = m-1;
        else {
            if (tmp.length() > res.length())
                res = tmp;
            l = m+1;
        }
    }
    return res;
}

```

#### 4.2.7 1156. Swap For Longest Repeated Character Substring - Medium

You are given a string `text`. You can swap two of the characters in the text.

Return the length of the longest substring with repeated characters.

给你一个字符串，如何找最长的重复子串，博主会数连续相同的字符，若此时有一个不同字符出现了，只要后面还有相同的字符，就会继续数下去，因为有一次交换的机会，什么时候停止呢，当再次出现不同字符的时候就停止，或者是当前统计个数等于该字符出现的总个数时也停止，因为得到的结果不可能超过某个字符出现的总个数。所以可以先统计每个字符的出现次数，然后开始遍历字符，对于每个遍历到的字符，都开始数之后跟其相等的字符，新建变量 `j`, `cnt`, 和 `diff`, 当 `j` 小于 `n`, 且当前字符和比较字符相同，或者 `diff` 等于 0，且 `cnt` 小于比较字符出现的总个数时进行遍历，若当前遍历到的字符和要比较的字符不相等，说明该使用交换操作了，`diff` 自增 1，此时将 `i` 更新为 `j-1`，这是一个优化操作，可以避免一些不必要的计算，下次从这个位置往后统计，也相当于重置了 `diff`。还有就是这个 `cnt` 小于字符出现总个数这个条件卡的非常好，即便下一个还是相同字符，也不能再统计了，因为最后的这个相同字符可能是要用来自换前面的断点位置的。每次用统计出来的 `cnt` 更新结果 `res`，但是一个方向的遍历可能无法应对所有情况，比如“`acbaaa`”，若只是从前往后遍历，那么最终只能得到 3，而正确的答案是 4，因为可以将 `b` 和第一个 `a` 交换，所以还需要从后往前进行一次相同的操作，这样才能得到正确的答案，参见代码如下：

```

public int maxRepOpt1(String s) { // O(n^2)
    int n = s.length(), ans = 0;
    Map<Character, Integer> charCnt = new HashMap<>();
    for (char c : s.toCharArray())
        charCnt.put(c, charCnt.getOrDefault(c, 0) + 1);
    for (int i = 0; i < n; i++) {
        char cur = s.charAt(i);
        int j = i, cnt = 0, dif = 0;
        while (j < n && (cur == s.charAt(j) || dif == 0) && cnt < charCnt.get(cur)) {
            if (cur != s.charAt(j)) {
                ++dif;
                i = j-1; // exchanged once, i moves to be the repeated sequence tail
            }
            ++cnt;
            ++j;
        }
        ans = Math.max(ans, cnt);
    }
    for (int i = n-1; i >= 0; i--) {
        char cur = s.charAt(i);
        int j = i, cnt = 0, dif = 0;
        while (j >= 0 && (cur == s.charAt(j) || dif == 0) && cnt < charCnt.get(cur)) {
            if (cur != s.charAt(j)) {
                ++dif;
                i = j+1;
            }
            ++cnt;
            --j;
        }
        ans = Math.max(ans, cnt);
    }
}

```

```

        }
        ++cnt;
        --j;
    }
    ans = Math.max(ans, cnt);
}
return res;
}

```

- O(N) 解法

上面的解法严格来说还是平方级的，再来看一种线性时间的解法，可能比较难想，由于这里需要关注的是相同字符的出现位置，所以可以将所有相同的字符的位置都放到一个数组中，那么这里就建立一个字符和其出现位置数组之间的映射。由于题目中限制了只有英文字母，所以可以按照每个字母进行遍历，直接遍历每个字符的位置数组，这里新建变量 `cnt`, `cnt2`, 和 `mx`, 其中 `cnt` 统计的是连续字母的个数，`cnt2` 相当于一个临时变量，当使用交换操作时，保存之前的 `cnt` 值，`mx` 为二者之和。在遍历完某个字母位置数组之后，最后看一下若该字母出现总个数大于 `mx`，则说明交换后的字母还没有统计进去，不管之前有没有使用交换操作，都需要加上这个额外的一个，参见代码如下：

```

public int maxRepOpt1(String s) { // O(n^2)
    int n = s.length(), ans = 0;
    Map<Character, List<Integer>> idxMap = new HashMap<>();
    for (int i = 0; i < n; i++)
        idxMap.computeIfAbsent(s.charAt(i), k -> new ArrayList<>()).add(i);
    for (char c = 'a'; c <= 'z'; c++) {
        if (!idxMap.containsKey(c)) continue;
        int cnt = 1, cntb = 0, max = 0;
        List<Integer> idxs = idxMap.get(c);
        for (int i = 1; i < idxs.size(); i++) {
            if (idxs.get(i) == idxs.get(i-1) + 1) // aa
                ++cnt;
            else {
                cntb = (idxs.get(i) == idxs.get(i-1) + 2) ? cnt : 0; // aba ?
                cnt = 1;
            }
            max = Math.max(max, cnt + cntb);
        }
        ans = Math.max(ans, max + (idxs.size() > max ? 1 : 0)); // aaaaabaaaaaca 多于两个重复子段，中间替换字符可是是相同的
    }
    return ans;
}

```

## 算法

(动态规划)  $O(n)$

1. 我们分别考察每种字符所能得到的最大答案。
2. 假设我们当前考察的是字符 `c`。我们通过遍历一次数组的方式，找到单字符就是 `c` 的情况下的最长单重复子串。
3. 设 `cnt` 为字符串中 `c` 出现的次数；`f(i)` 表示以  $i$  结尾且没有替换过不是 `c` 的字符时的最长但重复子串；`g(i)` 表示以  $i$  结尾且替换过不是 `c` 的字符时的最长但重复子串。
4. 当 `c == text[i]` 时，累加 `cnt`， $f(i) = f(i-1) + 1$ ,  $g(i) = g(i-1) + 1$ ，也就是都可以沿着上一次的结果往后累加 1 的长度。
5. 如果 `c != text[i]`，则  $g(i) = f(i-1) + 1$ ，表示我们强制替换 `text[i]` 的字符为 `c` (注意不是交换)，所以要从没有替换过的 `f` 数组转移； $f(i) = 0$ ，表示如果不替换，那么长度只能归 0。交换只是替换的一种特殊形式，我们都先假设总能交换。如果不能交换，如 `aaabaaa` 我们无法把中间的 `b` 用一个新的 `a` 交换时，可以通过答案不能超过 `cnt` 来特判。
6. 答案为  $\min(cnt, \max(f(i), g(i)))$ ，也就是所有的 `f` 和 `g` 的最大值，但结果不能超过 `cnt`。
7. 这里的 `f` 和 `g` 数组因为之和上一位有关，所以可以化简为变量。

## 时间复杂度

- 枚举每种字符时，只需要遍历数组一次，故时间复杂度为  $O(n)$ 。

## 空间复杂度

- 动态规划    • 优化后，只需要常数个变量，故空间复杂度为  $O(1)$ 。

```

private int solve(char c, String s) {
    int n = s.length(), max = 0;
    int f = 0, g = 0, cnt = 0;
    for (int i = 0; i < n; i++) {
        if (c == s.charAt(i)) {

```

```

        f++;
        g++;
        cnt++;
    } else {
        g = f + 1;
        f = 0;
    }
    max = Math.max(max, Math.max(f, g));
}
return Math.min(max, cnt);
}
public int maxRepOpt1(String s) {
    int n = s.length(), ans = 0;
    for (char i = 'a'; i <= 'z'; i++)
        ans = Math.max(ans, solve(i, s));
    return ans;
}
}

```

#### 4.2.8 395. Longest Substring with At Least K Repeating Characters - Medium

Given a string  $s$  and an integer  $k$ , return the length of the longest substring of  $s$  such that the frequency of each character in this substring is greater than or equal to  $k$ .

```

// 由于字母只有 26 个，而整型 mask 有 32 位，足够用了,
// 每一位代表一个字母，如果为 1，表示该字母不够 k 次，如果为 0 就表示已经出现了 k 次，这种思路真是太聪明了,
// 隐约记得这种用法在之前的题目中也用过，但是博主并不能举一反三（沮丧脸:(），还得继续努力啊。
// 遍历字符串，对于每一个字符，都将其视为起点，然后遍历到末尾，增加 HashMap 中字母的出现次数，如果其小于 k，将 mask 的对应位改为 1，如果大于等于 k，将 mask 对应位改为 0。然后看 mask 是否为 0，是的话就更新 res 结果，然后把当前满足要求的子字符串的起始位置 j 保存到 max_idx 中，等内层循环结束后，将外层循环变量 i 赋值为 max_idx+1，继续遍历。
public int longestSubstring(String s, int k) { // O(N^2)
    int n = s.length(), res = 0, i = 0;
    while (i + k <= n) {
        int[] m = new int[26];
        int mask = 0, maxIdx = i;
        for (int j = i; j < n; j++) {
            int t = s.charAt(j) - 'a';
            m[t]++;
            if (m[t] < k) mask |= (1 << t);
            else mask &= (~(1 << t));
            if (mask == 0) {
                res = Math.max(res, j - i + 1);
                maxIdx = j;
            }
        }
        i = maxIdx + 1;
    }
    return res;
}

```

- 双指针 sliding window O(N)

```

public int longestSubstring(String s, int k) {
    int n = s.length(), res = 0;
    for (int cnt = 1; cnt <= 26; cnt++) {
        int start = 0, i = 0, uniqueCnt = 0;
        int[] charCntr = new int[26];
        while (i < n) {
            boolean valid = true;
            if (charCntr[s.charAt(i++) - 'a']++ == 0) ++ uniqueCnt;
            while (uniqueCnt > cnt) {
                if (--charCntr[s.charAt(start++) - 'a'] == 0) --uniqueCnt;
                for (int j = 0; j < 26; j++)
                    if (charCntr[j] > 0 && charCntr[j] < k) valid = false;
                if (valid) res = Math.max(res, i - start);
            }
        }
    }
    return res;
}

```

- 分治：分而治之

```

public int longestSubstring(String s, int k) { // str.split("[dkfldjf]")
    int n = s.length();
    if (n < k) return 0;
    if (n == k && s.chars().distinct().count() == 1) return k;
    int[] cnt = new int[26];
    for (int i = 0; i < n; i++)
        cnt[s.charAt(i) - 'a']++;
    if (Arrays.stream(cnt).max().getAsInt() < k) return 0;
    StringBuilder sb = new StringBuilder("[");
    for (int i = 0; i < 26; i++)
        if (cnt[i] < k && cnt[i] != 0)
            sb.append((char)(i + 'a'));
    sb.append(']');
    if (sb.length() == 2) return n;
    String[] sa = s.split(sb.toString()); // str.split("[-*/=]") pay attention to the format
}

```

```

System.out.println(Arrays.toString(sa));
int max = 0;
for (int i = 0; i < sa.length; i++)
    max = Math.max(max, longestSubstring(sa[i], k));
return max;
}
public int longestSubstring(String s, int k) { // 人工手动拆分
    int n = s.length(), maxIdx = 0, res = 0;
    int[] cnt = new int[128];
    boolean valid = true;
    for (char c : s.toCharArray())
        cnt[c]++;
    for (int i = 0; i < n; i++) {
        if (cnt[s.charAt(i)] < k) {
            res = Math.max(res, longestSubstring(s.substring(maxIdx, i), k));
            valid = false;
            maxIdx = i+1;
        }
    }
    return valid ? n : Math.max(res, longestSubstring(s.substring(maxIdx, n), k));
}

```

## 4.2.9 1830. Minimum Number of Operations to Make String Sorted - Hard 排列组合费马小定理

You are given a string  $s$  (0-indexed). You are asked to perform the following operation on  $s$  until you get a sorted string:

Find the largest index  $i$  such that  $1 \leq i < s.length$  and  $s[i] < s[i - 1]$ . Find the largest index  $j$  such that  $i \leq j < s.length$  and  $s[k] < s[i - 1]$  for all the possible values of  $k$  in the range  $[i, j]$  inclusive. Swap the two characters at indices  $i - 1$  and  $j$ . Reverse the suffix starting at index  $i$ . Return the number of operations needed to make the string sorted. Since the answer can be too large, return it modulo  $10^9 + 7$ .

- 解题思路与分析 题中每次对字符串  $s$  执行的操作，是将其变为由当前字母组成的前一字典序的字符串。因此求最少操作次数，等价于求解该字符串在由当前字母组成的所有排列中的字典序；

求比当前字符串  $s$  小的排列个数，可通过排列组合公式计算得到；

排列组合公式中的阶乘逆元取模，可通过费马小定理，转化为对模数的乘方进行计算；

可通过快速乘方算法，进一步提高对乘方的计算效率。

```

private int quickmul(int base, int exp) { // 快速乘方算法
    long ans = 1L;
    while (exp > 0) {
        if ((exp & 1) == 1) // 指数是奇数，就先乘一次 base
            ans = ans * base % mod;
        base = (int)((long)base * base % mod); // 底平方（指数变偶数之后）
        exp >>= 1; // 指数除 2，快速计算
    }
    return (int)ans;
}
int mod = (int)1e9 + 7;
public int makeStringSorted(String t) {
    int n = t.length();
    char[] s = t.toCharArray();
    int[] cnt = new int[26]; // 记录剩余字符串中各字母个数
    Arrays.fill(cnt, 0);
    for (int i = 0; i < n; i++)
        cnt[s[i] - 'a']++;
    int[] fact = new int[n+1];
    int[] finv = new int[n+1];
    fact[0] = 1;
    finv[0] = 1;
    for (int i = 1; i <= n; i++) {
        fact[i] = (int)((long)fact[i-1] * i % mod); // fact[i] = i! % mod
        finv[i] = quickmul(fact[i], mod - 2); // 费马小定理计算乘法逆元，finv[i] = (i!) ^ -1 % mod
    }
    long ans = 0L;
    for (int i = 0; i < n-1; i++) {
        int lessCnt = 0; // 比当前位置小的字母总数
        for (int j = 0; j < s[i] - 'a'; j++)
            lessCnt += cnt[j];
        long upper = (long)lessCnt * fact[n-1-i] % mod; // 排列公式分子
        for (int j = 0; j < 26; j++)
            upper = upper * finv[cnt[j]] % mod;
        ans = (ans + upper) % mod;
        cnt[s[i] - 'a']--; // 指针右移
    }
    return (int)ans;
}

```

### 4.2.10 1960. Maximum Product of the Length of Two Palindromic Substrings - Hard 马拉车算法 todo

You are given a 0-indexed string  $s$  and are tasked with finding two non-intersecting palindromic substrings of odd length such that the product of their lengths is maximized.

More formally, you want to choose four integers  $i, j, k, l$  such that  $0 \leq i \leq j < k \leq l < s.length$  and both the substrings  $s[i\dots j]$  and  $s[k\dots l]$  are palindromes and have odd lengths.  $s[i\dots j]$  denotes a substring from index  $i$  to index  $j$  inclusive.

Return the maximum possible product of the lengths of the two non-intersecting palindromic substrings.

A palindrome is a string that is the same forward and backward. A substring is a contiguous sequence of characters in a string.

#### 1. 解题思路与分析

这个马拉车，好像还是很半生不熟，要好好理解消化一下

[https://leetcode.com/problems/maximum-product-of-the-length-of-two-palindromic-substrings/discuss/1393288/Java-100-O\(n\)-time-O\(n\)-space-using-Manacher's-algorithm](https://leetcode.com/problems/maximum-product-of-the-length-of-two-palindromic-substrings/discuss/1393288/Java-100-O(n)-time-O(n)-space-using-Manacher's-algorithm)

```
private static int[] manacherOdd(String str) {
    int n = str.length();
    char[] s = str.toCharArray();
    int[] ans = new int[n];
    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int len = i > r ? 1 : Math.min(ans[l+r-i], r-i+1);
        int maxLen = Math.min(i, n-1-i);
        int x = i - len, y = i + len;
        while (len <= maxLen && s[x--] == s[y++]) len++;
        ans[i] = len--;
        if (i + len > r) {
            l = i - len;
            r = i + len;
        }
    }
    return ans;
}
public long maxProduct(String s) { // 这个马拉车，得多写几遍
    int n = s.length();
    int[] d = manacherOdd(s);
    int[] l = new int[n], r = new int[n];
    for (int i = 0; i < n; i++) {
        l[i+d[i]-1] = Math.max(l[i+d[i]-1], 2 * d[i]-1);
        r[i-d[i]+1] = 2 * d[i] - 1;
    }
    for (int i = n-2, j = n-1; i >= 0; i--, j--) {
        l[i] = Math.max(l[i], l[j]-2);
        for (int i = 1, j = 0; i < n; i++, j++)
            r[i] = Math.max(r[i], r[j]-2);
        for (int i = 1, j = 0; i < n; i++, j++)
            l[i] = Math.max(l[i], l[j]);
        for (int i = n-2, j = n-1; i >= 0; i--, j--)
            r[i] = Math.max(r[i], r[j]);
    }
    long ans = 1;
    for (int i = 1; i < n; i++)
        ans = Math.max(ans, (long)l[i-1] * r[i]);
    return ans;
}
```

#### 2. DP 解：要再好好理解消化一下

### 4.2.11 854. K-Similar Strings - Hard

Strings  $s_1$  and  $s_2$  are  $k$ -similar (for some non-negative integer  $k$ ) if we can swap the positions of two letters in  $s_1$  exactly  $k$  times so that the resulting string equals  $s_2$ .

Given two anagrams  $s_1$  and  $s_2$ , return the smallest  $k$  for which  $s_1$  and  $s_2$  are  $k$ -similar.

#### 1. 解题思路与分析

```
public int kSimilarity(String s, String t) {
    if (s.equals(t)) return 0;
    ArrayDeque<String> q = new ArrayDeque<>();
    Set<String> vis = new HashSet<>();
    q.offerLast(s);
    vis.add(s);
    int cnt = 0;
    while (!q.isEmpty()) {
        for (int z = q.size()-1; z >= 0; z--) {
            String cur = q.pollFirst();
            // if (cur.equals(t)) return cnt;
            for (String next : getAllNeighbour(cur, t)) {
                if (!vis.contains(next)) {
                    vis.add(next);
                    q.offerLast(next);
                }
            }
        }
    }
    return cnt;
}
```

```

        if (vis.contains(next)) continue;
        if (next.equals(t)) return cnt + 1;
        q.offerLast(next);
        vis.add(next);
    }
    cnt++;
}
return -1;
}
List<String> getAllNeighbour(String ss, String tt) {
    int n = ss.length(), i = 0;
    char [] s = ss.toCharArray();
    char [] t = tt.toCharArray();
    List<String> ans = new ArrayList<>();
    while (i < n && s[i] == t[i]) i++;
    for (int j = i+1; j < n; j++)
        if (s[j] == t[i]) {
            swap(i, j, s);
            ans.add(new String(s));
            swap(i, j, s);
        }
    return ans;
}
void swap(int i, int j, char [] s) {
    char c = s[i];
    s[i] = s[j];
    s[j] = c;
}
}

```

#### 4.2.12 Queue ArrayDeque LinkedList 效率比较

##### 1. 为什么 ArrayDeque 比 LinkedList 快

了解完数据结构特点之后，接下来我们从两个方面分析为什么 ArrayDeque 作为队列使用时可能比 LinkedList 快。

- 从速度的角度：ArrayDeque 基于数组实现双端队列，而 LinkedList 基于双向链表实现双端队列，数组采用连续的内存地址空间，通过下标索引访问，链表是非连续的内存地址空间，通过指针访问，所以在寻址方面数组的效率高于链表。
- 从内存的角度：虽然 LinkedList 没有扩容的问题，但是插入元素的时候，需要创建一个 Node 对象，换句话说每次都要执行 new 操作，当执行 new 操作的时候，其过程是非常慢的，会经历两个过程：类加载过程、对象创建过程。
  - 类加载过程
    - \* 会先判断这个类是否已经初始化，如果没有初始化，会执行类的加载过程
    - \* 类的加载过程：加载、验证、准备、解析、初始化等等阶段，之后会执行 `<clinit>()` 方法，初始化静态变量，执行静态代码块等等
  - 对象创建过程
    - \* 如果类已经初始化了，直接执行对象的创建过程
    - \* 对象的创建过程：在堆内存中开辟一块空间，给开辟空间分配一个地址，之后执行初始化，会执行 `<init>()` 方法，初始化普通变量，调用普通代码块

集合类型	数据结构	初始化及扩容	插入/删除时间复杂度	查询时间复杂度	是否是线程安全
ArrqyDeque	循环数组	初始化：16 扩容：2 倍	O(n)	O(1)	否
LinkedList	双向链表	无	O(1)	O(n)	否

- 最好的做法是自己用 arrayDeque 封装一个一端进出的 stack

#### 4.3 AC 自动机：

- 亲爱的表哥的活宝妹，最近第一次，接触到这个数据结构的题目。不过，觉得这个【AC 自动机】不难。

##### 4.3.1 3292. Minimum Number of Valid Strings to Form Target II

- You are given an array of strings words and a string target.

A string x is called valid if x is a prefix of any string in words.

Return the minimum number of valid strings that can be concatenated to form target. If it is not possible to form target, return -1.

```

// 【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】
public class Node {
    Node [] n;
    Node f; // 当 cur.son[i] 不能匹配 target 中的某个字符时, cur.fail.son[i] 即为下一个待匹配节点 (等于 root 则表示没有匹配)
    int l; // l: length
    public Node(int l) {
        n = new Node [26];
        this.l = l;
    }
}
public class AhoCorasick {
    Node root = new Node(0);
    public void put(String s) {
        Node r = root;
        for (char j : s.toCharArray()) {
            j -= 'a';
            if (r.n[j] == null)
                r.n[j] = new Node(r.l + 1);
            r = r.n[j];
        }
    }
    public void buildFail() {
        Node r = root;
        r.f = r; // 【根节点】: f 指向自己, 方便转圈【避开、不必要的、非空检测】???
        ArrayDeque<Node> q = new ArrayDeque<>();
        // 处理: 【根节点】所有可能的子节点
        for (int i = 0; i < 26; i++) {
            Node s = r.n[i];
            if (s == null) // 它说, 这里是, 【虚拟指针】。什么意思..
                // 不存在, 就指向父节点的, 这个指针
                // r.n[i] = r.f.n[i]; // 【写错了】: 根节点, 不存在还是不存在; 就是后来的子节点, 能转什么吗 ???
                r.n[i] = r; // 根节点, 不存在的子节点, 就指向自己
            else {
                // s.f = r.f.n[i]; // 【写错了】: 根节点 f 指向自己
                s.f = r;
                q.offerFirst(s);
            }
        }
        // 处理后代: 跟【根节点】写法一模一样, 为什么要写两遍?
        while (!q.isEmpty()) {
            r = q.pollLast(); // 全部、非空
            for (int i = 0; i < 26; i++) {
                Node s = r.n[i];
                if (s == null) // 它说, 这里是, 【虚拟指针】。什么意思..
                    // 虚拟子节点 cur.son[i], 和 cur.fail.son[i] 是同一个
                    // 方便失配时直接跳到下一个可能匹配的位置 (但不一定是某个 words[k] 的最后一个字母)
                    r.n[i] = r.f.n[i]; // 当 r.f 非空, 就指向了 r.f.n[i] 节点
                else {
                    // 下面: 一次【赋值跳转】, 但在【字典树】中, 因为【自顶向下】的 f 指针构建
                    s.f = r.f.n[i];
                    q.offerFirst(s);
                }
            }
        }
    }
    public int minValidStrings(String [] sa, String S) {
        int n = S.length(); char [] s = S.toCharArray();
        AhoCorasick ac = new AhoCorasick();
        for (String si : sa) ac.put(si);
        ac.buildFail();
        int [] f = new int [n+1];
        Node r = ac.root;
        for (int i = 0; i < n; i++) {
            // 下面: 如果没有匹配, 相当于移动到, 父节点 r 的 fail 节点的 son[s[i]-'a']
            r = r.n[s[i] - 'a']; // 隐藏在幕后的、无数链接跳转...
            // 没有任何字符串的前缀与 target[..i] 的后缀匹配
            if (r == ac.root)
                return -1;
            f[i+1] = f[i+1 - r.l] + 1;
        }
        return f[n];
    }
}

```



# Chapter 5

## Bit Manipulations

### 5.1 基本概念：原码、反码、与补码（对负数尤其重要）

#### 5.1.1 1、原码：

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

比如 00000000 00000000 00000000 00000101 是 5 的原码。

10000000 00000000 00000000 00000101 是 -5 的原码。

备注：

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

#### 5.1.2 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。

#### 5.1.3 3、补码

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

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

那么，补码为：

11111111 11111111 11111111 11111010 + 1 = 11111111 11111111 11111111 11111011

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

2、补码却规定 0 没有正负之分

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

## 5.2 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  $x \hat{=} num$  的异或操作便消除）。

isolate rightmost 1-bit  
which is different for  
x and y

bitmask

	0	0	0	0	0	0	0	0
x = 1	0	0	0	0	0	0	0	1
y = 2	0	0	0	0	0	0	1	0
a = 3	0	0	0	0	0	0	1	1
a = 3	0	0	0	0	0	0	1	1

bitmask =  
bitmask ^ x ^ y ^ a ^ a

diff =  
bitmask & (-bitmask)

	0	0	0	0	0	0	1	1
	0	0	0	0	0	0	0	1

### 5.3 371. 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。然后运用 << 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 1011

为何-5 是这样：首先把上面的 bits -1

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：

```
11111111111111111111111111111111011
1111
111111111111111111111111111111110100
10110
.....
1111000000000000000000000000000001010
10000000000000000000000000000000000000000
111000000000000000000000000000000000000001010
10000000000000000000000000000000000000000
110000000000000000000000000000000000000001010
10000000000000000000000000000000000000000
100000000000000000000000000000000000000001010
10000000000000000000000000000000000000000
100000000000000000000000000000000000000001010
1010
0
10
```

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

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

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

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

### 5.3.1 201. Bitwise AND of Numbers Range

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

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

### 5.3.2 1835. Find XOR Sum of All Pairs Bitwise AND - Hard

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

For example, the XOR sum of [1,2,3,4] is equal to 1 XOR 2 XOR 3 XOR 4 = 4, and the XOR sum of 4 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  $\text{arr1}[i]$  AND  $\text{arr2}[j]$  (bitwise AND) for every  $(i, j)$  pair where  $0 \leq i < \text{arr1.length}$  and  $0 \leq j < \text{arr2.length}$ .

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;
}
```

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

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

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

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

```
// '平生不识 TwoSum, 刷尽 LeetCode 也枉然' 还好不至于哭死呀.....
public int countTriplets(int[] arr) {
    Map<Integer, Integer> m = new HashMap<>();
    int v = 0, res = 0;
    for (int i = 0; i < arr.length; i++) {
        for (int j = 0; j < arr.length; j++) {
            v = arr[i] & arr[j];
            m.put(v, m.getOrDefault(v, 0) + 1);
        }
    }
    for (int i = 0; i < arr.length; i++)
        for (int k : m.keySet())
            if ((arr[i] & k) == 0) res += m.get(k);
    return res;
}
public int countTriplets(int[] arr) { // 这种方法执行起来效率更高一点儿
    int res = 0, v = 0;
    int[] cnt = new int[1 << 16];
    Arrays.fill(cnt, -1);
    for (int a : arr)
        for (int b : arr) {
            v = a & b;
            if (cnt[v] == -1) {
                cnt[v] = 0;
                for (int c : arr)
                    if ((v & c) == 0) ++cnt[v];
            }
            res += cnt[v];
        }
    return res;
}
```

### 5.3.4 187. Repeated DNA Sequences - Medium

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

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

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

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

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

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

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

```

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

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

```

### 5.3.5 1915. Number of Wonderful Substrings - Medium

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

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

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

```

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

```

### 5.3.6 782. Transform to Chessboard- Hard

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

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

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

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

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

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

```

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

```

```

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

```

- 方法一：分维度计算【通过】

### 思路

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

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

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

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

### 算法

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

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

```

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

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

```

```

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

```

### 5.3.7 1803. Count Pairs With XOR in a Range - Hard

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

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

- 1. 暴力算法

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

```

public int countPairs(int[] arr, int low, int high) {
    int[] freq = new int[20002]; // nums[i]<=20000
    for (int v : arr)
        freq[v]++;
    int val = 0, res = 0;
    for (int v : arr) {
        for (int i = low; i <= high; i++) {
            val = v ^ i; // num ^ i = num2 <=> num ^ num2 = i
            if (val <= 20000)
                res += freq[val]; // v^val=i 中当前 v 对应 val 出现的次数
        }
        freq[v]--;
    }
    return res;
}

```

- 2. 字典树 (Trie 树)

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

```

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

```

```

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

```

### 5.3.8 1734. Decode XORed Permutation - Medium

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

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

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

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

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

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

```

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

```

### 5.3.9 957. Prison Cells After N Days - Medium

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

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

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

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

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

```

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

```

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

```

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

```

```

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

```

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

```

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

```

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

```

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

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

```

- 还有一个大神级的思路

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

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

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

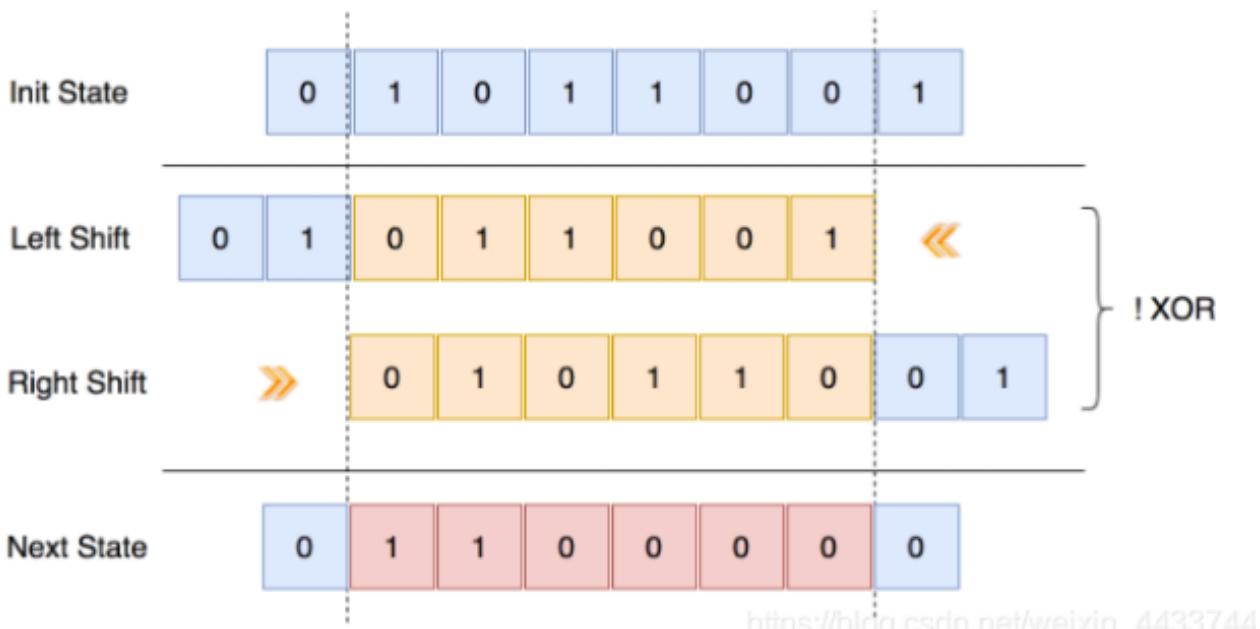
This is the method called fast-forward.

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

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

Solution2:

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



[https://hcdr.csdn.net/walzlin\\_4433744](https://hcdr.csdn.net/walzlin_4433744)

```

public int[] prisonAfterNDays(int[] cells, int N) {
    HashMap<Integer, Integer> seen = new HashMap<>();
    boolean isFastForwarded = false;
    // step 1). convert the cells to bitmap
    int stateBitmap = 0x0;
    for (int cell : cells) {
        stateBitmap <= 1;
        stateBitmap = (stateBitmap | cell);
    }
    // step 2). run the simulation with hashmap
    while (N > 0) {
        if (!isFastForwarded) {
            if (seen.containsKey(stateBitmap)) {
                // the length of the cycle is seen[state_key] - N
                N %= seen.get(stateBitmap) - N;
                isFastForwarded = true;
            } else
                seen.put(stateBitmap, N);
        }
        // check if there is still some steps remained,
        // with or without the fast forwarding.
        if (N > 0) {
            N -= 1;
            stateBitmap = this.nextDay(stateBitmap);
        }
    }
    // step 3). convert the bitmap back to the state cells
    int ret[] = new int[cells.length];
    for (int i = cells.length - 1; i >= 0; i--) {
        ret[i] = (stateBitmap & 0x1);
        stateBitmap = stateBitmap >> 1;
    }
    return ret;
}
protected int nextDay(int stateBitmap) {
    stateBitmap = ~(stateBitmap << 1) ^ (stateBitmap >> 1);
    // set the head and tail to zero
    stateBitmap = stateBitmap & 0x7e;
    return stateBitmap;
}

```

# Chapter 6

## Bit Masks

### 6.0.1 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:  $(S \& (1 \ll i)) == 0$ , 意思是将 1 左移 i 位与 S 进行与运算后，看结果是否为零。
  - 将第 i 位设置为 1:  $S | (1 \ll i)$ , 意思是将 1 左移 i 位与 S 进行或运算。
  - 将第 i 位设置为 0:  $S \& \sim(1 \ll i)$ , 意思是将 S 与第 i 位为 0, 其余位为 1 的数进行与运算;
  - 取第 i 位的值:  $S \& (1 \ll i)$

### 6.0.2 1655. Distribute Repeating Integers - Hard

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

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

#### 1. 解题思路与分析: dfs 回溯

```
private boolean backTracking(int[] arr, int[] quantity, int idx) {
    if (idx < 0) return true;
    Set<Integer> vis = new HashSet<>();
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] < quantity[idx] || vis.contains(arr[i])) continue; // 去杂去重
        vis.add(arr[i]);
        arr[i] -= quantity[idx];
        if (backTracking(arr, quantity, idx-1)) return true;
        arr[i] += quantity[idx];
    }
    return false;
}
public boolean canDistribute(int[] nums, int[] quantity) {
    Map<Integer, Integer> map = new HashMap<>();
    for (Integer v : nums)
        map.put(v, map.getOrDefault(v, 0) + 1);
    int[] arr = new int[map.size()];
    int i = 0;
    for (Integer val : map.values()) arr[i++] = val;
    Arrays.sort(quantity); // decreasing frequency: 是一种裁枝优化
    return backTracking(arr, quantity, quantity.length-1);
}
```

- 不用 map 的操作相对快一点儿

```

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

```

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

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

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

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

因此, 当且仅当能找到这样的子集 s 时,  $dp[i][j]=true$ 。

```

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

```

### 6.0.3 1659. Maximize Grid Happiness - Hard

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

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

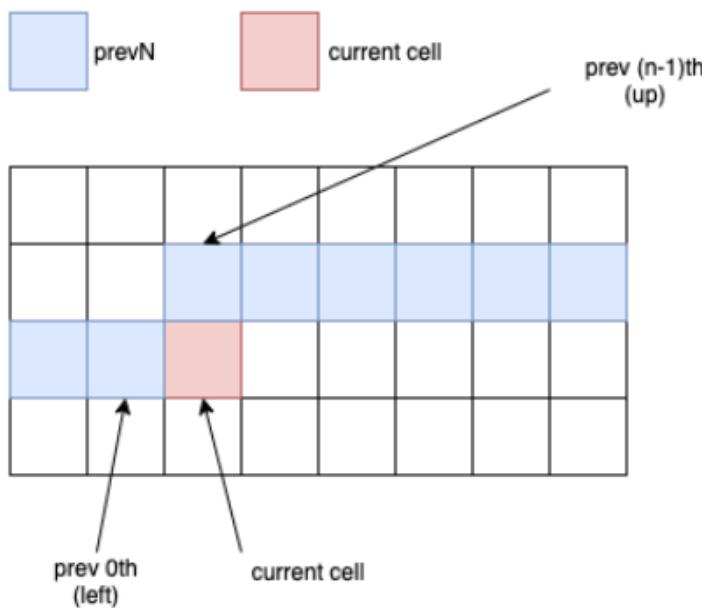
The happiness of each person is calculated as follows:

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

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

### Key Notes:

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



```

public int getMaxGridHappiness(int m, int n, int introvertsCount, int extrovertsCount) {
    this.m = m; this.n = n;
    dp = new Integer [m][n][introvertsCount+1][extrovertsCount+1][243]; // 3^5=243
    return dfs(0, 0, introvertsCount, extrovertsCount, 0);
}
Integer[][][][][] dp;
int m, n, mod = 243;
private int dfs(int i, int j, int inCnt, int exCnt, int preMask) {
    if (inCnt == 0 && exCnt == 0) return 0;
    if (j == n) return dfs(i+1, 0, inCnt, exCnt, preMask); // 有到达边界，直接看下一行
    if (i == m) return 0;
    if (dp[i][j][inCnt][exCnt][preMask] != null) return dp[i][j][inCnt][exCnt][preMask];
    int up = getBit(preMask, n-1), left = getBit(preMask, 0);
    int ans = dfs(i, j+1, inCnt, exCnt, setBit(preMask, 0)); // 不安排任何人在这个格
    if (inCnt > 0) { // 考虑安排一个内向的人在这个格子，算最优可能性
        int addon = 120, cur = setBit(preMask, 1);
        if (i >= 1 && up != 0) { // up: 1 / 2
            addon -= 30; // 当前格内向的人，上面有人他不开心
            if (up == 1) addon -= 30; // 当前格上一格，如果是内向的人，他不开心
            else addon += 20; // 当前格上一格，如果是外向的人，他很开心
        }
        if (j >= 1 && left != 0) {
            addon -= 30;
            if (left == 1) addon -= 30;
            else addon += 20;
        }
    }
}
```

```

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

```

- 简洁版的代码

```

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

```

### 1. 解题思路与分析: 有一种状压, 叫做滑动窗口状压 (112ms)

```

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

```

```

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

```

## 2. 解题思路与分析: 其它

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

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

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

```

xxxx
xxxC

```

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

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

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

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

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

```

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

```

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

## 6.0.4 1755. Closest Subsequence Sum - Hard 分成两半: 由 $O(2^N)$ 降为 $O(N \log N)$

You are given an integer array  $\text{nums}$  and an integer  $\text{goal}$ .

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

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

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

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

- 另一种我都怀疑是不是自己写出来的, 居然会忘了。。。。。

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

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

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

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

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

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

Return the minimum possible absolute difference.

```

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

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

```

- 看一下另一个写法

```

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

```

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

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

- 解题思路与分析: dfs 记忆化搜索? 没想到吧。。。。。。

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

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

算法

$\text{dp}[i][s]$  表示当我们使用  $\text{rods}[j] (j \geq i)$  时能得到的最大 score, 由于之前写下的数字和为  $s$  (不统计在 score 内)。例如,  $\text{rods} = [1, 2, 3, 6]$ , 可以有  $\text{dp}^2, 2 = 5$ , 在写下 1 之后, 可以写下  $+2 + 3 - 6$  使得剩下的  $\text{rods}[i:]$  获得 score 为 5。

边界情况:  $\text{dp}[\text{rods.length}][s]$  是 0 当  $s == 0$ , 剩余情况为  $-\infty$ 。递推式为  $\text{dp}[i][s] = \max(\text{dp}[i+1][s], \text{dp}[i+1][s - \text{rods}[i]], \text{rods}[i] + \text{dp}[i+1][s + \text{rods}[i]])$ 。

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

## 2. 解题思路与分析: 折腰轨半搜索 (和上面两个题比一比)

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

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

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

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

算法

将钢筋分成左右两半: 左侧和右侧。

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

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

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

- 之前看别人的，还需要理解消化，官方 dp 解也需要消化一下

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

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

```

```

// 定义一个数对键值: (i, j): i 表示两个子序列的累加和差值的绝对值, j 表示这个差值下, 子序列中累加和的最大值, 定义一个 dp 的 map 存放前 m 个数的所有子序列的累加和差值的绝对值
// 新建一个 HashMap temp 用于存放第 m 个数对之前子序列累加和只差的状态转移结果
// 对于新到来的 rod, 只能够有 3 中情况:
// 1. rod 不加入任何列表:
//    从 dp 中拿出每个子序列的差值 k 的累加和最大值 v1, 每个结果与 temp 中相应 k 的累加和最大值 v2 比较, 如果 v1>v2, 那么更新 temp 中 k 对应的累加和最大值为 v1
// 2. rod 加入累加和较大的序列:
//    从 dp 中拿出每个子序列的差值 k 的累加和最大值 v1, 并加上 rod, 这时差值变成 k + rod, 累加和最大值变成 v1+ rod, 每个结果与 temp 中相应 k + rod 对应的累加和最大值为 v1+ rod
// 3. rod 加入累加和较小的序列:
//    从 dp 中拿出每个子序列的差值 k 的累加和最大值 v1, 累加和较小的子序列加上了 rod, 那么和累加和较大的子序列之差为 k - rod, k - rod 为负数时, 说明累加和较小的子序列加上了 rod, 那么和累加和较大的子序列之差为 k + rod, k + rod 为负数时, 说明累加和较大的子序列加上了 rod, 那么和累加和较小的子序列之差为 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);
}

```

### 6.0.7 1982. Find Array Given Subset Sums - Hard

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

Return the array  $\text{ans}$  of length  $n$  representing the unknown array. If multiple answers exist, return any of them.

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

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

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

## 所有元素均为非负数

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

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

复杂度  $\mathcal{O}(2^n n)$ 。

## 所有元素可以为任意整数

### 解法

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

复杂度仍为 复杂度  $\mathcal{O}(2^n n)$ 。

### 证明

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

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

### 1. 解题思路与分析 求子集和可以用二进制状压 DP

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

最后找与偏移量相等的子集和的时候还能复用到

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

```

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

```

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

### 6.0.8 1815. Maximum Number of Groups Getting Fresh Donuts - Hard

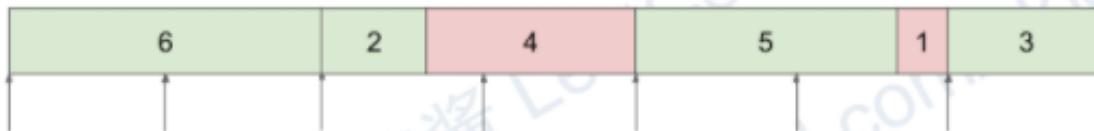
There is a donut shop that bakes donuts in batches of batchSize. They have a rule where they must serve all of the donuts of a batch before serving any donuts of the next batch. You are given an integer batchSize and an integer array groups, where groups[i] denotes that there is a group of groups[i]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

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

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

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

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



Key observation:

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



1. 解题思路与分析：// 回溯 DFS + 记忆化搜索

```

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

```

```

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

```

- 另一种 DP 超时的做法，掌握这个方法

```

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

```

## 6.0.9 691. Stickers to Spell Word - Hard

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

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

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

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

- 【位图法】因为待匹配串 target 的数量最多是 15 个，因此其子集的数量最多有  $2^{15}$ 
  - 一个，而 int 类型占用四个字节，能够容纳标识所有 target 的子集。所以我们可以将 target 的子集映射到 int 的整型数中。
- 【int 与 target 子集之间的映射关系】将 int 类型分解为二进制的形式后，有些位置为 0，有些位置为 1. 表明在 target 中哪些位置的字符是否保留（1 表示保留）。
- 【动态规划】dp 中存储的是得到子集  $i$ , 需要的最少的单词的数量。

```

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

```

- 另一种解法

```

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

```

```

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

```

- 上面的这个，因为使用了图，以及必要的优化，性能还比较好

什么叫状态压缩？其实就是用二进制数来表示动态规划状态转移过程中的状态。

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

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

后得到的状态对应的二进制数一定大于它的父状态。所以我们可以很自然地从 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-su-dpji-you-hua-by-lucifer/>

```

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];
}

```

### 6.0.10 1994. The Number of Good Subsets

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

For example, if `nums` = [1, 2, 3, 4]: [2, 3], [1, 2, 3], and [1, 3] are good subsets with products  $6 = 2 \cdot 3$ ,  $6 = 2 \cdot 3$ , and  $3 = 3$  respectively. [1, 4] and <sup>1</sup> are not good subsets with products  $4 = 2 \cdot 2$  and  $4 = 2 \cdot 2$  respectively. Return the number of different good subsets in `nums` modulo 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.

### 6.0.11 1494. Parallel Courses II - Hard

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

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

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

```

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

```

<sup>1</sup>DEFINITION NOT FOUND.

```

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

```

- 另一种慢很多的：通过递归来优化选课时间的

## 算法

(状态压缩动态规划)  $O(n \cdot 2^n + 3^n)$

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

$$f(S_0 \text{ bit or } S) = \min(f(S_0 \text{ bit or } S), f(S_0) + 1).$$
5. 最终答案为  $f((1 << n) - 1)$ 。

## 时间复杂度

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

## 空间复杂度

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

```

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

```

## 6.0.12 分析一下

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

```

static int mod = (int) 1e9 + 7;
static int[] map = new int[31];
static {
    int[] prime = new int[] {2, 3, 5, 7, 11, 13, 17, 19, 23, 29}; // length: 10
    for (int i = 2; i <= 30; ++i) {
        if (i % 4 == 0 || i % 9 == 0 || i == 25) continue;
        int mask = 0;
        for (int j = 0; j < 10; ++j)
            if (i % prime[j] == 0)
                mask |= 1 << j;
        map[i] = mask;
    }
}
public int numberOfGoodSubsets(int[] nums) {
    int n = nums.length, one = 0;
    int[] dp = new int[1024], cnt = new int[31]; // 1024 ?
    dp[0] = 1;
    for (int i : nums) {
        if (i == 1) one++;
        else if (map[i] != 0) cnt[i]++;
    }
    for (int i = 0; i < 31; ++i) {
        if (cnt[i] == 0) continue;
        for (int j = 0; j < 1024; ++j) {
            if ((j & map[i]) != 0) continue; // 含有某个公共质因子 val 子集的统计数 * 当前 val 的重复次数
            dp[j | map[i]] = (int) ((dp[j | map[i]] + dp[j] * (long) cnt[i]) % mod);
        }
    }
    long res = 0;
    for (int i : dp) res = (res + i) % mod;
    res--; // 应该是减去一个 1 吧
    if (one != 0) res = res * pow(one) % mod;
    return (int) res;
}
private long pow(int n) { // 快速幂
    long res = 1, m = 2;
    while (n != 0) {
        if ((n & 1) == 1) res = (res * m) % mod;
        m = m * m % mod;
        n >>= 1;
    }
    return res;
}

```

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

```

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

```

```

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

```

- // Speed up using frequency array. O( $30 \times 1024 + N$ ) = linear time
- 看不懂：<https://leetcode.com/problems/the-number-of-good-subsets/discuss/1444661/Java-DP-%2B-Bitmask-or>

```

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

```

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

### 6.0.13 1349. Maximum Students Taking Exam - Hard

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

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

Students must be placed in seats in good condition.

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

- 自己写的，bug 终于是被自己找出来了。。。。。

```
public int maxStudents(char[][][] seats) {
    m = seats.length;
    n = seats[0].length;
    int r = 1 << n;
    int [][] dp = new int [m+1][r];
    for (int i = 0; i <= m; i++)
        Arrays.fill(dp[i], -1);
    dp[0][0] = 0;
    for (int i = 1; i <= m; i++)
        for (int k = 0; k < r; k++) {
            if (!isValid(seats, k, i-1)) continue;
            if ((k & (k >> 1)) != 0) continue; // BUG: 要保证每个 set 过的位左右无人，这一行不可少
            for (int j = 0; j < r; j++) {
                if (dp[i-1][j] == -1) continue; // 前一排坐位的遍历，j 可能不是有效子集，比如坐墙上或左右有人
                if (((k >> 1) & j) == 0 && (k & (j >> 1)) == 0)
                    dp[i][k] = Math.max(dp[i][k], dp[i-1][j] + Integer.bitCount(k));
            }
        }
    int max = 0;
    for (int i = 0; i < r; i++)
        max = Math.max(max, dp[m][i]);
    return max;
}
int m, n;
private boolean isValid(char [][] arr, int v, int idx) { // 只是检查不会坐在墙上：方法应该是对的，但可能会比较慢
    for (int i = 0; i < n; i++)
        if (((v >> i) & 1) == 1 && arr[idx][i] != '.') return false;
    return true;
}
```

- 有个姑娘总结的 dfs 记忆化搜索

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

```

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

```

## 6.0.14 1434. Number of Ways to Wear Different Hats to Each Other - Hard

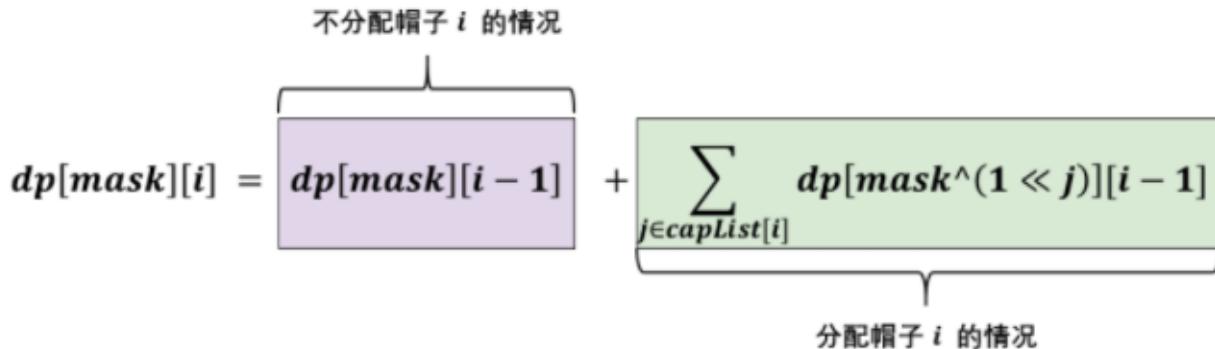
There are n people and 40 types of hats labeled from 1 to 40.

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

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

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

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



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

则  $i = 2$  时的 DP Table 如下所示:

		mask							
		000	001	010	011	100	101	110	111
$i$		0	1	0	0	0	0	0	0
		1	1	1	0	0	0	0	0
2	1	1	1	1	0	0	0	0	0
3	1	1	1	1	0	0	0	0	0
4	1	1	1	1	0	0	0	0	0
5									
...	...	...	...	...	...	...	...	...	...
100									

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

```

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

```

### 6.0.15 1595. Minimum Cost to Connect Two Groups of Points - Hard 自顶向下 (dfs + 记忆数组); 自底向上: DP table

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

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

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

```

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

```

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

```

public int connectTwoGroups(List<List<Integer>> cost) {
    int m = cost.size();
    int n = cost.get(0).size();
    int [][] dp = new int [m][1 << 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)]);
            }
        }
    }
}

```

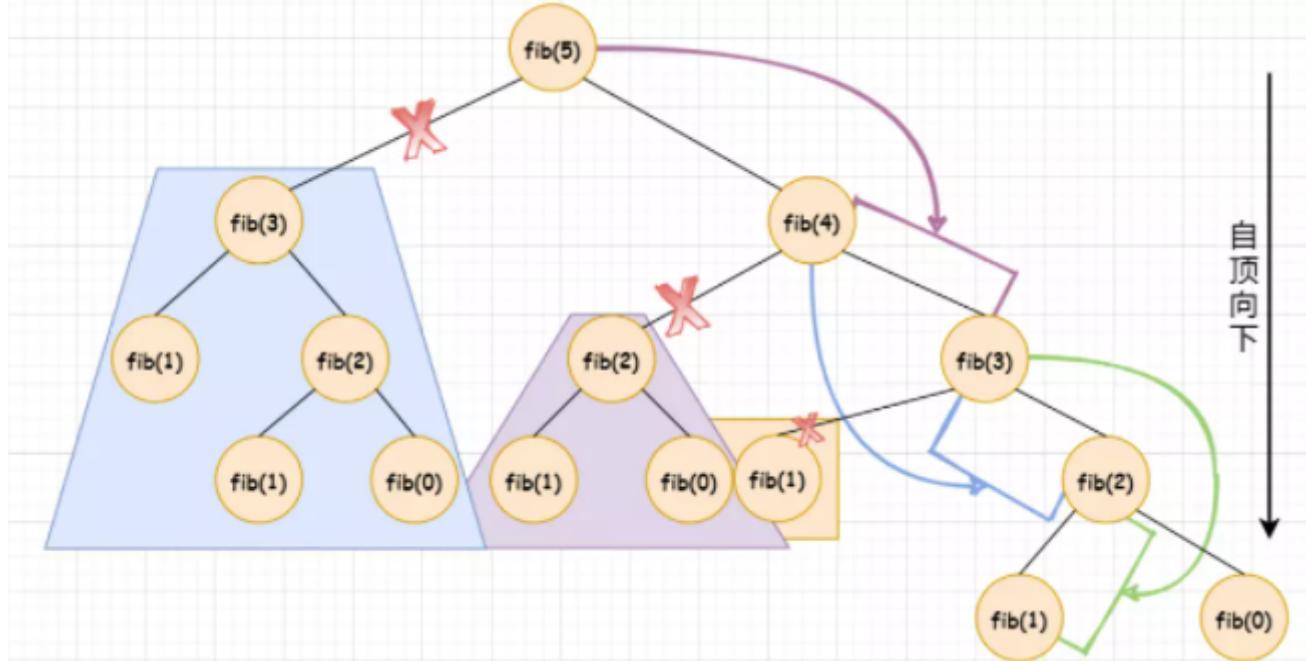
```

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

```

### 6.0.16 自顶向下与自底向上

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



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



# **Chapter 7**



# Chapter 8

## Graph

### 8.1 拓扑排序

#### 8.1.1 1857. Largest Color Value in a Directed Graph - Hard

There is a directed graph of n colored nodes and m edges. The nodes are numbered from 0 to n - 1.

You are given a string colors where colors[i] is a lowercase English letter representing the color of the ith node in this graph (0-indexed). You are also given a 2D array edges where edges[j] = [aj, bj] indicates that there is a directed edge from node aj to node bj.

A valid path in the graph is a sequence of nodes  $x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \dots \rightarrow x_k$  such that there is a directed edge from  $x_i$  to  $x_{i+1}$  for every  $1 \leq i < k$ . The color value of the path is the number of nodes that are colored the most frequently occurring color along that path.

Return the largest color value of any valid path in the given graph, or -1 if the graph contains a cycle.

##### 1. 解题思路与分析

```
public int largestPathValue(String colors, int[][] edges) {
    List<List<Integer>> adj = new ArrayList<>();
    int n = colors.length(), cnt = 0;
    int[] ins = new int[n], topo = new int[n];
    for (int i = 0; i < n; i++) adj.add(new ArrayList<>());
    for (int[] e : edges) {
        ins[e[1]]++;
        adj.get(e[0]).add(e[1]);
    }
    Queue<Integer> q = new LinkedList<>();
    for (int i = 0; i < n; i++)
        if (ins[i] == 0)
            q.offer(i);
    while (!q.isEmpty()) {
        int u = q.poll();
        topo[cnt++] = u; // 将所有的节点按照拓扑排序
        for (Integer v : adj.get(u))
            if (--ins[v] == 0)
                q.offer(v);
    }
    if (cnt < n) return -1; // 说明图中有环
    int ans = 0;
    char[] s = colors.toCharArray();
    for (int i = 0; i < 26; i++) {
        char c = (char)(i + 'a');
        int[] dp = new int[n];
        for (int j = n-1; j >= 0; j--) {
            int u = topo[j];
            for (Integer v : adj.get(u))
                dp[u] = Math.max(dp[u], dp[v]);
            if (s[u] == c) dp[u]++;
            ans = Math.max(ans, dp[u]);
        }
    }
    return ans;
}
```

- 另一种更为简洁的写法

```
public int largestPathValue(String colors, int[][] edges) {
    int n = colors.length(), ans = 0, processed = 0;
    int[] ins = new int[n];
    ArrayList<Integer>[] adj = new ArrayList[n];
    Queue<Integer> q = new LinkedList<>();
    int[][] cnt = new int[n][26];
```

```

for (int i = 0; i < n; ++i) adj[i] = new ArrayList<>();
for (int [] e : edges) {
    adj[e[0]].add(e[1]);
    ++ins[e[1]];
}
for (int i = 0; i < n; ++i)
    if (ins[i] == 0)
        q.offer(i);
char [] s = colors.toCharArray();
while (!q.isEmpty()) {
    int u = q.poll();
    ++processed;
    ans = Math.max(ans, ++cnt[u][s[u] - 'a']);
    for (int v : adj[u]) {
        for (int i = 0; i < 26; ++i)
            cnt[v][i] = Math.max(cnt[v][i], cnt[u][i]); // 这里是不是可以再简化一下？
        if (--ins[v] == 0)
            q.offer(v);
    }
}
return processed == n ? ans : -1;
}

```

### 8.1.2 1203. Sort Items by Groups Respecting Dependencies - Hard

There are  $n$  items each belonging to zero or one of  $m$  groups where  $group[i]$  is the group that the  $i$ -th item belongs to and it's equal to -1 if the  $i$ -th item belongs to no group. The items and the groups are zero indexed. A group can have no item belonging to it.

Return a sorted list of the items such that:

The items that belong to the same group are next to each other in the sorted list. There are some relations between these items where  $beforeItems[i]$  is a list containing all the items that should come before the  $i$ -th item in the sorted array (to the left of the  $i$ -th item). Return any solution if there is more than one solution and return an empty list if there is no solution.

#### 1. 解题思路与分析

```

public int[] sortItems(int n, int m, int[] g, List<List<Integer>> bef) {
    List<List<Integer>> its = new ArrayList<>(); // 下标: 元素所属的组编号; 组内元素
    List<List<Integer>> gps = new ArrayList<>(); // 组图: 下标: 组编号; 组内元素
    List<Integer> ids = new ArrayList<>();
    for (int i = 0; i < m+n; i++) {
        ids.add(i);
        its.add(new ArrayList<>());
        gps.add(new ArrayList<>());
    }
    List<List<Integer>> itg = new ArrayList<>(); // itemGraph, n items, 元素先后顺序图: 每个元素, 之后的元素包括()
    for (int i = 0; i < n; i++)
        itg.add(new ArrayList<>());
    int [] insIts = new int [n]; // 每个元素各自入度
    int [] insGps = new int [m+n]; // 每个组的各自入度
    int resId = m; // 把先前剩余的、不属于任何组的, 分别分布到只含其一个元素的编号较大 (编号不被其它元素占用 [0, m-1]) 的组中
    for (int i = 0; i < n; i++) {
        if (g[i] == -1)
            g[i] = resId++;
        its.get(g[i]).add(i); // 下标: 元素所属的组编号; 组内元素
    }
    for (int i = 0; i < n; i++) {
        int gid = g[i];
        for (int it : bef.get(i)) { // 在现元素 i 之前的元素链表
            int befGid = g[it];
            if (befGid == gid) { // 在同一个组内, 可以进行组内排序
                insIts[i]++;
                itg.get(it).add(i); // 从之前的元素可以连接到现元素 <== 会进行拓扑排序, 所有先发生的在前, 后发生的在后
            } else { // 属于不同的组, 须进行组间排序
                if (gps.get(gid).contains(befGid)) return new int [0]; // 这里发生了矛盾, 需返回
                insGps[gid]++;
                gps.get(befGid).add(gid); // 从之前的组可以连接到现组 <== 会进行拓扑排序, 所有先发生的在前, 后发生的在后
            }
        }
    }
    List<Integer> gsort = topologicalSort(insGps, gps, ids); // 组间排序
    if (gsort.size() == 0) return new int [0];
    int [] ans = new int [n];
    int i = 0;
    for (Integer gid : gsort) { // 对排好序的各组, 进行组内排序
        int size = its.get(gid).size();
        if (size == 0) continue;
        List<Integer> li = topologicalSort(insIts, itg, its.get(gid)); // 进行组内排序
        if (li.size() == 0) return new int [0];
        for (int it : li)
            ans[i++] = it;
    }
    return ans;
}

```

```

}
List<Integer> topologicalSort(int [] ins, List<List<Integer>> adj, List<Integer> li) {
    Deque<Integer> q = new ArrayDeque<>() ;
    for (Integer v : li)
        if (ins[v] == 0) q.offerLast(v);
    List<Integer> ans = new ArrayList<>();
    while (!q.isEmpty()) {
        int cur = q.pollFirst();
        ans.add(cur);
        for (Integer v : adj.get(cur))
            if (--ins[v] == 0) q.offerLast(v);
    }
    return ans;
}

```

### 8.1.3 2045. Second Minimum Time to Reach Destination - Hard

A city is represented as a bi-directional connected graph with  $n$  vertices where each vertex is labeled from 1 to  $n$  (inclusive). The edges in the graph are represented as a 2D integer array  $\text{edges}$ , where each  $\text{edges}[i] = [u_i, v_i]$  denotes a bi-directional edge between vertex  $u_i$  and vertex  $v_i$ . Every vertex pair is connected by at most one edge, and no vertex has an edge to itself. The time taken to traverse any edge is time minutes.

Each vertex has a traffic signal which changes its color from green to red and vice versa every change minutes. All signals change at the same time. You can enter a vertex at any time, but can leave a vertex only when the signal is green. You cannot wait at a vertex if the signal is green.

The second minimum value is defined as the smallest value strictly larger than the minimum value.

For example the second minimum value of  $[2, 3, 4]$  is 3, and the second minimum value of  $[2, 2, 4]$  is 4. Given  $n$ ,  $\text{edges}$ , time, and change, return the second minimum time it will take to go from vertex 1 to vertex  $n$ .

Notes:

You can go through any vertex any number of times, including 1 and  $n$ . You can assume that when the journey starts, all signals have just turned green.

Observation:

We are tasked to find 2nd minimum time to reach city  $N$ .

We can use Dijkstra's algorithm to find minimum time taken to the city  $N$ . However to find 2nd min time we need a little modification on vanilla Dijkstra. i.e. visit cities again beyond their optimal times once found

The Question now becomes ?

How many times do we need to visit any city at maximum ? - answer is 2 times :) . We don't need to care about 3rd min times and beyond for any of the cities. that's it!

Algorithm:

1. Use Dijkstra to find min elapsed time by visiting cities in order and store this time
2. when visiting the city again, check if this elapsed time is 2nd min time to previously recorded value
  - 2.1 if yes. continue evaluation
  - 2.2 if no, ignore this state and continue to next state in the heap
- 3 Also check if we already have 2 min times for the city already, if yes. make sure never evaluate for this city again.

This will optimize your solution.

```

public int secondMinimum(int n, int[][] edges, int time, int change) {
    Map<Integer, List<Integer>> adj = new HashMap<>();
    for (int [] e : edges) {
        adj.computeIfAbsent(e[0], z -> new ArrayList<>().add(e[1]));
        adj.computeIfAbsent(e[1], z -> new ArrayList<>().add(e[0]));
    }
    Queue<int []> q = new PriorityQueue<>((a, b)->(a[1] - b[1]));
    q.offer(new int []{1, 0});
    Map<Integer, Integer> cache = new HashMap<>(); // use cache temp record min time per city
    // modification: we want to visit each city maximum two times with different times,
    // this will help in early termination when we visit the city again (3rd time or more)
    Set<Integer> exhausted = new HashSet<>();
    while (!q.isEmpty()) {
        int [] top = q.poll();
        int cur = top[0], t = top[1];
        // Base Termination : we have found our 2nd min time for city n
        if (cur == n && t > cache.getOrDefault(cur, Integer.MAX_VALUE))
            return t;
        if (!cache.containsKey(cur)) // we visited this city for first time, so elapsed time is min for this city
            cache.put(cur, t);
        // early termination, if we are trying to visit the city 3rd time or more ,
        // or the elapsed time will not help in finding the solution
        else if (cache.get(cur) == t || exhausted.contains(cur)) continue;
        else // this means we are visiting the city with 2nd optimal time , we dont need to visit the city ever again
            exhausted.add(cur);
        // we visit the city on elapsedTime, we need to check if on basis of change time, whether this time falls in cycle (green or red)
        // if odd cycle (red), we must wait for this cycle to end
        int factor = t / change;
        if (factor % 2 == 1)
            t = (factor + 1) * change;
    }
}

```

```

        for (int nb : adj.getOrDefault(cur, new ArrayList<>())) { // visit the neighbours
            int visTime = t + time;
            if (!exhausted.contains(nb))
                q.offer(new int [] {nb, visTime});
        }
    }
    return -1;
}

```

- 另一个也是写得直接了当的

```

public int secondMinimum(int n, int [][] edges, int time, int change) {
    Map<Integer, Set<Integer>> map = new HashMap<>();
    for (int [] e : edges) {
        map.computeIfAbsent(e[0], z -> new HashSet<>()).add(e[1]);
        map.computeIfAbsent(e[1], z -> new HashSet<>()).add(e[0]);
    }
    Queue<int []> q = new PriorityQueue<>((a, b)->(a[1]-b[1]));
    Map<Integer, Set<Integer>> vis = new HashMap<>();
    q.offer(new int [] {1, 0});
    int min = -1;
    while (!q.isEmpty()) {
        int [] top = q.poll();
        int cur = top[0], t = top[1];
        if (cur == n) {
            if (min == -1 || min == t) min = t;
            else return t;
        }
        if (t % (2 * change) >= change)
            t += 2 * change - t % (2 * change);
        // 源码中传入 key 和 value, 根据 key 获取看是否存在 value, 如果 value==null, 然后调用 put 方法把传入的 key 和 value put 进 map, 返回根据 key 获取的老
        // 如果传入 key 对应的 value 已经存在, 就返回存在的 value, 不进行替换. 如果不存在, 就添加 key 和 value, 返回 null
        vis.putIfAbsent(cur, new HashSet<>());
        if (!vis.get(cur).add(t) || vis.get(cur).size() >= 3) continue;
        if (map.containsKey(cur))
            for (int next : map.get(cur))
                q.offer(new int [] {next, t + time});
    }
    return -1;
}

```

### 8.1.4 1334. Floyd 算法 - Find the City With the Smallest Number of Neighbors at a Threshold Distance - Medium

There are  $n$  cities numbered from 0 to  $n-1$ . Given the array edges where  $\text{edges}[i] = [\text{from}_i, \text{to}_i, \text{weight}_i]$  represents a bidirectional and weighted edge between cities  $\text{from}_i$  and  $\text{to}_i$ , and given the integer  $\text{distanceThreshold}$ .

Return the city with the smallest number of cities that are reachable through some path and whose distance is at most  $\text{distanceThreshold}$ . If there are multiple such cities, return the city with the greatest number.

Notice that the distance of a path connecting cities  $i$  and  $j$  is equal to the sum of the edges' weights along that path.

```

public int findTheCity(int n, int[][] edges, int distanceThreshold) {
    // 1. 创建邻接矩阵
    int [][] graph = new int [n][n]; // 相比于我只会用 HashMap 来建邻接关系, 邻接链表与数组都可能, 看哪个用起来方便
    for (int i = 0; i < n; i++)
        Arrays.fill(graph[i], Integer.MAX_VALUE); // pre filled n equivalent to Integer.MAX_VALUE
    for (int [] eg : edges) {
        graph[eg[0]][eg[1]] = eg[2];
        graph[eg[1]][eg[0]] = eg[2];
    }
    // 2. floyd 算法
    for (int k = 0; k < n; k++) // 中间结点
        for (int i = 0; i < n; i++) // 开始结点
            for (int j = 0; j < n; j++) // 结尾结点
                if (i == j || graph[i][k] == Integer.MAX_VALUE || graph[k][j] == Integer.MAX_VALUE) continue;
                graph[i][j] = Math.min(graph[i][j], graph[i][k] + graph[k][j]);
    // 3. 每个城市距离不大于 distanceThreshold 的邻居城市的数目
    int [] mark = new int [n]; // 记录小于 distanceThreshold 的邻居城市个数
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (graph[i][j] <= distanceThreshold)
                mark[i]++;
    // 4. 找数目少, 编号最大的
    int min = n;
    int ans = 0;
    for (int i = 0; i < n; i++)
        if (min >= mark[i]) {
            min = mark[i];
            ans = i;
        }
    return ans;
}

```

- 另一种解法

```

// 之前用原创想法也写了很多图的题，但缺乏归纳总结，原创想法更多的是解决了题目，但解法与效率、与优化算法间的距离还需要很多比较归纳与总结，才能把图这一块吃透
// https://leetcode.jp/leetcode-1334-find-the-city-with-the-smallest-number-of-neighbors-at-a-threshold-distance-%E8%A7%A3%E9%A2%98%E6%80%9D%E8%B7%AF%E5%
// map: 图结构
// city: 当前城市
// dis: 当前所剩距离
// v: 已经被记录为邻居的节点
// maxDis: 走到某个节点时，剩余距离的最大值
// 返回值为当前城市的邻居数。
private int dfs(int [][] arr, int city, int dis, boolean [] vis, int [] maxDis) {
    int res = 0;
    for (int i = 0; i < arr[0].length; i++) { // 循环当前城市的所有相邻城市
        int distance = arr[city][i]; // 与相邻城市之间的距离，如果为 0，说明与该城市不相连
        int diffDis = dis - distance; // 到达相邻城市后，与阈值相比的剩余距离。
        if (distance > 0 && diffDis >= maxDis[i]) { // 与该城市相连并且剩余距离大于等于访问数组中的值
            maxDis[i] = diffDis; // 更新访问数组中的剩余距离
            if (!vis[i]) {
                vis[i] = true;
                res++;
            }
        }
        res += dfs(arr, i, diffDis, vis, maxDis); // 递归 dfs 与该城市相连的其他城市：图中我似乎还很没有 dfs 以及递归的概念
    }
}
return res;
}
public int findTheCity(int n, int [][] edges, int distanceThreshold) {
    int [][] map = new int [n][n];
    for (int [] eg : edges) {
        map[eg[0]][eg[1]] = eg[2];
        map[eg[1]][eg[0]] = eg[2];
    }
    int min = n;
    int res = 0;
    for (int i = 0; i < n; i++) {
        boolean [] vis = new boolean [n];
        vis[i] = true;
        int cnt = dfs(map, i, distanceThreshold, vis, new int [n]);
        if (cnt <= min) {
            min = cnt;
            res = i;
        }
    }
}
return res;
}

```

### 8.1.5 1129. Shortest Path with Alternating Colors - Medium

Consider a directed graph, with nodes labelled 0, 1, ..., n-1. In this graph, each edge is either red or blue, and there could be self-edges or parallel edges.

Each [i, j] in red\_edges denotes a red directed edge from node i to node j. Similarly, each [i, j] in blue\_edges denotes a blue directed edge from node i to node j.

Return an array answer of length n, where each answer[X] is the length of the shortest path from node 0 to node X such that the edge colors alternate along the path (or -1 if such a path doesn't exist).

```

// 找最短路径应该用 queue 来做，入队列的时候需要标记红边或是蓝边以便找交替路径
public int[] shortestAlternatingPaths(int n, int [][] red_edges, int [][] blue_edges) {
    HashMap<Integer, List<Integer>> [] maps = new HashMap [2]; // 0 : red; 1: blue
    for (int i = 0; i < 2; i++)
        maps[i] = new HashMap<>();
    for (int i = 0; i < red_edges.length; i++)
        maps[0].computeIfAbsent(red_edges[i][0], k->new ArrayList<>()).add(red_edges[i][1]);
    for (int i = 0; i < blue_edges.length; i++)
        maps[1].computeIfAbsent(blue_edges[i][0], k->new ArrayList<>()).add(blue_edges[i][1]);
    int [] ans = new int[n];
    Arrays.fill(ans, -1);
    Queue<int []> q = new LinkedList<>();
    q.offer(new int [] {0, 0}); // red edge
    q.offer(new int [] {0, 1}); // blue edge
    boolean [][] inQueue = new boolean [n][2]; // 0: red, 1: blue
    inQueue[0][0] = true;
    inQueue[0][1] = true;
    int cnt = 0, color = 0;
    while (!q.isEmpty()) {
        for (int size = q.size(); size > 0; size--) {
            int [] cur = q.poll();
            System.out.println(Arrays.toString(cur));
            color = cur[1];
            if (ans[cur[0]] == -1) ans[cur[0]] = cnt;
            List<Integer> nextNodes = maps[1-color].get(cur[0]);
            if (nextNodes == null) continue;
            for (Integer next : nextNodes)
                if (!inQueue[next][1-color])
                    q.offer(new int [] {next, 1-color});
                    inQueue[next][1-color] = true;
        }
    }
}

```

```

        ++cnt;
    }
    return ans;
}

• 不是总喜欢省掉大括号吗，试试省掉下面的。。。。。。
}

public int[] shortestAlternatingPaths(int n, int[][] red_edges, int[][] blue_edges) {
    int [][] red = new int[n][2]; // 红 0 蓝 1
    int [][] blue = new int[n][2];
    for (int i = 1; i < n; i++) {
        red[i][0] = i;
        red[i][1] = 0xffffffff; // 初始化红边权值
    }
    red [0][0] = 0;
    red [0][1] = 0;
    for (int i = 1; i < n; i++) {
        blue[i][0] = i;
        blue[i][1] = 0xffffffff;
    }
    blue [0][0] = 0;
    blue [0][1] = 0;
    dfs(red, blue, 0, 0, red_edges, blue_edges);
    dfs(red, blue, 1, 0, red_edges, blue_edges);
    int [] ans = new int[n];
    for(int i = 0; i < n; i++){
        ans[i] = Math.min(red[i][1], blue[i][1]);
        if (ans[i] == 0xffffffff) // 没有改变说明不存在
            ans[i] = -1;
    }
    return ans;
}
public void dfs(int [][] red, int [][] blue, int color, int node, int[][] red_edges, int[][] blue_edges){
    if (color == 0) { // 这个括号可以省吗 ???
        for (int [] blue_to : blue.edges) // 以 node 为 from to 为终的边
            if (node == blue_to[0] && red[node][1]+1 < blue[blue_to[1]][1]) { // 0 到 from 点加 1 是否小于 0 到 to 的距离
                blue[blue_to[1]][1] = red[node][1]+1; // 作距离的更新
                dfs(red, blue, 1-color, blue_to[1], red_edges, blue_edges);
            }
    } else for (int [] red_to : red.edges) // 以 node 为 from to 为终的边
        if (node == red_to[0] && blue[node][1]+1 < red[red_to[1]][1]) { // 0 到 from 点加 1 是否小于 0 到 to 的距离
            red[red_to[1]][1] = blue[node][1]+1;
            dfs(red, blue, 1-color, red_to[1], red_edges, blue_edges);
    }
}
}

```

### 8.1.6 882. Reachable Nodes In Subdivided Graph - Hard

You are given an undirected graph (the "original graph") with  $n$  nodes labeled from 0 to  $n - 1$ . You decide to subdivide each edge in the graph into a chain of nodes, with the number of new nodes varying between each edge.

The graph is given as a 2D array of edges where  $\text{edges}[i] = [\text{ui}, \text{vi}, \text{cnti}]$  indicates that there is an edge between nodes  $\text{ui}$  and  $\text{vi}$  in the original graph, and  $\text{cnti}$  is the total number of new nodes that you will subdivide the edge into. Note that  $\text{cnti} == 0$  means you will not subdivide the edge.

To subdivide the edge  $[\text{ui}, \text{vi}]$ , replace it with  $(\text{cnti} + 1)$  new edges and  $\text{cnti}$  new nodes. The new nodes are  $\text{x}_1, \text{x}_2, \dots, \text{x}_{\text{cnti}}$ , and the new edges are  $[\text{ui}, \text{x}_1], [\text{x}_1, \text{x}_2], [\text{x}_2, \text{x}_3], \dots, [\text{x}_{\text{cnti}-1}, \text{x}_{\text{cnti}}], [\text{x}_{\text{cnti}}, \text{vi}]$ .

In this new graph, you want to know how many nodes are reachable from the node 0, where a node is reachable if the distance is  $\text{maxMoves}$  or less.

Given the original graph and  $\text{maxMoves}$ , return the number of nodes that are reachable from node 0 in the new graph.

再进一步来分析，其实上对于每个结点来说（不论有没有编号），若我们能算出该结点离起始结点的最短距离，且该距离小于等于  $M$  的话，那这个结点就一定可以到达。这样来说，其实本质就是求单源点的最短距离，此时就要祭出神器迪杰斯特拉算法 Dijkstra Algorithm 了，LeetCode 中使用了该算法的题目还有 Network Delay Time 和 The Maze II。该算法的一般形式是用一个最小堆来保存到源点的最小距离，这里我们直接统计到源点的最小距离不是很方便，可以使用一个小 trick，即用一个最大堆来统计当前结点所剩的最大步数，因为剩的步数越多，说明距离源点距离越小。由于 Dijkstra 算法是以起点为中心，向外层层扩展，直到扩展到终点为止。根据这特性，用 BFS 来实现时再好不过了，首先来建立邻接链表，这里可以使用一个  $N \times N$  的二维数组  $\text{graph}$ ，其中  $\text{graph}[i][j]$  表示从大结点  $i$  往大结点  $j$  方向会经过的小结点个数，建立邻接链表的时候对于每个 edge，要把两个方向都赋值，前面解释过了这里要当作有向图来做。然后使用一个最大堆，里面放剩余步数和结点编号组成的数对儿，把剩余步数放前面就可以默认按步数从大到小排序了，初始化时把  $\{M, 0\}$  存入最大堆。还需要一个一维数组  $\text{visited}$  来记录某个结点是否访问过。

```

public int reachableNodes(int[][] edges, int maxMoves, int n) {
    int [][] graph = new int [n][n];
    for (int i = 0; i < n; i++)
        Arrays.fill(graph[i], -1);
    for (int [] v : edges) {
        graph[v[0]][v[1]] = v[2];
        graph[v[1]][v[0]] = v[2];
    }
}

```

```

        graph[v[1]][v[0]] = v[2];
    }
    Queue<int []> q = new PriorityQueue<>((a, b) -> (b[0] - a[0]));
    boolean [] vis = new boolean [n];
    q.offer(new int [] {maxMoves, 0});
    int res = 0;
    while (!q.isEmpty()) {
        int [] cur = q.poll();
        int cnt = cur[0], u = cur[1];
        if (vis[u]) continue;
        vis[u] = true;
        ++res;
        for (int i = 0; i < n; i++) {
            if (graph[u][i] == -1) continue;
            if (cnt > graph[u][i] && !vis[i])
                q.offer(new int [] {cnt - graph[u][i]-1, i});
            graph[i][u] -= Math.min(cnt, graph[u][i]);
            res += Math.min(cnt, graph[u][i]);
        }
    }
    return res;
}

```

- 我们也可以使用 HashMap 来建立邻接链表，最后的运行速度果然要比二维数组形式的邻接链表要快一些，其他的地方都不变，参见代码如下：

```

public int reachableNodes(int[][] edges, int maxMoves, int n) {
    int res = 0;
    Map<Integer, Map<Integer, Integer>> graph = new HashMap<>();
    for (int [] v : edges) {
        graph.computeIfAbsent(v[0], k->new HashMap<>()).put(v[1], v[2]);
        graph.computeIfAbsent(v[1], k->new HashMap<>()).put(v[0], v[2]);
    }
    Queue<int []> q = new PriorityQueue<>((a, b) -> (b[0] - a[0]));
    boolean [] vis = new boolean [n];
    q.offer(new int [] {maxMoves, 0});
    while (!q.isEmpty()) {
        int [] cur = q.poll();
        int cnt = cur[0], u = cur[1];
        if (vis[u]) continue;
        vis[u] = true;
        ++res;
        for (int i = 0; i < n; i++) {
            if (!graph.containsKey(u) || !graph.get(u).containsKey(i) || graph.get(u).get(i) == -1) continue;
            if (cnt > graph.get(u).get(i) && !vis[i])
                q.offer(new int [] {cnt - graph.get(u).get(i)-1, i});
            graph.get(i).put(u, graph.get(u).get(i) - Math.min(cnt, graph.get(u).get(i)));
            res += Math.min(cnt, graph.get(u).get(i));
        }
    }
    return res;
}

```

### 8.1.7 1782. Count Pairs Of Nodes - Hard

You are given an undirected graph defined by an integer  $n$ , the number of nodes, and a 2D integer array  $\text{edges}$ , the edges in the graph, where  $\text{edges}[i] = [\text{ui}, \text{vi}]$  indicates that there is an undirected edge between  $\text{ui}$  and  $\text{vi}$ . You are also given an integer array  $\text{queries}$ .

Let  $\text{incident}(a, b)$  be defined as the number of edges that are connected to either node  $a$  or  $b$ .

The answer to the  $j$ th query is the number of pairs of nodes  $(a, b)$  that satisfy both of the following conditions:

$a < b$   $\text{incident}(a, b) > \text{queries}[j]$  Return an array  $\text{answers}$  such that  $\text{answers.length} == \text{queries.length}$  and  $\text{answers}[j]$  is the answer of the  $j$ th query.

Note that there can be multiple edges between the same two nodes.

```

// https://leetcode.com/problems/count-pairs-of-nodes/discuss/1096740/C%2B%2BJavaPython3-Two-Problems-0(q--(n-%2B-e))
public int[] countPairs(int n, int[][] edges, int[] queries) { // 别人家的思路好清晰
    int [] cnt = new int [n+1], sortedCnt = new int [n+1], ans = new int [queries.length];
    Map<Integer, Integer> [] m = new HashMap[n+1];
    for (var e : edges) {
        sortedCnt[e[0]] = cnt[e[0]] = cnt[e[0]] + 1;
        sortedCnt[e[1]] = cnt[e[1]] = cnt[e[1]] + 1;
        int min = Math.min(e[0], e[1]), max = Math.max(e[0], e[1]);
        m[min] = m[min] == null ? new HashMap<>() : m[min];
        m[min].put(max, m[min].getOrDefault(max, 0) + 1); // 仍然是当作有向图、单向图来做
    }
    Arrays.sort(sortedCnt);
    int res = 0, cur = 0;
    for (int k = 0; k < queries.length; k++) {
        for (int i = 1, j = n; i < j;) {
            if (queries[k] < sortedCnt[i] + sortedCnt[j])
                ans[k] += (j--) - i;
            else ++i;
        }
    }
}

```

```

for (int i = 1; i <= n; i++) {
    if (m[i] != null) {
        for (var en : m[i].entrySet()) {
            int j = en.getKey(), sharedCnt = en.getValue();
            if (queries[k] < cnt[i] + cnt[j] && cnt[i] + cnt[j] - sharedCnt <= queries[k])
                ans[k]--;
        }
    }
}
return ans;
}

```

### 8.1.8 2115. Find All Possible Recipes from Given Supplies

You have information about  $n$  different recipes. You are given a string array `recipes` and a 2D string array `ingredients`. The  $i$ th recipe has the name `recipes[i]`, and you can create it if you have all the needed ingredients from `ingredients[i]`. Ingredients to a recipe may need to be created from other recipes, i.e., `ingredients[i]` may contain a string that is in `recipes`.

You are also given a string array `supplies` containing all the ingredients that you initially have, and you have an infinite supply of all of them.

Return a list of all the recipes that you can create. You may return the answer in any order.

Note that two recipes may contain each other in their ingredients.

#### 1. 解题思路与分析

```

public List<String> findAllRecipes(String[] re, List<List<String>> ing, String[] sup) { // 菜谱 菜谱原材料 食材 BUG BUG BUG
    Map<String, Set<String>> adj = new HashMap<>(); // 每种材料可以做成的菜的 清单
    Map<String, Integer> ins = new HashMap<>(); // 每种材料或是菜谱的 入度
    for (int i = 0; i < re.length; i++)
        for (String it : ing.get(i)) {
            adj.computeIfAbsent(it, z -> new HashSet<>()).add(re[i]);
            ins.put(re[i], ins.getOrDefault(re[i], 0) + 1);
        }
    List<String> ans = new ArrayList<>();
    Deque<String> q = new ArrayDeque<>();
    for (String s : sup) q.offerLast(s); // 把初始的原材料放入队列
    while (!q.isEmpty()) { // 拓扑排序
        String cur = q.pollFirst();
        if (adj.containsKey(cur))
            for (String one : adj.get(cur)) { // 遍历某种原材料可以做成的所有菜，其入度是否为 0
                ins.put(one, ins.get(one)-1); // 入度 ins--
                if (ins.get(one) == 0) {
                    ans.add(one);
                    q.offerLast(one);
                }
            }
    }
}
return ans;
}

```

## 8.2 基环内向树

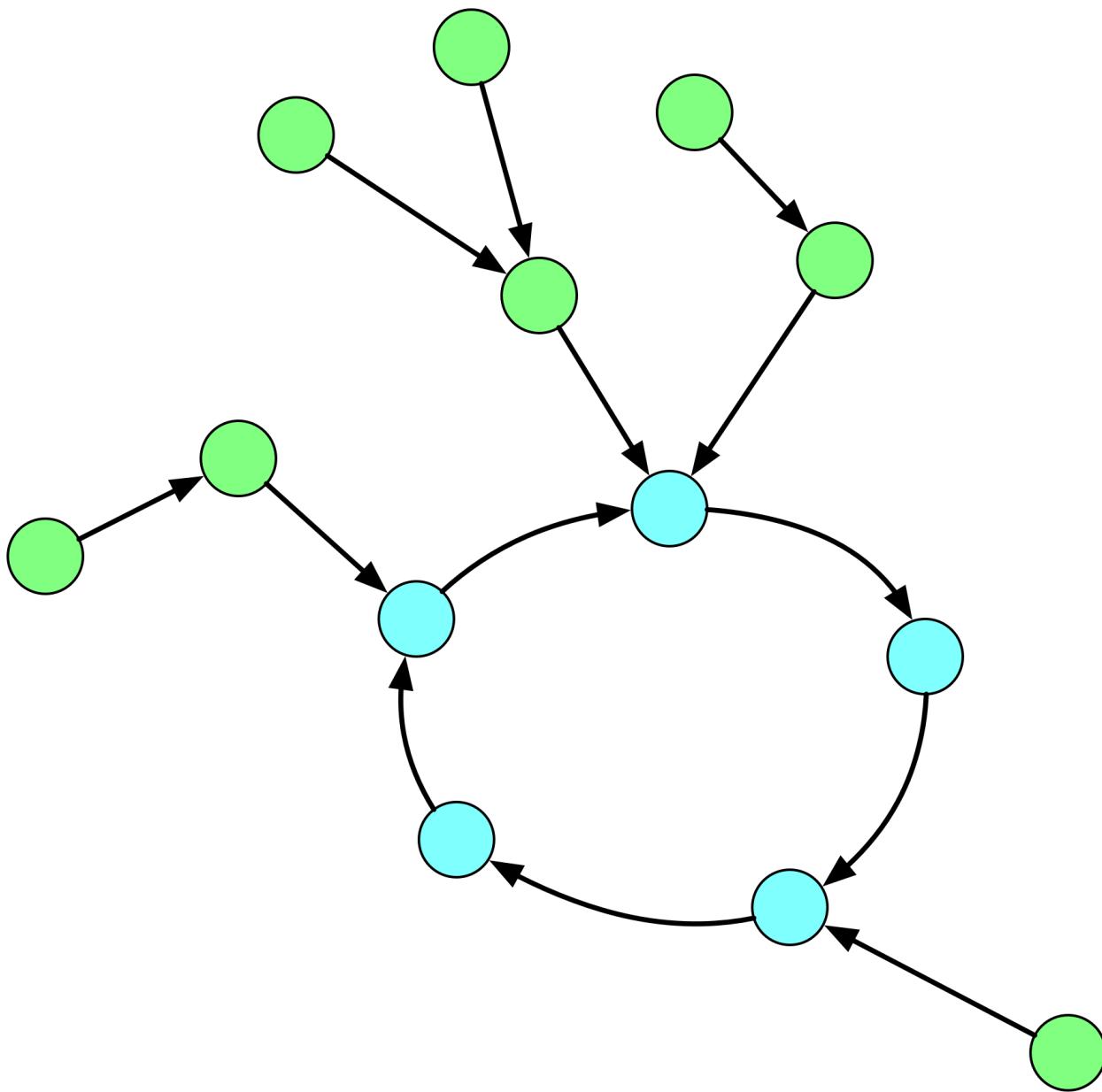
### 8.2.1 2127. Maximum Employees to Be Invited to a Meeting - Hard 基环内向树

A company is organizing a meeting and has a list of  $n$  employees, waiting to be invited. They have arranged for a large circular table, capable of seating any number of employees.

The employees are numbered from 0 to  $n - 1$ . Each employee has a favorite person and they will attend the meeting only if they can sit next to their favorite person at the table. The favorite person of an employee is not themselves.

Given a 0-indexed integer array `favorite`, where `favorite[i]` denotes the favorite person of the  $i$ th employee, return the maximum number of employees that can be invited to the meeting.

#### 1. 解题思路与分析 如果我们把每个员工看成图上的一个节点，如果员工 $xx$ 喜欢员工 $yy$ ，就在从 $xx$ 对应的节点到 $yy$ 对应的节点连一条边，那么形成的图是什么结构的？形成的图会由若干颗「基环内向树」组成。所谓「基环内向树」，就是形如下图所示的结构：



我们从任意一个节点  $xx$  开始在图上进行「游走」，由于每个员工只有一位喜欢的员工，因此每个节点在图上只会有一条出边，即「游走」的过程是唯一的。由于图上有  $nn$  个节点，因此在  $n+1n+1$  步以内，一定会走到一个重复的节点，那么在第一次经过该节点之后，到第二次经过该节点之前的所有节点，以及该节点本身，就组成了一个环，如上图的蓝色节点所示。

对于不在环上的节点，我们已经说明了从它们开始「游走」也一定会进入到环中。在到达环上的节点之前，它们不会重复经过节点（否则就有两个环了，我们可以证明一个连通分量中是不可能有两个环的：因为每个节点只有一条出边，因此如果有两个环并且它们连通，那么必然某个环上有一个点有两条出边，一条出边指向同一个环上的节点，另一条出边可以使得它到达另一个环，这就产生了矛盾），那么它们就形成了类似树的结构，如上图的绿色节点所示。

需要注意的是，一个单独的环也是「基环内向树」，它是一种特殊情况，即没有绿色的节点。

### 思路与算法

既然我们知道了图由若干颗「基环内向树」组成，那么我们就可以想一想，每一颗「基环内向树」的哪一部分可以被安排参加会议。

我们首先讨论特殊的情况，即一个单独的环（或若干个环），并且所有环的大小都  $\geq 33$ 。可以发现，我们按照环上的顺序给对应的员工安排座位是满足要求的，因为对于每一个环上的员工，它喜欢的员工就在它的旁边。并且，我们必须安排环上的所有员工，因为如果有缺失，那么喜欢那位缺失了的员工的员工就无法满足要求了。

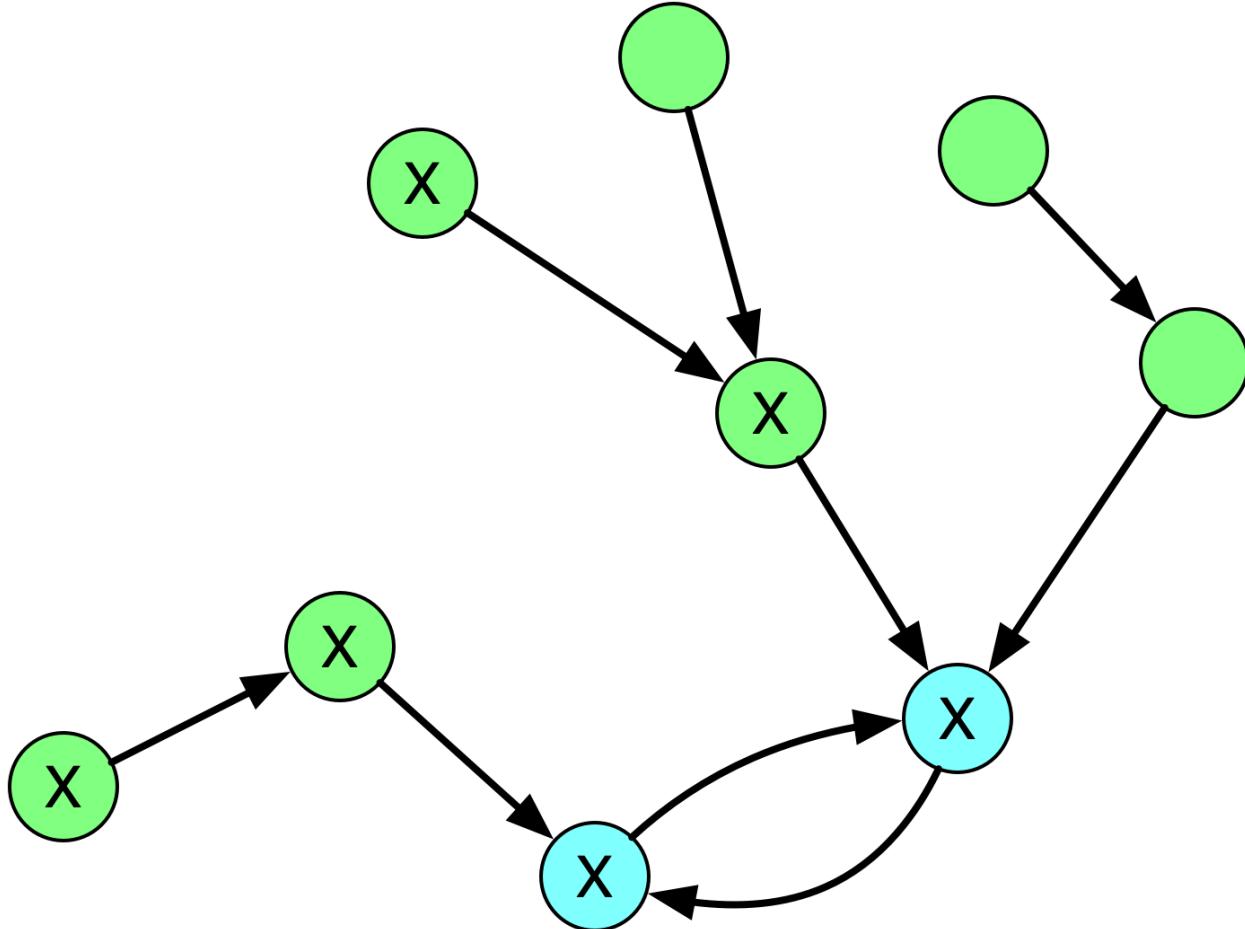
但如果我们将已经安排了某一个环上的所有员工，剩余的环就没有办法安排了。这是因为已经安排的那个环是没有办法被「断开」的：断开的本质就是相邻位置员工的缺失。因此，我们可以得出一个重要的结论：

如果我们想安排大小  $\geq 33$  的环，我们最多只能安排一个，并且环需要是完整的。

那么如果是环大小  $\geq 33$  的「基环内向树」呢？如果我们安排了不在环上的节点，那么从该节点开始，我们需要不断安排当前节点喜欢的员工，这实际上就是「游走」的过程。而当我们游走到环上并到达环上最后一个未经过的节点时，该节点的下一个节点（即喜欢的员工）已经被安排过，所以最后一个未经过的节点就无法被安排，不满足要求。因此，我们不能安排任何不在环上的节点，只能安排在环上的节点，就得出了另一个的结论：

所有环大小  $\geq 33$  的「基环内向树」与一个大小相同（指环的部分）的环是等价的。

那么最后我们只需要考虑大小  $=2=2$  的环或者「基环内向树」了。这里的特殊之处在于，大小  $=2=2$  的环可以安排多个：因为环上的两个点是互相喜欢的，因此只需要它们相邻即可，而没有其它的要求。而对于环大小  $=2=2$  的「基环内向树」，如果我们安排了不在环上的节点，那么游走完环上两个节点之后，同样是满足要求的，并且我们甚至可以继续延伸（反向「游走」），到另一个不在环上的节点为止。如下图所示，包含 XX 的节点就是可以安排参加会议的节点。



并且同样地，对于每一棵环大小  $=2=2$  的「基环内向树」，我们都可以取出这样一条「双向游走」路径进行安排，它们之间不会影响。综上所述，原问题的答案即为下面二者中的最大值：

最大的环的大小；

所有环大小  $=2=2$  的「基环内向树」上的最长的「双向游走」路径之和。

为了求解「基环内向树」上的最长的「双向游走」路径，我们可以使用拓扑排序 + 动态规划的方法。记  $f[i]$  表示到节点  $i$  为止的最长「游走」路径经过的节点个数，那么状态方程即为：

即我们考虑节点  $i$  的上一个节点  $j$ ，在图中必须有从  $j$  到  $i$  的一条有向边，这样我们就可以从  $j$  转移到  $i$ 。如果不存在满足要求的  $j$ （例如「基环内向树」退化成一个大小  $=2=2$  的环），那么  $f[i] = 1$ 。状态转移可以和拓扑排序同时进行。

在拓扑排序完成后，剩余没有被弹出过队列的节点就是环上的节点。我们可以找出每一个环。如果环的大小  $\geq 33$ ，我们就用其来更新最大的环的大小；如果环的大小  $=2=2$ ，设环上的两个节点为  $x$  和  $y$ ，那么该「基环内向树」上最长的「双向游走」的路径长度就是  $f[x] + f[y] - 1$ 。

```
public int maximumInvitations(int[] a) { // a: favorite
    // 统计入度，便于进行拓扑排序
    int n = a.length, ins[] = new int[n];
    for (int v : a) ins[v]++;
    boolean vis[] = new boolean[n];
    int f[] = new int[n];
    Arrays.fill(f, 1);
    Deque<Integer> q = new ArrayDeque<>();
```

```

for (int i = 0; i < n; i++) {
    if (ins[i] == 0) q.offerLast(i);
}
while (!q.isEmpty()) {
    int u = q.pollFirst();
    vis[u] = true;
    int v = a[u];
    f[v] = Math.max(f[v], f[u] + 1); // 动态规划：能够到达 v 的最长链的长度
    --ins[v];
    if (ins[v] == 0) q.offerLast(v);
}
// ring 表示最大的环的大小
// total 表示所有环大小为 2 的「基环内向树」上的最长的「双向游走」路径之和
int ring = 0, total = 0;
for (int i = 0; i < n; i++) {
    if (!vis[i]) {
        int j = a[i];
        if (a[j] == i) { // 说明环的大小为 2
            total += f[i] + f[j]; // 局部二元环可以叠加
            vis[i] = vis[j] = true;
        } else { // 否则环的大小至少为 3，我们需要找出环
            int u = i, cnt = 0;
            do { // 至少执行一次，evaluate after execute
                ++cnt;
                u = a[u];
                vis[u] = true;
            } while (u != i); // 再到达这一点，说明转了一圈，又回到了起点
            ring = Math.max(ring, cnt); // 找出一个节点数目最多的环
        }
    }
}
return Math.max(ring, total);
}

```

## 8.3 Tarjan 算法

- 图的一些基本概念：
- **关联 (incident)**：点为边的端点；
- **邻接 (adjacent)**：点与点关联同一条边，或边与边关联同一顶点；
- **子图**：图  $G'$  的点和边都是图  $G$  的子集，则  $G'$  为  $G$  的子图；
- **道路**：从点  $v$  到点  $u$  的路径；
  - **简单道路**：没有重复边的道路；
  - **回路**：起点与终点相同的道路；
  - **简单回路**：没有重复边的回路；
  - **连通**：两顶点间有道路；
  - **强连通**：有向图  $uv$  与  $vu$  都有道路；
  - **连通图**：任意两顶点间都有道路（若有向图除去方向后连通，则称有向图连通）；
  - **简单图**：没有重复边和自环的图；
  - **完全图**：任意两顶点间有一条边到达的简单图（有向完全图与无向完全图）；
  - **强连通 (strongly connected)**：在有向图  $G$  中，如果两个顶点间至少存在一条路径，称两个顶点强连通 (strongly connected)；
  - **强连通图**：如果有向图  $G$  的每两个顶点都强连通，称  $G$  是一个强连通图；
  - **强连通分量 (strongly connected components)**：非强连通图有向图的极大强连通子图，称为强连通分量 (strongly connected components)。
- **无向图的割点与桥**
  - 什么是无向图？简单来说，若一个图中每条边都是无方向的，则称为无向图。
  - **割点**：若从图中删除节点  $x$  以及所有与  $x$  关联的边之后，图将被分成两个或两个以上的不相连的子图，那么称  $x$  为图的割点。
  - **桥**：若从图中删除边  $e$  之后，图将分裂成两个不相连的子图，那么称  $e$  为图的桥或割边。
- 求强连通分量就是我们今天要解决的问题，根据强连通分量定义，用双向遍历取交集的方法求强连通分量，时间复杂度为  $O(N^2 + M)$ 。而 Tarjan 或 Kosaraju 算法，两者的时间复杂度都是  $O(N + M)$ 。

### 8.3.1 算法简介

在了解了 Tarjan 算法的背景以及图的割点与桥的基本概念之后，我们下面所面临的问题就是——如何求解图的割点与桥？开门见山，我们直接引出 Tarjan 算法在求解无向图的割点与桥的工作原理。

- 时间戳：时间戳是用来标记图中每个节点在进行深度优先搜索时被访问的时间顺序，当然，你可以理解成一个序号（这个序号由小到大），用  $\text{dfn}[x]$  来表示。
- 搜索树：在无向图中，我们以某一个节点  $x$  出发进行深度优先搜索，每一个节点只访问一次，所有被访问过的节点与边构成一棵树，我们可以称之为“无向连通图的搜索树”。
- 追溯值：追溯值用来表示从当前节点  $x$  作为搜索树的根节点出发，能够访问到的所有节点中，时间戳最小的值—— $\text{low}[x]$ 。那么，我们要限定下什么是“能够访问到的所有节点”？其需要满足下面的条件之一即可：
  - 以  $x$  为根的搜索树的所有节点
  - 通过一条非搜索树上的边，能够到达搜索树的所有节点

Tarjan 算法是基于对图深度优先搜索的算法，每个强连通分量为搜索树中的一棵子树。搜索时，把当前搜索树中未处理的节点加入一个堆栈，回溯时可以判断栈顶到栈中的节点是否为一个强连通分量。

- 定义：
  - $\text{DFN}(u)$  为节点  $u$  搜索的次序编号（时间戳）；
  - $\text{LOW}(u)$  为  $u$  或  $u$  的子树能够追溯到的最早的栈中节点的次序号；

由定义可以得出，当  $\text{DFN}(u)=\text{LOW}(u)$  时，以  $u$  为根的搜索子树上所有节点是一个强连通分量。

- 算法：
  - 当首次搜索到点  $u$  时  $\text{DFN}[u]=\text{LOW}[u]=\text{time}$ ；
  - 每当搜索到一个点，把该点压入栈顶；
  - 当  $u$  和  $v$  有边相连时：
    - 1) 如果  $v$  不在栈中（树枝边）， $\text{DFS}(v)$ ，并且  $\text{LOW}[u] = \min\{\text{LOW}(u), \text{LOW}(v)\}$ ；
    - 2) 如果  $v$  在栈中（前向边/后向边），此时  $\text{LOW}[u] = \min\{\text{LOW}[u], \text{DFN}[v]\}$
- 当  $\text{DFN}[u]=\text{LOW}[u]$  时，将它以及在它之上的元素弹出栈，此时，弹出栈的结点构成一个强连通分量；
- 继续搜索，知道图被遍历完毕。

由于在这个过程中每个点只被访问一次，每条边也只被访问一次，所以 Tarjan 算法的时间复杂度是  $O(n+m)$ 。

- 这个算法需要用到好几个辅助数组，下面我来详细介绍它们的作用
  - `int dfn[MAXN];`// 用来记录一个顶点第一次被访问时的时间戳
  - `int low[MAXN];`// 用来记录一个顶点不经过它的父亲顶点最高能访问到它的祖先节点中的最长时间戳，通俗易懂的来说，就是与结点  $i$  连接的所有点中  $\text{dfn}[]$  值最小的一个。
  - `int cut[MAXN];`// 用来记录该点是否是割点，因为一个割点可能多次被记录

### 8.3.2 1192. Critical Connections in a Network- Hard Tarjan 算法 Tarjan's algorithm Kosaraju 算法 - todo: 这个题不太懂

There are  $n$  servers numbered from 0 to  $n - 1$  connected by undirected server-to-server connections forming a network where  $\text{connections}[i] = [a_i, b_i]$  represents a connection between servers  $a_i$  and  $b_i$ . 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.

#### 1. 解题思路与分析

- <https://www.cnblogs.com/nullzx/p/7968110.html>

```

public List<List<Integer>> criticalConnections(int n, List<List<Integer>> connections) {
    depth = new int [n];
    Arrays.fill(depth, -1);
    adj = new ArrayList[n]; // 初始化结构图 map[i] 代表节点 i 可以连通哪些节点
    for (int i = 0; i < n; i++) adj[i] = new ArrayList<>();
    for (List<Integer> c : connections) {
        adj[c.get(0)].add(c.get(1));
        adj[c.get(1)].add(c.get(0));
    }
    dfs(0, 0, 0);
    return ans;
}
List<List<Integer>> ans = new ArrayList<>();
List<Integer> [] adj;
int [] depth;
int dfs(int cur, int pre, int dep) { // 返回值为当前节点所有 dfs 路径终点的最小深度
    depth[cur] = dep; // 将当前深度存入深度数组
    int res = Integer.MAX_VALUE;
    for (int v : adj[cur]) {
        if (v == pre) continue;
        int endDepth; // dfs 终点深度
        if (depth[v] == -1) {
            endDepth = dfs(v, cur, dep + 1);
            // 如果深度大于当前深度，说明当前点不在闭环上，当前点与下一节点 i 之间的连线为答案之一
            if (endDepth > dep)
                ans.add(List.of(cur, v));
        } else endDepth = depth[v];
        res = Math.min(res, endDepth);
    }
    return res;
}

```

## 8.4 无向图

### 8.4.1 2508. Add Edges to Make Degrees of All Nodes Even

There is an undirected graph consisting of  $n$  nodes numbered from 1 to  $n$ . You are given the integer  $n$  and a 2D array  $\text{edges}[i] = [\text{ai}, \text{bi}]$  indicates that there is an edge between nodes  $\text{ai}$  and  $\text{bi}$ . The graph can be disconnected.

You can add at most two additional edges (possibly none) to this graph so that there are no repeated edges and no self-loops.

Return true if it is possible to make the degree of each node in the graph even, otherwise return false.

The degree of a node is the number of edges connected to it.

#### 1. 怎么才能把问题分析透彻，看清问题考查的本质

把度数为奇数的节点记到  $odd$  中，记  $m$  为  $odd$  的长度，分类讨论：

- 如果  $m = 0$ ，那么已经符合要求。
- 如果  $m = 2$ ，记  $x = odd[0], y = odd[1]$ ：
  - 如果  $x$  和  $y$  之间没有边，那么连边之后就符合要求了。
  - 如果  $x$  和  $y$  之间有边，那么枚举  $[1, n]$  的所有不为  $x$  和  $y$  的点  $i$ ，由于  $i$  的度数一定是偶数，如果  $i$  和  $x$  以及  $i$  和  $y$  之间没有边，那么连边之后就符合要求了。
- 如果  $m = 4$ ，记  $a = odd[0], b = odd[1], c = odd[2], d = odd[3]$ ：
  - 如果  $a$  和  $b$  以及  $c$  和  $d$  之间没有边，那么连边之后就符合要求了。
  - 如果  $a$  和  $c$  以及  $b$  和  $d$  之间没有边，那么连边之后就符合要求了。
  - 如果  $a$  和  $d$  以及  $b$  和  $c$  之间没有边，那么连边之后就符合要求了。
- 其余情况无法满足要求。

```

public boolean isPossible(int n, List<List<Integer>> edges) {
    // 【建图】
    var g = new Set[n+1];
    Arrays.setAll(g, e -> new HashSet<Integer>()); // 初始化
    for (var e : edges) {
        int u = e.get(0), v = e.get(1);
        g[u].add(v);
        g[v].add(u);
    }
    // 【数与列举：度数为奇数的节点】
    var odd = new ArrayList<Integer>();

```

```

for (int i = 1; i <= n; i++)
    if (g[i].size() % 2 > 0) odd.add(i);
var m = odd.size();
// 【对度数为奇数的节点，结果，进行分析】
if (m == 0) return true;
if (m == 2) {
    int x = odd.get(0), y = odd.get(1);
    if (!g[x].contains(y)) return true;
    for (int i = 1; i <= n; i++)
        if (i != x && i != y && !g[x].contains(i) && !g[y].contains(i)) return true;
    return false;
}
if (m == 4) {
    int x = odd.get(0), y = odd.get(1), i = odd.get(2), j = odd.get(3);
    return !g[x].contains(y) && !g[i].contains(j) // [a, b], [c, d]
        || !g[x].contains(i) && !g[y].contains(j) // [a, c], [b, d]
        || !g[x].contains(j) && !g[y].contains(i); // [a, d], [b, c]
}
return false;
}

```

## 8.4.2 2538. Difference Between Maximum and Minimum Price Sum

There exists an undirected and initially unrooted tree with  $n$  nodes indexed from 0 to  $n - 1$ . You are given the integer  $n$  and a 2D integer array edges of length  $n - 1$ , where  $\text{edges}[i] = [a_i, b_i]$  indicates that there is an edge between nodes  $a_i$  and  $b_i$  in the tree.

Each node has an associated price. You are given an integer array price, where  $\text{price}[i]$  is the price of the  $i$ th node.

The price sum of a given path is the sum of the prices of all nodes lying on that path.

The tree can be rooted at any node root of your choice. The incurred cost after choosing root is the difference between the maximum and minimum price sum amongst all paths starting at root.

Return the maximum possible cost amongst all possible root choices.

### 1. 基本分析

#### 提示 1

由于价值都是正数，因此价值和最小的一条路径一定只有一个点。

#### 提示 2

根据提示 1，「价值和最大的一条路径与最小的一条路径的差值」等价于「去掉路径的一个端点」。

#### 提示 3

由于价值都是正数，一条路径能延长就尽量延长，这样路径和就越大，那么最优是延长到叶子。

根据提示 2，问题转换成去掉一个叶子后的最大路径和（这里的叶子严格来说是度为 1 的点，因为根的度数也可能是 1）。

#### 提示 4

最大路径和是一个经典树形 DP 问题，类似「树的直径」。由于我们需要去掉一个叶子，那么可以让子树返回两个值：

- 带叶子的最大路径和；
- 不带叶子的最大路径和。

对于当前节点，它有多颗子树，我们一颗颗 DFS，假设当前 DFS 完了其中一颗子树，它返回了「当前带叶子的路径和」和「当前不带叶子的路径和」，那么答案有两种情况：

- 前面最大带叶子的路径和 + 当前不带叶子的路径和；
- 前面最大不带叶子的路径和 + 当前带叶子的路径和；

然后更新「最大带叶子的路径和」和「最大不带叶子的路径和」。

最后返回「最大带叶子的路径和」和「最大不带叶子的路径和」，用来供父节点计算。

```

public long maxOutput(int n, int[][] edges, int[] price) {
    g = new ArrayList[n];
    Arrays.setAll(g, x -> new ArrayList<>());
    for (int[] e : edges) {

```

```

        int x = e[0], y = e[1];
        g[x].add(y);
        g[y].add(x);
    }
    dfs(0, -1, price);
    return ans;
}
List<Integer> [] g;
long ans;
// 返回【带叶子的最大路径和, 不带叶子的最大路径和】
long [] dfs(int x, int fa, int [] price) {
    long p = price[x], maxSone = p, maxStwo = 0;
    for (int v : g[x])
        if (v != fa) {
            var res = dfs(v, x, price);
            long s = res[0], ss = res[1];
            // 前面最大 带叶子的路径和 + 当前不带叶子的路径和【 带 + 不带】
            // 前面最大不带叶子的路径和 + 当前 带叶子的路径和【不带 + 带】
            ans = Math.max(ans, Math.max(maxSone + ss, maxStwo + s));
            maxSone = Math.max(maxSone, s + p);
            maxStwo = Math.max(maxStwo, ss + p); // 这里加上 p 是因为 x 必然不是叶子【什么情况下是叶了的情况, 我没想清楚】
        }
    return new long [] {maxSone, maxStwo};
}

```

## 8.5 欧拉回路: Hierholzer 算法, Fleury 算法

- AOV&AOE
- AOVAOV 网, 顶点表示活动, 弧表示活动间的优先关系的有向图。即如果  $a \rightarrow b$ , 那么  $a$  是  $b$  的先决条件。
- AOEAOE 网, 边表示活动, 是一个带权的有向无环图, 其中顶点表示事件, 弧表示活动, 权表示活动持续时间。

求拓扑序列就是 AOVAOV, 求关键路径就是 AOEAOE

入度: 入度 (indegree) 就是有向图中指向这个点的边的数量, 即有向图的某个顶点作为终点的次数和

出度: 出度 (outdegree) 就是从这个点出去的边的数量, 即有向图的某个顶点作为起点的次数和

- 定义

- 欧拉回路 (Eulerian Circuit): 从图上一个点  $u$  出发不重复地经过每一条边后, 再次回到点  $u$  的一条路径。
- 欧拉路径 (Eulerian Path) : 从图上一个点  $u$  出发不重复地经过每一条边的一条路径 (不必回到点  $u$ )。
- 欧拉图即存在欧拉回路的图, 半欧拉图即存在欧拉路径的图
- 欧拉迹/欧拉通路/一笔画: 通过图中每条边且行遍所有顶点的迹 (每条边恰一次的途径), 称为欧拉迹 (Euler trail)
- 半欧拉图: 具有欧拉通路但不具有欧拉回路的无向图称为半欧拉图, 有且仅有两个度数为奇数的结点
- 环游: 图的环游 (tour) 是指经过图的每条边至少一次的闭途径
- 欧拉环游/回路: 经过每条边恰好一次的环游/回路欧拉环游/回路 (Eular tour)
- 欧拉图: 一个图若包含欧拉环游, 则称为欧拉图 (Euleriangraph)
- 欧拉定理: 一个非空连通图是欧拉图当且仅当它的每个顶点的度数都是偶数
- 通过图中所有边恰好一次且行遍所有顶点的通路称为 欧拉通路。
- 通过图中所有边恰好一次且行遍所有顶点的回路称为 欧拉回路。
- 具有欧拉回路的无向图称为 欧拉图。
- 具有欧拉通路但不具有欧拉回路的无向图称为 半欧拉图。

就像是一笔画, 要求每条边只走一次, 但每个点可以多次经过, 而要求每个点只走一次的模型是哈密顿环注意欧拉回路必须回到起点, 欧拉路径则不必, 可以说欧拉回路一定是欧拉路径, 反之不成立

	欧拉回路	欧拉路径
无向图	每个节点都有偶数的度	每个节点都有偶数的度或只有两个节点有奇数的度 (这两个奇数度的节点是起点和终点)
有向图	每个节点都有相同的入度和出度	最多只有一个顶点的入度-出度 = 1 并且最多只有一个顶点的出度-入度 = 1, 其他节点的入度=出度

- 其他结论

- 无向图为 (半) 欧拉图时, 只需用 1 笔画成; 无向图为非 (半) 欧拉图时, 即奇点 (度为奇数的点) 数  $k > 2$ , 需用  $k/2$  笔画成。
- 可以用加边的方式把一个非欧拉图变成欧拉图。对于无向图来说, 每个奇点都需加一个度, 加的边为奇点数/2; 对于有向图来说, 每个点都需加上入度与出度之差, 加的边数为每个点入度与出度之差的绝对值之和再除以 2。

### 8.5.1 753. Cracking the Safe - Hard

There is a safe protected by a password. The password is a sequence of  $n$  digits where each digit can be in the range  $[0, k - 1]$ .

The safe has a peculiar way of checking the password. When you enter in a sequence, it checks the most recent  $n$  digits that were entered each time you type a digit.

For example, the correct password is "345" and you enter in "012345":  
After typing 0, the most recent 3 digits is "0", which is incorrect.  
After typing 1, the most recent 3 digits is "01", which is incorrect.  
After typing 2, the most recent 3 digits is "012", which is incorrect.  
After typing 3, the most recent 3 digits is "123", which is incorrect.  
After typing 4, the most recent 3 digits is "234", which is incorrect.  
After typing 5, the most recent 3 digits is "345", which is correct and the safe unlocks.

Return any string of minimum length that will unlock the safe at some point of entering it.

#### 1. 解题思路与分析: Hierholzer 算法

Hierholzer 算法可以在一个欧拉图中找出欧拉回路。

##### 方法一：Hierholzer 算法

Hierholzer 算法可以在一个欧拉图中找出欧拉回路。

具体地，我们将所有的  $n - 1$  位数作为节点，共有  $k^{n-1}$  个节点，每个节点有  $k$  条入边和出边。如果当前节点对应的数为  $a_1a_2 \dots a_{n-1}$ ，那么它的第  $x$  条出边就连向数  $a_2 \dots a_{n-1}x$  对应的节点。这样我们从一个节点顺着第  $x$  条边走到另一个节点，就相当于输入了数字  $x$ 。

在某个节点对应的数的末尾放上它的某条出边的编号，就形成了一个  $n$  位数，并且每个节点都能用这样的方式形成  $k$  个  $n$  位数。

例如  $k = 4$ ,  $n = 3$  时，节点分别为 00, 01, 02, …, 32, 33，每个节点的出边的编号分别为 0, 1, 2, 3，那么 00 和它的出边形成了 000, 001, 002, 003 这 4 个 3 位数，32 和它的出边形成了 320, 321, 322, 323 这 4 个 3 位数。

这样共计有  $k^{n-1} \times k = k^n$  个  $n$  位数，恰好就是所有可能的密码。

由于这个图的每个节点都有  $k$  条入边和出边，因此它一定存在一个欧拉回路，即可以从任意一个节点开始，一次性不重复地走完所有的边且回到该节点。因此，我们可以用 Hierholzer 算法找出这条欧拉回路：

设起始节点对应的数为  $u$ ，欧拉回路中每条边的编号为  $x_1, x_2, x_3, \dots$ ，那么最终的字符串即为

$$u \ x_1 \ x_2 \ x_3 \ \dots$$

Hierholzer 算法如下：

- 我们从节点  $u$  开始，任意地经过还未经过的边，直到我们「无路可走」。此时我们一定回到了节点  $u$ ，这是因为所有节点的入度和出度都相等。
- 回到节点  $u$  之后，我们得到了一条从  $u$  开始到  $u$  结束的回路，这条回路上仍然有些节点有未经过的出边。我么从某个这样的节点  $v$  开始，继续得到一条从  $v$  开始到  $v$  结束的回路，再嵌入之前的回路中，即

$$u \rightarrow \dots \rightarrow v \rightarrow \dots \rightarrow u$$

变为

$$u \rightarrow \dots \rightarrow v \rightarrow \dots \rightarrow v \rightarrow \dots \rightarrow u$$

以此类推，直到没有节点有未经过的出边，此时我们就找到了一条欧拉回路。

实际的代码编写具有一定的技巧性。

由于这个图的每个节点都有  $k$  条入边和出边，因此它一定存在一个欧拉回路，即可以从任意一个节点开始，一次性不重复地走完所有的边且回到该节点。因此，我们可以用 Hierholzer 算法找出这条欧拉回路：我们从节点  $u$  开始，任意地经过还未经过的边，直到我们「无路可走」。此时我们一定回到了节点  $u$ ，这是因为所有节点的入度和出度都相等。

回到节点  $u$  之后，我们得到了一条从  $u$  开始到  $u$  结束的回路，这条回路上仍然有些节点有未经过的出边。我么从某个这样的节点  $v$  开始，继续得到一条从  $v$  开始到  $v$  结束的回路，再嵌入之前的回路中，即  $u \rightarrow \dots \rightarrow v \rightarrow \dots \rightarrow u$  变为  $u \rightarrow \dots \rightarrow v \rightarrow \dots \rightarrow v \rightarrow \dots \rightarrow u$

```
Set<Integer> seen = new HashSet<Integer>();
StringBuffer ans = new StringBuffer();
int highest;
int k;
public String crackSafe(int n, int k) {
    highest = (int) Math.pow(10, n - 1);
    this.k = k;
    dfs(0);
    for (int i = 1; i < n; i++)
        ans.append('0');
    return ans.toString();
}
public void dfs(int node) {
    for (int x = 0; x < k; ++x) {
        int nei = node * 10 + x;
        if (!seen.contains(nei)) {
            seen.add(nei);
            dfs(nei % highest);
            ans.append(x); // 这里 dfs 之后才添加的顺序很重要
        }
    }
}
```

## 2. 解题思路与分析

密码共有  $n$  位，每一个位可以有  $k$  个数字，总共不同的密码总数就有  $k$  的  $n$  次方个。思路是先从  $n$  位都是 0 的密码开始，取出钥匙串的最后  $n-1$  个数字，然后在后面依次添加其他数字，用一个 HashSet 来记录所有遍历过的密码，这样如果不在此集合中，说明是一个新密码，而生成这个新密码也只是多加了一个数字，能保证钥匙串最短，这是一种贪婪的解法，相当的巧妙

```
public String crackSafe(int n, int k) {
    int N = (int) Math.pow(k, n); // 第个别位有  $k$  种取值，总共有  $k^n$  种不同的状态
    String s = "0".repeat(n);
    Set<String> ss = new HashSet<>(List.of(s.toString()));
    for (int i = 0; i < N; i++) {
        String pre = s.substring(s.length() - (n-1));
        // for (int j = 0; j < k; j++) { // 这里需要倒回来
        for (int j = k-1; j >= 0; j--) {
            String cur = pre + String.valueOf(j);
            if (!ss.contains(cur)) {
                ss.add(cur);
                s += "" + j;
                break;
            }
        }
    }
    return s;
}
```

其实在初看 Hierholzer 算法时，很容易产生一种想法，就是我只需要从一个节点遍历，每次把它经历的边加入到结果字符串中，当回到初始点时就完成一圈，但是这样实现的话有个明显的问题，就是每个节点都有自环，如果你遍历到某个节点时，直接跳过了自环，去了其他节点，那就失去了回来的机会（想想回家的时候虽然你可以绕小路，也可以走大路，但只要你走大路到家了，就不可能再回到学校从小路回家）。实际上不只是自环，还有可能有其他边没循环到，因为回到自身路径过多，很多边都可能没有利用。

而官方题解中的 dfs 巧妙的解决了这个问题，实际上它不只是沿着边走，而是把每一个边的组合都遍历到，并且在遍历之后才将有用的节点嵌套到字符串中。在 dfs 中，每次循环时，并不是直接将该边加入到字符串中，而是在循环之后，实际上可以想成是用了一个栈，反序的将合法的序列弹出了（dfs 中的每次循环都会探索一个节点能到达的结尾在哪里，并且因为记录了每一条删除的边，所以其并不会走之前走过的路，找到结尾后回溯到还有边可走的点，继续向下走，而在该点所有可行边都已回溯完毕后，才到他自己，所以其实所有边都已经到达，并且顺序是逆序）。所以在主函数中，在得到整个序列后，才将初始的节点放入字符串末尾（如果正序的话，你应该将它放到字符串的开头）。

## 3. 解题思路与分析: 递归写法

来看同一种解法的递归写法，思路和迭代的写法一模一样，写法略有不同而已

```
public String crackSafe(int n, int k) {
    N = (int) Math.pow(k, n); // 第个别位有  $k$  种取值，总共有  $k^n$  种不同的状态
    s = "0".repeat(n);
    Set<String> ss = new HashSet<>(List.of(s.toString()));
    dfs(ss, n, k);
```

```

    return s;
}
String s;
int N;
void dfs(Set<String> ss, int n, int k) {
    if (ss.size() == N) return;
    String pre = s.substring(s.length() - (n-1));
    for (int i = k-1; i >= 0; i--) {
        String cur = pre + i;
        if (ss.contains(cur)) continue;
        s += "" + i;
        ss.add(cur);
        dfs(ss, n, k);
    }
}

```

### 8.5.2 332. Reconstruct Itinerary - Medium 欧拉回路 Hierholzer 算法

You are given a list of airline tickets where  $\text{tickets}[i] = [\text{from}_i, \text{to}_i]$  represent the departure and the arrival airports of one flight. Reconstruct the itinerary in order and return it.

All of the tickets belong to a man who departs from "JFK", thus, the itinerary must begin with "JFK". If there are multiple valid itineraries, you should return the itinerary that has the smallest lexical order when read as a single string.

For example, the itinerary ["JFK", "LGA"] has a smaller lexical order than ["JFK", "LGB"]. You may assume all tickets form at least one valid itinerary. You must use all the tickets once and only once.

#### 1. 解题思路与分析

我们化简本题题意：给定一个  $n$  个点  $m$  条边的图，要求从指定的顶点出发，经过所有的边恰好一次（可以理解为给定起点的「一笔画」问题），使得路径的字典序最小。

- 这种「一笔画」问题与欧拉图或者半欧拉图有着紧密的联系，下面给出定义：
  - 通过图中所有边恰好一次且行遍所有顶点的通路称为 **欧拉通路**。
  - 通过图中所有边恰好一次且行遍所有顶点的回路称为 **欧拉回路**。
  - 具有欧拉回路的无向图称为 **欧拉图**。
  - 具有欧拉通路但不具有欧拉回路的无向图称为 **半欧拉图**。

因为本题保证至少存在一种合理的路径，也就告诉了我们，这张图是一个欧拉图或者半欧拉图。我们只需要输出这条欧拉通路的路径即可。

- 如果没有保证至少存在一种合理的路径，我们需要判别这张图是否是欧拉图或者半欧拉图，具体地：
  - 对于无向图  $G$ ,  $G$  是欧拉图当且仅当  $G$  是连通的且没有奇度顶点。
  - 对于无向图  $G$ ,  $G$  是半欧拉图当且仅当  $G$  是连通的且  $G$  中恰有 2 个奇度顶点。
  - 对于有向图  $G$ ,  $G$  是欧拉图当且仅当  $G$  的所有顶点属于同一个强连通分量且每个顶点的入度和出度相同。
  - 对于有向图  $G$ ,  $G$  是半欧拉图当且仅当  $G$  的所有顶点属于同一个强连通分量且
    - \* 恰有一个顶点的出度与入度差为 1;
    - \* 恰有一个顶点的入度与出度差为 1;
    - \* 所有其他顶点的入度和出度相同。

#### 2. 解题思路与分析: Hierholzer 算法

- Hierholzer 算法用于在连通图中寻找欧拉路径，其流程如下：
  - 从起点出发，进行深度优先搜索。
  - 每次沿着某条边从某个顶点移动到另外一个顶点的时候，都需要删除这条边。
  - 如果没有可移动的路径，则将所在节点加入到栈中，并返回。

当我们顺序地考虑该问题时，我们也许很难解决该问题，因为我们无法判断当前节点的哪一个分支是「死胡同」分支。不妨倒过来思考。我们注意到只有那个入度与出度差为 1 的节点会导致死胡同。而该节点必然是最后一个遍历到的节点。我们可以改变入栈的规则，当我们遍历完一个节点所连的所有节点后，我们才将该节点入栈（即逆序入栈）。

对于当前节点而言，从它的每一个非「死胡同」分支出发进行深度优先搜索，都将会搜回到当前节点。而从它的「死胡同」分支出发进行深度优先搜索将不会搜回到当前节点。也就是说当前节点的死胡同分支将会优先于其他非「死胡同」分支入栈。

这样就能保证我们可以「一笔画」地走完所有边，最终的栈中逆序地保存了「一笔画」的结果。我们只要将栈中的内容反转，即可得到答案。

```

public List<String> findItinerary(List<List<String>> tickets) {
    for (List<String> t : tickets)
        m.computeIfAbsent(t.get(0), z -> new PriorityQueue<String>()).offer(t.get(1));
    List<String> ans = new ArrayList<String>();
    dfs("JFK", ans);
    Collections.reverse(ans);
    return ans;
}
Map<String, PriorityQueue<String>> m = new HashMap<String, PriorityQueue<String>>(); // PriorityQueue 已经默认是最小字典序, 免去了排序的操作
void dfs(String s, List<String> l) {
    Queue<String> next = m.get(s);
    while (next != null && next.size() > 0)
        dfs(next.poll(), l);
    l.add(s);
}

```

### 3. 解题思路与分析: Hierholzer 算法, 同上, 但用 LinkedList 可以从头插入

Greedy DFS, building the route backwards when retreating.

这题其实和我之前用 DFS 处理 topological sort 的代码非常像, 主要区别在于存 graph 的方式不同, 这里是一个 String 直接连着对应的 next nodes, 而且形式是 min heap:

- 原题给的是 edges, 所以图是自己用 hashmap 建的。
  - min heap 可以自动保证先访问 lexicographical order 较小的;
  - 同时 poll 出来的 node 自动删除, 免去了用 List 的话要先 collections.sort 再 remove 的麻烦。
  - 这种以 “edge” 为重心的算法多靠 heap, 比如 dijkstra.

Hierholzer 算法的精髓是当每次访问一条边的时候, 删除这条边, 当遍历完一个节点所连的所有节点后, 才将该节点入栈, 最后将栈中的节点反转, 即可得到欧拉路径

```

public List<String> findItinerary(String[][] tickets) {
    LinkedList<String> ans = new LinkedList<String>();
    for (String[] t : tickets)
        map.computeIfAbsent(t.get(0), z -> new ArrayList<String>()).offer(t.get(1));
    dfs("JFK", ans);
    return new ArrayList<String>(ans); // LinkedList 最后需要转换成 ArrayList
}
HashMap<String, PriorityQueue<String>> map = new HashMap<String, PriorityQueue<String>>();
void dfs(String airport, LinkedList<String> list) {
    while (map.containsKey(airport) && !map.get(airport).isEmpty())
        dfs(map.get(airport).poll(), list);
    list.offerFirst(airport); // LinkedList 可以这么写
}

```

### 4. 解题思路与分析: Fleury 算法: leetcode 还有一道割点割边的题, 找出来 todo

- 一些概念:
  - 割点: 在一个无向图中, 如果有一个顶点集合, 删除这个顶点集合以及这个集合中所有顶点相关联的边以后, 图的连通分量增多, 就称这个点集为割点集合, 如果某个割点集合只含有一个顶点 X (也即 {X} 是一个割点集合), 那么 X 称为一个割点
  - 割边: 在一个无向图中, 如果有一个边集合, 删除这个边集合以后, 图的连通分量增多, 就称这个边集为割边集合, 如果某个割边集合只含有一条边 X (也即 {X} 是一个边集合), 那么 X 称为一个割边, 也叫做桥
- 步骤
  - 1. 如果要找欧拉回路, 可以从任意点开始, 如果要找欧拉路径, 需要从有着奇数度的两个及顶点中的一个开始, 如果有奇数度顶点的话
  - 2. 选择当前点相连的边, 确保删除该边, 不会将欧拉图分成两个不同的联通分量
  - 3. 将该边加入到路径中, 并将该边从欧拉图中删除, 如果当前的选择有一个桥与非桥的边时候, 优先选非桥的边, 不到万不得已, 不选桥
  - 4. 持续该过程直到路径收集完成
- 分析: 上面的步骤中, 选桥边与非桥边的时候, 如何判断当前的边是否是桥, 这个过程很关键, 大体的思路是:
  - 从当前节点 u 出发, 计数, 哪些顶点可以通过 u 可达, 直接可达和间接可达均可以, 记为 cnt1
  - 移除掉 u-v 这条边
  - 从当前节点 v 出发, 哪些顶点可以通过 v 可达, 直接可达和间接可达均可以, 记为 cnt2
  - 恢复 u-v 这条边
  - 返回 cnt1 与 cnt2 的大小, 如果 cnt2 要比 cnt1 小, 说明移除 u-v 这条边, 从 v 可达的顶点数量减少, 产生了额外的联通分量, 此时返回 false, 说明这条边是桥, 反之返回 true

```

public List<String> findItinerary(List<List<String>> tickets) { // 这个算法还比较陌生
    for (List<String> t : tickets)
        adj.computeIfAbsent(t.get(0), z -> new ArrayList<>()).add(t.get(1));
    for (List<String> values : adj.values()) Collections.sort(values);
    String u = "JFK";
    ans.add(u);
    fleuryProcess(u);
    return ans;
}
Map<String, List<String>> adj = new HashMap<>();
List<String> ans = new ArrayList<>();
private void fleuryProcess(String u) {
    if (!adj.containsKey(u)) return ;
    for (int i = 0; i < adj.get(u).size(); i++) {
        String v = adj.get(u).get(i);
        if (isValidNextEdge(u, v)) {
            ans.add(v);
            adj.get(u).remove(v);
            fleuryProcess(v);
        }
    }
}
private boolean isValidNextEdge(String u, String v) { // 判断是否是割边:
    if (adj.get(u).size() == 1) return true;
    // boolean[] visited = new boolean[adj.get(u).size()];
    Map<String, Boolean> vis = new HashMap<>(); // vis: visited
    int cnt1 = dfs(u, vis);
    adj.get(u).remove(v);
    vis.clear(); // vis = new HashMap<>();
    int cnt2 = dfs(v, vis);
    adj.get(u).add(0, v);
    return cnt1 <= cnt2; // 如果 cnt2 要比 cnt1 小, 说明移除 u-v 这条边, 从 v 可达的顶点数量减少, 产生了额外的联通分量, 此时返回 false, 说明这
}
private int dfs(String u, Map<String, Boolean> vis) {
    vis.put(u, true);
    int cnt = 1;
    if (adj.containsKey(u))
        for (String v : adj.get(u))
            if (vis.get(v) == null || (vis.get(v) != null && !vis.get(v)))
                cnt += dfs(v, vis);
    return cnt;
}

```

### 8.5.3 2097. Valid Arrangement of Pairs - Hard 欧拉回路

You are given a 0-indexed 2D integer array pairs where pairs[i] = [starti, endi]. An arrangement of pairs is valid if for every index i where  $1 \leq i < \text{pairs.length}$ , we have  $\text{endi}-1 == \text{starti}$ .

Return any valid arrangement of pairs.

Note: The inputs will be generated such that there exists a valid arrangement of pairs.

#### 1. 解题思路与分析

```

// One thing different is that we need to find the start point. it is obvious that if indegree is larger than 0, that is the start point.
public int[][] validateArrangement(int[][] pairs) {
    Map<Integer, Integer> ins = new HashMap<>();
    for (int [] p : pairs) {
        adj.computeIfAbsent(p[0], z -> new ArrayList<>()).add(p[1]);
        ins.put(p[0], ins.getOrDefault(p[0], 0) + 1);
        ins.put(p[1], ins.getOrDefault(p[1], 0) - 1);
    }
    int bgn = -1;
    for (Integer key : ins.keySet()) {
        if (ins.get(key) > 0) {
            bgn = key;
            break;
        }
    }
    if (bgn == -1) bgn = pairs[0][0]; // 如果没有, 就可以随便从某一个点开始 ?
    dfs(bgn);
    int n = pairs.length;
    int [][] ans = new int [n][];
    for (int i = n-1; i >= 0; i--) // 所以这里添加答案, 也需要反序回正
        ans[n-1-i] = ll.get(i);
    return ans;
}
Map<Integer, List<Integer>> adj = new HashMap<>();
List<int []> ll = new ArrayList<>();
void dfs(int node) {
    while (adj.get(node) != null && adj.get(node).size() > 0) {
        List<Integer> nextNodesCandi = adj.get(node);
        int next = nextNodesCandi.get(nextNodesCandi.size()-1); // 从后往前遍历, 方便从后往前删除已经遍历过的节点
        adj.get(node).remove(nextNodesCandi.size()-1);
        dfs(next);
        ll.add(new int [] {node, next}); // 这里的顺序是倒着加的, dfs 完接下来的答案、之后再加的
    }
}

```

## 2. 解题思路与分析: todo: 这个答案没有看懂

```
// In a word, this solution is to first determine whether the target is an Euler circuit or an Euler path, then solve it.
public int[][] validArrangement(int[][] pairs) {
    int n = pairs.length;
    Map<Integer, Integer> outdegree = new HashMap<>();
    Map<Integer, Deque<Integer>> out = new HashMap<>();
    for (int[] pair : pairs) {
        outdegree.put(pair[0], outdegree.getOrDefault(pair[0], 0) + 1);
        outdegree.put(pair[1], outdegree.getOrDefault(pair[1], 0) - 1);
    }
    int[][] ans = new int[n][2];
    for (int i = 0; i < n; i++)
        Arrays.fill(ans[i], -1);
    for (Map.Entry<Integer, Integer> en : map.entrySet()) { // 试图寻找起始和结束的位置
        if (en.getValue() == 1) ans[0][0] = en.getKey();
        if (en.getValue() == -1) ans[n-1][1] = en.getKey();
    }
    if (ans[0][0] == -1) { // 这里为什么就可以从第一个往后搜、从两边往中间搜呢？
        ans[0][0] = pairs[0][0];
        ans[n-1][1] = pairs[0][0];
    }
    for (int[] p : pairs) {
        out.computeIfAbsent(p[0], z -> new ArrayDeque<>()).offerLast(p[1]);
        // out.computeIfAbsent(p[0], k -> new ArrayDeque<>());
        out.computeIfAbsent(p[1], k -> new ArrayDeque<>()); // 需要加上
        // out.get(p[0]).offerLast(p[1]);
    }
    int i = 0, j = n-1;
    while (i < j) { // 没看明白这中间在是做什么 ???
        int from = ans[i][0];
        Deque<Integer> toList = out.get(from); // 这里是个栈，上面如果不加上，这里会是 null
        if (toList.size() == 0) {
            i--;
            ans[j][0] = ans[i][0];
            j--;
            ans[j][1] = ans[j+1][0];
        } else {
            ans[i+1][1] = toList.pollLast();
            // ans[i+1][1] = toList.removeLast();
            ans[i][0] = ans[i-1][1];
        }
    }
    return ans;
}
```

### 8.5.4 1591. Strange Printer II - Hard

There is a strange printer with the following two special requirements:

On each turn, the printer will print a solid rectangular pattern of a single color on the grid. This will cover up the existing colors in the rectangle. Once the printer has used a color for the above operation, the same color cannot be used again. You are given a  $m \times n$  matrix `targetGrid`, where `targetGrid[row][col]` is the color in the position  $(row, col)$  of the grid.

Return true if it is possible to print the matrix `targetGrid`, otherwise, return false.

#### 1. 解题思路与分析: 邻接有向图 + 拓扑排序

这道题可以认为是在研究: 是否有一种颜色序列, 按照这个序列进行染色, 最终矩阵就会呈现输入的状态。

矩形上的某一个像素点, 可能会先后经历多次染色。比如先染红, 再染绿, 再染黄, 最后染蓝, 最后呈现出的就是蓝色。我们知道这个像素现在是蓝色;

而它在红色/绿色/黄色矩形范围内, 说明这个像素曾经红过/绿过/黄过。

此时我们可以提炼出信息: 假定先染的优先于后染的, 那么红色优于蓝色, 绿色优于蓝色, 黄色优于蓝色。(红绿黄之间的顺序未定)。

题中指出, 颜色最多有 6060 种, 我们可以建立一个有向图, 图中的结点就是这 6060 个颜色 1~60160。

按照刚才的方法找出所有的有向边, 进行拓扑排序即可判断出结果。

```
public boolean isPrintable(int[][] a) {
    int m = a.length, n = a[0].length, max = Math.max(m, n);
    for (int i = 0; i < m; i++)
        max = Math.max(max, Arrays.stream(a[i]).max().getAsInt());
    int N = max + 1;
    int [] up = new int [N], down = new int [N], left = new int [N], right = new int [N];
    Arrays.fill(up, m);
    Arrays.fill(left, n);
    Arrays.fill(down, -1);
    Arrays.fill(right, -1);
    for (int i = 0; i < m; i++) // 界定每一种着色的上下左右边界, 以便接下来排序
```

```

for (int j = 0; j < n; j++) {
    int k = a[i][j];
    up[k] = Math.min(up[k], i);
    down[k] = Math.max(down[k], i);
    left[k] = Math.min(left[k], j);
    right[k] = Math.max(right[k], j);
}
// 根据每种着色的界定范围，建立拓扑排序：这后半部分还有点儿不熟练
// 当前位置颜色 cur 在某个矩阵 k 中但是不为矩阵 k 的颜色时，建立从 k 到 cur 的边，cur 可以存在于多个矩阵中
boolean [][] nei = new boolean [N][N]; // neighbours
List<Integer>[] adj = new ArrayList[N]; // 邻接有向图：按照染色的先后顺序
int [] ins = new int [N];
for (int i = 0; i < N; i++) adj[i] = new ArrayList<>();
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        int cur = a[i][j]; // 当前格的最终打印着色
        for (int k = 1; k < N; k++) { // 遍历所有的着色：暴搜当前着色 cur 是否会在某种着色 k 之后染色
            if (k == cur) continue;
            if (i >= up[k] && i <= down[k] && j >= left[k] && j <= right[k]) // 现着色 cur 完全处于先前染色 k 的内部，所以 cur 是后着色
                // if (!nei[k][cur]) { // BUG：是有向图：这里顺序很重要，先染色 是否与后染色相连/相前后
                if (!nei[k][cur]) { // k 先染后， cur 后染色
                    adj[k].add(cur);
                    ins[cur]++;
                    nei[k][cur] = true;
                }
            }
        }
    }
}
List<Integer> l = new ArrayList<>();
while (true) { // 寻找入度为 0 的颜色点，减小该点连结的点的入度，直到所有点的入度都为 0
    int i;
    for (i = 1; i < N; i++)
        if (ins[i] == 0) {
            l.add(i);
            for (int v : adj[i]) ins[v]--;
            ins[i] = -1;
            break;
        }
    if (i == N) break;
}
return l.size() == max; // 按照拓扑排序，这所有的染色都可以有序地染出来，那么合法
}

```

## 2. 解题思路与分析: topological sort

```

public boolean isPrintable(int[][] a) {
    int m = a.length, n = a[0].length;
    Set<Integer> col = new HashSet<>();
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            col.add(a[i][j]);
    for (Integer c : col) {
        int fi = -1, fj = Integer.MAX_VALUE, li = -1, lj = -1; // f: first, f row, f col, l: last, l row, l col
        for (int i = 0; i < m; i++)
            for (int j = 0; j < n; j++)
                if (a[i][j] == c) {
                    if (fi == -1) fi = i; // 只记最早出现的第一次
                    fj = Math.min(fj, j);
                    li = i;
                    lj = Math.max(lj, j);
                }
        for (int i = fi; i <= li; i++)
            for (int j = fj; j <= lj; j++)
                if (a[i][j] != c) // a[i][j] 是会在当前染色 c 之后染色的
                    adj.computeIfAbsent(c, z -> new HashSet<>()).add(a[i][j]);
    }
    Set<Integer> vis = new HashSet<>(); // visiting: 只保证先染的着色不会在后染的着色里再次出现
    for (Integer c : col)
        if (!topologicalSort(vis, c)) return false;
    return true;
}
Map<Integer, Set<Integer>> adj = new HashMap<>(); // 在 key 之后染色的着色集合
private boolean topologicalSort(Set<Integer> vis, int c) { // 这种写法好陌生
    if (vis.contains(c)) return false;
    vis.add(c);
    for (Integer nei : adj.getOrDefault(c, Collections.emptySet()))
        if (!topologicalSort(vis, nei)) return false;
    vis.remove(c);
    return true;
}

```

## 8.6 双端队列 BFS

### 8.6.1 1368. Minimum Cost to Make at Least One Valid Path in a Grid - Hard

Given a  $m \times n$  grid. Each cell of the grid has a sign pointing to the next cell you should visit if you are currently in this cell. The sign of  $grid[i][j]$  can be: 1 which means go to the cell to the right. (i.e go from  $grid[i][j]$  to  $grid[i][j + 1]$ ) 2 which means go to the cell to the left. (i.e go from  $grid[i][j]$  to  $grid[i][j - 1]$ ) 3 which means go to the lower cell. (i.e go from  $grid[i][j]$  to  $grid[i + 1][j]$ ) 4 which means go to the upper cell. (i.e go from  $grid[i][j]$  to  $grid[i - 1][j]$ ) Notice that there could be some invalid signs on the cells of the grid which points outside the grid.

You will initially start at the upper left cell  $(0,0)$ . A valid path in the grid is a path which starts from the upper left cell  $(0,0)$  and ends at the bottom-right cell  $(m - 1, n - 1)$  following the signs on the grid. The valid path doesn't have to be the shortest.

You can modify the sign on a cell with cost = 1. You can modify the sign on a cell one time only.

Return the minimum cost to make the grid have at least one valid path.

#### 1. 解题思路与分析: 0-1 广度优先搜索 (最优解法)

这道题其实是个经典的双端队列 BFS。将每个格子看成是图的顶点，相邻格子是有边相连接的。如果从顶点  $(x, y)$  到  $(u, v)$  的实际方向和矩阵在  $(x, y)$  所表示的方向相同，则令这条边的边权为 0，否则令其边权为 1。原题相当于在问，在此图中，从起点到终点的最短路长度是多少。由于边权只有 0 和 1 两种，所以可以用双端队列 BFS 来做。每次拓展的时候，如果是沿着边权 0 的边走的，则插入队头，否则插入队尾。从队列里取元素的时候永远都从队头取。然后用堆优化的 Dijkstra 算法模板来写即可。时空复杂度  $O(mn)$ 。

0-1 广度优先搜索的实现其实与 Dijkstra 算法非常相似。在 Dijkstra 算法中，我们用优先队列保证了距离的单调递增性。而在 0-1 广度优先搜索中，实际上任意时刻队列中的节点与源点的距离均为  $dd$  或  $d + 1d+1$  (其中  $dd$  为某一非负整数)，并且所有与源点距离为  $dd$  的节点都出现在队首附近，所有与源点距离为  $d + 1d+1$  的节点都出现在队尾附近。因此，我们只要使用双端队列，对于边权为 00 和 11 的两种情况分别将对应节点添加至队首和队尾，就保证了距离的单调递增性。

```
public int minCost(int[][] g) {
    int m = g.length, n = g[0].length;
    int [][] d = new int [m][n]; // dist to [0, 0]
    for (int i = 0; i < m; i++)
        Arrays.fill(d[i], Integer.MAX_VALUE);
    d[0][0] = 0;
    int [][] dirs = {{0, 0}, {0, 1}, {0, -1}, {1, 0}, {-1, 0}}; // [0, 1, 2, 3, 4]
    boolean [][] vis = new boolean [m][n];
    ArrayDeque<Integer> q = new ArrayDeque<>();
    q.offerFirst(0);
    while (!q.isEmpty()) {
        int idx = q.pollFirst();
        int i = idx / n, j = idx % n;
        if (vis[i][j]) continue;
        if (i == m-1 && j == n-1) return d[i][j];
        vis[i][j] = true;
        for (int k = 1; k < 5; k++) {
            int x = i + dirs[k][0], y = j + dirs[k][1];
            if (x < 0 || x >= m || y < 0 || y >= n) continue;
            int cost = k == g[i][j] ? 0 : 1;
            if (!vis[x][y] && d[x][y] > d[i][j] + cost) {
                d[x][y] = d[i][j] + cost;
                if (cost == 0) q.offerFirst(x * n + y);
                else q.offerLast(x * n + y);
            }
        }
    }
    return -1;
}
```

#### 2. 解题思路与分析: 最短路径问题总结 BFS

##### (a) 题目分析

- 虽然题目的描述中写了有效路径不需要是最短路径，但其实这道题目还是一个最短路径问题，只不过要求的最短距离并不是在网格中行走的距离，而是改变方向的次数。
- 所谓最短路径问题，就是对于图  $G(V,E)G(V,E)$ ，寻找从  $u \in V \cup V$  到  $v \in V \cup V$  的最短距离。最短路径的算法有很多，包括 Dijkstra, Floyd, Bellman-Ford, SPFA 等。

## Dijkstra

Dijkstra 算法是求取单源最短路径的常用算法，其基本思想是每次用当前未拓展且具有最小权值的点来更新源点到其余顶点的距离。Dijkstra 算法的时间复杂度包含两个部分：找到最小节点并将其移除的用时  $T_{extract\_min}$ ，以及更新某一节点权值的用时  $T_{decrease\_key}$ 。因为每个节点最多充当一次最小节点，而每条边最多参与一次更新，算法整体的时间复杂度可以表示为

$$\Theta(V) \cdot T_{extract\_min} + \Theta(E) \cdot T_{decrease\_key}$$

数据结构	$T_{extract\_min}$	$T_{decrease\_key}$	总时间复杂度
数组	$O(V)$	$O(1)$	$O(V^2 + E)$
优先队列（小根堆）	$O(\log V)$	$O(\log V)$	$O((V + E) \log V)$
Fibonacci 堆	均摊 $O(\log V)$	均摊 $O(1)$	均摊 $O(V \log V + E)$

由于 Fibonacci 堆实现较为复杂，各语言标准库未提供实现（C++ 的 Boost 库实现了这一数据结构），并且其实际运行效率与优先队列相比的优势并不明显，所以基于优先队列的实现是 Dijkstra 算法最常见的实现方式。

需要注意的是，Dijkstra 算法在图中存在负环的情况下不适用！对于无向图来说，只要有一条负边，就构成了一个两节点的负环，所以在无向图中只要有负边就不能使用 Dijkstra 算法。

对于 BFS，相信大家一定都很熟悉了。与 DFS 相比，BFS 的特点是按层遍历，从而可以保证首先找到最优解（最少步数、最小深度）。从这个意义上讲，BFS 解决的其实也是最短路径问题。这一问题对应的图 GG 包含的所有顶点即为状态空间，而每一个可能的状态转移都代表了一条边。

比如，在经典的迷宫问题中，每一个状态  $(x,y)$  代表了一个顶点，而一个无障碍格子与其相邻的无障碍格子之间则存在一条无向边。

那么，这个图 GG 和一般的图相比，有什么特点呢？

关键就在于边的权值。在 BFS 问题中，所有边的权值均为 1！因为我们每一次从一个状态转移到一个新的状态，就多走了一步。正因为边权值均为 1，我们用一个队列记录所有状态，前面的状态对应的总权值一定小于后面的状态，所以我们就可以在  $O(1)O(1)$  的时间内实现找到最小节点并将其移除的操作（只要取队头，然后出队就可以了），从而寻找最短路径的时间复杂度就减小到了  $O(V+E)O(V+E)$ 。

但普通的 BFS 算法，在本题中并不适用，因为存在权值为 0 的边！如果从一个格子到另一个格子，不需要修改格子上的标记，那么这一步移动的权值就为 0。如果我们还沿用普通 BFS 的做法，就无法保证队头元素一定是当前具有最小权值的节点。

怎么办呢？简单粗暴的做法是：允许多次扩展同一个点。只要当前边能够更新节点的权值，就将节点再次入队。

```
public int minCost(int[][] g) {
    int [][] dirs = {{0, 0}, {0, 1}, {0, -1}, {1, 0}, {-1, 0}};
    int m = g.length, n = g[0].length;
    int [][] d = new int [m][n]; // dist to [0, 0]
    for (int i = 0; i < m; i++)
        Arrays.fill(d[i], Integer.MAX_VALUE);
    d[0][0] = 0;
    Queue<int []> q = new LinkedList<>();
    q.offer(new int [] {0, 0});
    while (!q.isEmpty()) {
        int [] cur = q.poll();
        int i = cur[0], j = cur[1];
        for (int k = 1; k < 5; k++) {
            int x = i + dirs[k][0], y = j + dirs[k][1];
            if (x < 0 || x >= m || y < 0 || y >= n) continue;
            int newDist = d[i][j] + (k == g[i][j] ? 0 : 1);
            if (newDist < d[x][y]) {
                d[x][y] = newDist;
                q.offer(new int [] {x, y});
            }
        }
    }
    return d[m-1][n-1];
}
```

- 解题思路与分析：SPFA 如果一个节点已经在队列中，其实就没有必要将其再次入队了。这是 SPFA 算法的基本思想。可以看到，与上面的 BFS 方法相比，就是增加了一个 in 数组来判断当前节点是否已经在队列中。

SPFA 算法是一个十分依赖于数据的算法。在特定的数据下，SPFA 会退化为 Bellman-Ford，时间复杂度为  $O(V \cdot E)$ 。一般的编程竞赛中，涉及到最短路径的题目，都会有专门卡 SPFA 的数据，所以一般情况下还是使用 Dijkstra 算法。本题的测试数据相对较弱，BFS 和 SPFA 都可以顺利通过，甚至 SPFA 的运行时间还要长于 BFS（修改 in 数组状态带来了额外的开销）。

SPFA 的好处是可以判断负环。我们可以用一个数组记录每个顶点的入队次数，如果有顶点的入队次数超过了  $V$  次，则代表图中存在负环。

```
public int minCost(int[][] g) {
    int [][] dirs = {{0, 0}, {0, 1}, {0, -1}, {1, 0}, {-1, 0}};
    int m = g.length, n = g[0].length;
    int [][] d = new int [m][n];
    for (int i = 0; i < m; i++)
        Arrays.fill(d[i], Integer.MAX_VALUE);
    d[0][0] = 0;
    boolean [][] in = new boolean [m][n];
    Queue<int []> q = new LinkedList<>();
    q.offer(new int [] {0, 0});
    in[0][0] = true;
    while (!q.isEmpty()) {
        int [] cur = q.poll();
        int i = cur[0], j = cur[1];
        in[i][j] = false;
        for (int k = 1; k < 5; k++) {
            int x = i + dirs[k][0], y = j + dirs[k][1];
            if (x < 0 || x >= m || y < 0 || y >= n) continue;
            int newDist = d[i][j] + (k == g[i][j] ? 0 : 1);
            if (newDist < d[x][y]) {
                d[x][y] = newDist;
                if (!in[x][y]) {
                    q.offer(new int [] {x, y});
                    in[x][y] = true;
                }
            }
        }
    }
    return d[m-1][n-1];
}
```

#### 4. 解题思路与分析

### 8.6.2 126. Word Ladder II - Hard BFS

A transformation sequence from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:

Every adjacent pair of words differs by a single letter. Every si for  $1 \leq i \leq k$  is in wordList. Note that beginWord does not need to be in wordList. sk == endWord Given two words, beginWord and endWord, and a dictionary wordList, return all the shortest transformation sequences from beginWord to endWord, or an empty list if no such sequence exists. Each sequence should be returned as a list of the words [beginWord, s1, s2, ..., sk].

#### 1. 解题思路与分析: 广度优先搜索

- 官方题解: <https://leetcode-cn.com/problems/word-ladder-ii/solution/dan-ci-jie-long-ii-by-leetcode-solution/>

```
public List<List<String>> findLadders(String bgn, String end, List<String> list) {
    Set<String> ss = new HashSet<>(list);
    if (!ss.contains(end)) return ans;
    ss.remove(bgn);
    // BFS: 第 1 步: 广度优先遍历建图
    Map<String, Integer> cnt = new HashMap<>(); // 记录扩展出的单词是在第几次扩展的时候得到的, key: 单词, value: 在广度优先遍历的第几层
    cnt.put(bgn, 0);
    Map<String, List<String>> from = new HashMap<>(); // 记录了单词是从哪些单词扩展而来, key: 单词, value: 单词列表, 这些单词可以变换到 key, 它们
    int step = 1, n = bgn.length();
    boolean found = false;
    Queue<String> q = new LinkedList<>();
    q.offer(bgn);
    while (!q.isEmpty()) {
        for (int size = q.size() - 1; size >= 0; size--) {
            String cur = q.poll();
            char [] s = cur.toCharArray();
            for (int i = 0; i < n; i++) {
                char ori = s[i];
                for (char c = 'a'; c <= 'z'; c++) {
                    if (s[i] == c) continue; //
                    s[i] = c;
                    String next = String.valueOf(s);
                    if (cnt.containsKey(next) && step == cnt.get(next)) //
                        from.get(next).add(cur); //
                    if (!ss.contains(next)) continue; // BUG: 还没有想明白, 为什么我把这行写前面会少掉答案呢?
                    ss.remove(next); // 如果从一个单词扩展出来的单词以前遍历过, 距离一定更远, 为了避免搜索到已经遍历到, 且距离更远的单词, 需要将它从
                    q.offer(next); // 这一层扩展出的单词进入队列
                }
            }
        }
    }
}
```

```

        from.computeIfAbsent(next, z -> new ArrayList<>()).add(cur); // 记录 next Word 从 cur Word 而来
        cnt.put(next, step);
        if (next.equals(end)) found = true;
    }
    s[i] = ori;
}
step++;
if (found) break;
}
// 第 2 步: 深度优先遍历找到所有解, 从 end 恢复到 bgn , 所以每次尝试操作 path 列表的头部
if (found) {
    Deque<String> path = new ArrayDeque<>();
    path.add(end);
    dfs(from, path, bgn, end);
}
return ans;
}
List<List<String>> ans = new ArrayList<>();
void dfs(Map<String, List<String>> from, Deque<String> path, String end, String cur) {
    if (cur.equals(end)) {
        ans.add(new ArrayList<>(path)); // 这个写法学习一下, 第一次见
        return ;
    }
    for (String precursor : from.get(cur)) {
        path.offerFirst(precursor);
        dfs(from, path, end, precursor);
        path.pollFirst();
    }
}
}

```

2. 解题思路与分析: 详细通俗的思路分析, 多解法: DFS + BFS 双向搜索 (two-end BFS) 双向 BFS 搜索

- <https://leetcode-cn.com/problems/word-ladder-ii/solution/xiang-xi-tong-su-de-si-lu-fen-xi-duo-jie/>

这个题解和自己最初解法比较接近, 需要再好好学习一下

```

public List<List<String>> findLadders(String beginWord, String endWord, List<String> wordList) {
    List<List<String>> ans = new ArrayList<>();
    if (!wordList.contains(endWord)) return ans;
    // 利用 BFS 得到所有的邻居节点
    HashMap<String, ArrayList<String>> map = new HashMap<>();
    bfs(beginWord, endWord, wordList, map);
    ArrayList<String> temp = new ArrayList<String>();
    // temp 用来保存当前的路径
    temp.add(beginWord);
    findLaddersHelper(beginWord, endWord, map, temp, ans);
    return ans;
}
private void findLaddersHelper(String beginWord, String endWord, HashMap<String, ArrayList<String>> map,
                               ArrayList<String> temp, List<List<String>> ans) {
    if (beginWord.equals(endWord)) {
        ans.add(new ArrayList<String>(temp));
        return;
    }
    // 得到所有的下一个的节点
    ArrayList<String> neighbors = map.getOrDefault(beginWord, new ArrayList<String>());
    for (String neighbor : neighbors) {
        temp.add(neighbor);
        findLaddersHelper(neighbor, endWord, map, temp, ans);
        temp.remove(temp.size() - 1);
    }
}
// 利用递归实现了双向搜索
private void bfs(String beginWord, String endWord, List<String> wordList, HashMap<String, ArrayList<String>> map) {
    Set<String> set1 = new HashSet<String>();
    set1.add(beginWord);
    Set<String> set2 = new HashSet<String>();
    set2.add(endWord);
    Set<String> wordSet = new HashSet<String>(wordList);
    bfsHelper(set1, set2, wordSet, true, map);
}
// direction 为 true 代表向下扩展, false 代表向上扩展
private boolean bfsHelper(Set<String> set1, Set<String> set2, Set<String> wordSet, boolean direction,
                         HashMap<String, ArrayList<String>> map) {
    // set1 为空了, 就直接结束
    // 比如下边的例子就会造成 set1 为空
    /*^I"hot"
     "dog"
     ["hot", "dog"]*/
    if (set1.isEmpty()) return false;
    // set1 的数量多, 就反向扩展
    if (set1.size() > set2.size())
        return bfsHelper(set2, set1, wordSet, !direction, map);
    // 将已经访问过单词删除
    wordSet.removeAll(set1);
    wordSet.removeAll(set2);
    boolean done = false;
    for (String word : set1) {
        for (String neighbor : map.getOrDefault(word, new ArrayList<String>())) {
            if (wordSet.contains(neighbor)) {
                wordSet.remove(neighbor);
                if (neighbor.equals(endWord)) done = true;
                else set2.add(neighbor);
            }
        }
    }
    if (done) return true;
    return bfsHelper(set1, set2, wordSet, direction, map);
}

```

```

// 保存新扩展得到的节点
Set<String> set = new HashSet<String>();
for (String str : set1) {
    // 遍历每一位
    for (int i = 0; i < str.length(); i++) {
        char[] chars = str.toCharArray();
        // 尝试所有字母
        for (char ch = 'a'; ch <= 'z'; ch++) {
            if(chars[i] == ch) continue;
            chars[i] = ch;
            String word = new String(chars);
            // 根据方向得到 map 的 key 和 val
            String key = direction ? str : word;
            String val = direction ? word : str;
            ArrayList<String> list = map.containsKey(key) ? map.get(key) : new ArrayList<String>();
            // 如果相遇了就保存结果
            if (set2.contains(word)) {
                done = true;
                list.add(val);
                map.put(key, list);
            }
            // 如果还没有相遇，并且新的单词在 word 中，那么就加到 set 中
            if (!done && wordSet.contains(word)) {
                set.add(word);
                list.add(val);
                map.put(key, list);
            }
        }
    }
}
// 一般情况下新扩展的元素会多一些，所以我们下次反方向扩展 set2
return done || bfsHelper(set2, set, wordSet, !direction, map);
}

```

## 8.7 环与入度等

### 8.7.1 685. Redundant Connection II - Hard

In this problem, a rooted tree is a directed graph such that, there is exactly one node (the root) for which all other nodes are descendants of this node, plus every node has exactly one parent, except for the root node which has no parents.

The given input is a directed graph that started as a rooted tree with  $n$  nodes (with distinct values from 1 to  $n$ ), with one additional directed edge added. The added edge has two different vertices chosen from 1 to  $n$ , and was not an edge that already existed.

The resulting graph is given as a 2D-array of edges. Each element of edges is a pair  $[ui, vi]$  that represents a directed edge connecting nodes  $ui$  and  $vi$ , where  $ui$  is a parent of child  $vi$ .

Return an edge that can be removed so that the resulting graph is a rooted tree of  $n$  nodes. If there are multiple answers, return the answer that occurs last in the given 2D-array.

#### 1. 解题思路与分析

```

// 对于有向图中，如果存在  $\text{indegree}(v) == 2$  的点，那么要删除的一定是这个点的某条有向边 ( $u \rightarrow v$ )
// 因为题目最后保证是一个 rooted tree (every node has exactly one parent)，具体看第一段定义。
// 如果不存在  $\text{indegree}(v) == 2$  的点，那么直接删去最后一个造成环存在的有向边即可。
// 现在存在三种情况：
// (1) 有向图中只有环。这种情况就简单将两个节点具有共同根节点的边删去就好。
// (2) 有向图中没有环，但有个节点有两个父节点。这种情况就将第二次出现不同父节点的边删去就好。
// (3) 有向图中既有环，而且有个节点还有两个父节点。这时就检测当除去第二次出现父节点的边后，剩余边是不是合法的，如果不合法证明应该删掉的是另一个父节点的边。
public int[] findRedundantDirectedConnection(int[][] egs) {
    Set<Integer> p = new HashSet<>();
    Map<Integer, Integer> par = new HashMap<>(); // k, v: v <- u
    List<int[]> candi = new ArrayList<>();
    for (int[] e : egs) {
        int u = e[0], v = e[1];
        p.add(u);
        p.add(v);
        if (!par.containsKey(v)) { // 对于入度为 1 的右端点边，记在字典里
            par.put(v, u);
            continue;
        }
        candi.add(new int[] {par.get(v), v}); // 第一次出现的边
        candi.add(new int[] {u, v}); // 第二次出现的边，答案是这两者之一
        e[1] = -1; // 先将后出现的第二条边废掉，验证答案
    }
    UnionFind uf = new UnionFind(p.size()); // 总顶点的个数
    for (int[] e : egs) {
        if (e[1] == -1) continue; // 跳过正在验证是否为的答案后出现的第二条边
        int u = e[0]-1, v = e[1]-1; // uf 0-based
        if (!uf.union(u, v)) { // 在已经废除后出现的第二条边之后，还是出现了环，那么后出现的第二条边不是答案，第一条才是
            if (candi.isEmpty()) return e; // 不存在入度为 2 的点，直接删去最后一条造成环存在的有向边，即当前边即可
            return candi.get(0);
        }
    }
}

```

```

    }
    return candi.get(1); // 删除第二条边后，所有的存在都合法，那么就返回第二条边
}
class UnionFind {
    int[] parent;
    int[] rank;
    int size;
    public UnionFind(int size) {
        this.size = size;
        parent = new int[size];
        rank = new int[size];
        for (int i = 0; i < size; i++) parent[i] = i;
    }
    public int find(int x) {
        if (parent[x] != x)
            parent[x] = find(parent[x]);
        return parent[x];
    }
    public boolean union(int x, int y) {
        int xp = find(x);
        int yp = find(y);
        if (xp == yp) return false; // 已经在同一个强连通分量中
        if (rank[xp] > rank[yp])
            parent[yp] = xp;
        else if (rank[xp] < rank[yp])
            parent[xp] = yp;
        else {
            parent[yp] = xp;
            rank[xp]++;
        }
        return true;
    }
}

```

## 8.8 普通 BFS, 但 state 状态很重要

### 8.8.1 1293. Shortest Path in a Grid with Obstacles Elimination - Hard

You are given an  $m \times n$  integer matrix grid where each cell is either 0 (empty) or 1 (obstacle). You can move up, down, left, or right from and to an empty cell in one step.

Return the minimum number of steps to walk from the upper left corner  $(0, 0)$  to the lower right corner  $(m - 1, n - 1)$  given that you can eliminate at most  $k$  obstacles. If it is not possible to find such walk return -1.

#### 1. 解题思路与分析

- 在普通的 bfs 中，为了避免走重复的路径，我们一般采用 visited 数组来记录下走过的格子，当路径遇到一个已走过的格子时，便不能再通过该点。但本题略有不同，上面说过，本题存在两种路径，因此 visited 数组中的状态应该包含 3 种状态：
  - 0：该位置没有走过
  - 1：该位置被普通路径走过
  - 2：该路径被穿越路径走过
- 对于以上三种状态，
  - 当前格子为非障碍物时，普通路径可以走访问状态为 0 的格子，走过后，访问状态更新为 1
  - 当前格子为非障碍物时，穿越路径可以走访问状态为 0 的格子，走过后，访问状态更新为 2
  - 当前格子为非障碍物时，普通路径可以走访问状态为 2 的格子，走过后，访问状态更新为 1
  - 当前格子为障碍物时，如果还保有穿越次数，并且该格子的访问状态为 0 时，所有路径可以通过此处，通过后，该路径变为穿越路径，并且当前格子访问状态更新为 1（更新为 2 也无所谓）

```

public int shortestPath(int[][] a, int k) {
    int [][] dirs = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
    int m = a.length, n = a[0].length;
    int [][] vis = new int [m][n];
    Deque<int []> q = new ArrayDeque<()>();
    q.offerLast(new int [] {0, 0, k});
    int cnt = 0;
    while (!q.isEmpty()) {
        for (int z = q.size()-1; z >= 0; z--) {
            int [] cur = q.pollFirst();
            if (cur[0] == m-1 && cur[1] == n-1) return cnt;
            for (int [] d : dirs) {
                int i = cur[0] + d[0], j = cur[1] + d[1];
                if (i >= 0 && i < m && j >= 0 && j < n) {
                    // 1. 当前格子为非障碍物时，普通路径可以走访问状态为 0 的格子，走过后，访问状态更新为 1
                    // 2. 当前格子为非障碍物时，普通路径可以走访问状态为 2 的格子，走过后，访问状态更新为 1
                    if (a[i][j] == 0 && (vis[i][j] == 0 || vis[i][j] == 2 && cur[2] == k)) { // 当前格为非障碍物
                        if (vis[i][j] == 0) // 普通路径：可以走状态为 0 的格子，走过后变为 1 or 2

```

```

        // 2. 当前格子为非障碍物时, 穿越路径可以走访问状态为 0 的格子, 走过后, 访问状态更新为 2
        vis[i][j] = (cur[2] == k ? 1 : 2);
    } else vis[i][j] = 1;
    q.offerLast(new int[] {i, j, cur[2]});
    // 4. 当前格子为障碍物时, 如果还保有穿越次数, 并且该格子的访问状态为 0 时, 所有路径可以通过此处,
    //     通过后, 该路径变为穿越路径, 并且当前格子访问状态更新为 1 (更新为 2 也无所谓)
} else if (a[i][j] == 1 && vis[i][j] == 0 && cur[2] > 0) {
    vis[i][j] = 2;
    q.offerLast(new int[] {i, j, cur[2]-1});
}
}
}
cnt++;
}
return -1;
}

```

## 2. 解题思路与分析

- 思路是 BFS。将已经移除了多少个障碍物加进状态，然后做隐式图搜索即可。代码如下：

```

public int shortestPath(int[][] a, int k) { // 这个思路极为简洁, 但速度跟上一个方法比, 会慢很多
    if (a.length <= 1 && a[0].length <= 1) return 0;
    int [][] dirs = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
    int m = a.length, n = a[0].length;
    boolean [[[k+1]] vis = new boolean [m][n][k+1];
    Deque<int []> q = new ArrayDeque<>()();
    q.offerLast(new int [] {0, 0, 0}); // 前两个数代表坐标, 最后一个数代表已经移除了多少个障碍物
    vis[0][0][0] = true;
    int cnt = 0;
    while (!q.isEmpty()) {
        for (int z = q.size()-1; z >= 0; z--) {
            int [] cur = q.pollFirst();
            for (int [] d : dirs) {
                int i = cur[0] + d[0], j = cur[1] + d[1], x = -1;
                if (i < 0 || i >= m || j < 0 || j >= n) continue;
                x = cur[2] + a[i][j];
                if (k >= x && !vis[i][j][x]) {
                    if (i == a.length-1 && j == a[0].length-1) return cnt+1;
                    q.offerLast(new int [] {i, j, x});
                    vis[i][j][x] = true;
                }
            }
        }
        cnt++;
    }
    return -1;
}

```

## 8.9 最小生成树

### 8.9.1 1489. Find Critical and Pseudo-Critical Edges in Minimum Spanning Tree - Hard

Given a weighted undirected connected graph with  $n$  vertices numbered from 0 to  $n - 1$ , and an array edges where  $\text{edges}[i] = [a_i, b_i, \text{weight}_i]$  represents a bidirectional and weighted edge between nodes  $a_i$  and  $b_i$ . A minimum spanning tree (MST) is a subset of the graph's edges that connects all vertices without cycles and with the minimum possible total edge weight.

Find all the critical and pseudo-critical edges in the given graph's minimum spanning tree (MST). An MST edge whose deletion from the graph would cause the MST weight to increase is called a critical edge. On the other hand, a pseudo-critical edge is that which can appear in some MSTs but not all.

Note that you can return the indices of the edges in any order.

## 1. 解题思路与分析

```

public List<List<Integer>> findCriticalAndPseudoCriticalEdges(int n, int[][] edges) {
    List<int []> lia = new ArrayList<>()();
    for (int [] e : edges) lia.add(e);
    Collections.sort(lia, (x, y) -> x[2] - y[2]);
    List<Integer> critical = new ArrayList<>()();
    List<Integer> pseudo = new ArrayList<>()();
    int minCost = getCostOfMST(n, lia, null, null);
    for (int i = 0; i < edges.length; i++) {
        int [] e = edges[i];
        if (getCostOfMST(n, lia, null, e) > minCost) // 优先条件: 如果已经是 critical, 就不用考虑后面的了
            critical.add(i);
        else if (getCostOfMST(n, lia, e, null) == minCost)
            pseudo.add(i);
    }
    return Arrays.asList(critical, pseudo);
}

```

```

int getCostOfMST(int n, List<int []> adj, int [] req, int [] avd) { // avoid
    int ans = 0;
    DSU dsu = new DSU(n);
    if (req != null) {
        dsu.union(req[0], req[1]);
        ans += req[2];
    }
    for (int [] e : adj) {
        if (e != avd && dsu.union(e[0], e[1])) // 可以合并
            ans += e[2];
        if (dsu.getCnt() == 1)
            return ans;
    }
    return Integer.MAX_VALUE; // 无法生成合法的最小生成树
}
static class DSU {
    int [] par;
    int [] rank; // 关于 rank 和 size 的部分
    int size;
    public boolean union(int x, int y) {
        int px = find(x), py = find(y);
        if (px == py) return false;
        // if (rank[x] >= rank[y]) { // BUG
        if (rank[px] >= rank[py]) {
            par[py] = px;
            if (rank[px] == rank[py])
                ++rank[px];
            // rank[x] += rank[y]; // BUG: 这里自己写错了
        } else {
            par[px] = py;
            // rank[y] += rank[x];
        }
        --size;
        return true;
    }
    public int getCnt() {
        return size;
    }
    int find(int x) {
        // if (par[x] != x) par[x] = find(par[x]); // BUG
        while (par[x] != x) {
            par[x] = par[par[x]];
            x = par[x];
        }
        return par[x];
    }
    public DSU(int n) {
        size = n;
        par = new int [n];
        rank = new int [n];
        for (int i = 0; i < n; i++)
            par[i] = i;
    }
}

```

## 2. 解题思路与分析

- 这是自己慢慢掰出来的，掰还第二遍的时候掰错了。。。最好是能够参考一下别人的最优解法

```

HashMap<Integer, int []> m = new HashMap<>(); // 这是参考以前写的
HashSet<Integer> s = new HashSet<>();
List<List<Integer>> ll = new ArrayList<>();
List<Integer> l = new ArrayList<>();
int minw = 0;
UnionFind uf;
public List<List<Integer>> findCriticalAndPseudoCriticalEdges(int n, int[][] edges) {
    HashMap<Integer, int []> map = new HashMap<>();
    for (int i = 0; i < edges.length; i++)
        map.put(i, edges[i]);
    Comparator<Map.Entry<Integer, int []>> c = new Comparator<Map.Entry<Integer, int []>>() {
        @Override
        public int compare(Map.Entry<Integer, int []> a, Map.Entry<Integer, int []> b) {
            int cmp1 = (a.getValue())[2] - (b.getValue())[2];
            if (cmp1 != 0)
                return cmp1;
            else return a.getKey().compareTo(b.getKey());
        }
    };
    m = map.entrySet().stream().sorted(c)
        .collect(toMap(Map.Entry::getKey, Map.Entry::getValue, (e1, e2) -> e1, LinkedHashMap::new));
    uf = new UnionFind(n);
    kruskal(n, s, -1, true);
    int minG = minw;
    l = new ArrayList<>();
    uf = new UnionFind(n);
    for (Integer val : s) {
        minw = 0;
        uf.reset();

```

```

kruskal(n, new HashSet<>(), val, true);
    if (minw > minG || uf.getCnt() > 1) l.add(val);
}
ll.add(l);
l = new ArrayList<>();
for (Integer i : m.keySet()) {
    if (ll.get(0).contains(i)) continue;
    minw = 0;
    uf.reset();
    kruskal(n, new HashSet<>(), i, false);
    if (minw == minG) l.add(i);
}
ll.add(l);
return ll;
}
private void kruskal(int n, HashSet<Integer> s, int i, boolean vis) {
    if (!vis) {
        uf.merge((m.get(i))[0], (m.get(i))[1]);
        s.add(i);
        minw += (m.get(i))[2];
    }
    for (Map.Entry<Integer, int []> entry : m.entrySet()) {
        int [] cur = entry.getValue();
        if (uf.sameGroup(cur[0], cur[1]) || (entry.getKey() == i && vis)) continue;
        if (s.size() < n-1) {
            uf.merge(cur[0], cur[1]);
            s.add(entry.getKey());
            minw += cur[2];
        }
    }
}
public class UnionFind {
    int [] pare;
    // int [] rank;
    int cnt;
    int n;
    public UnionFind(int x) {
        this.n = x;
        cnt = n;
        pare = new int[n];
        // rank = new int[n];
        for (int i = 0; i < n; i++)
            pare[i] = i;
    }
    public void reset() {
        for (int i = 0; i < n; i++)
            pare[i] = i;
        cnt = n;
        // rank = new int [n];
    }
    public int find(int val) {
        while (val != pare[val]) {
            pare[val] = pare[pare[val]];
            val = pare[val];
        }
        return val;
    }
    public void merge(int p, int q) {
        int rp = find(p);
        int rq = find(q);
        // if (rank[rp] < rank[rq])
        //     swap(rp, rq);
        pare[rq] = rp;
        // rank[rp] += rank[rq];
        --cnt; //
    }
    public boolean sameGroup(int x, int y) {
        return find(x) == find(y);
    }
    public int getCnt() {
        return cnt;
    }
    // private void swap(int x, int y) {
    //     int tmp = x;
    //     x = y;
    //     y = tmp;
    // }
}

```

## 8.10 跳表 Skiplist

### 8.10.1 1206. Design Skiplist - Hard

Design a Skiplist without using any built-in libraries.

A skiplist is a data structure that takes  $O(\log(n))$  time to add, erase and search. Comparing with treap and red-black

tree which has the same function and performance, the code length of Skiplist can be comparatively short and the idea behind Skiplists is just simple linked lists.

For example, we have a Skiplist containing [30,40,50,60,70,90] and we want to add 80 and 45 into it. The Skiplist works this way:

Artyom Kalinin [CC BY-SA 3.0], via Wikimedia Commons

You can see there are many layers in the Skiplist. Each layer is a sorted linked list. With the help of the top layers, add, erase and search can be faster than O(n). It can be proven that the average time complexity for each operation is O(log(n)) and space complexity is O(n).

See more about Skiplist: [https://en.wikipedia.org/wiki/Skip\\_list](https://en.wikipedia.org/wiki/Skip_list)

Implement the Skiplist class:

Skiplist() Initializes the object of the skiplist. bool search(int target) Returns true if the integer target exists in the Skiplist or false otherwise. void add(int num) Inserts the value num into the SkipList. bool erase(int num) Removes the value num from the Skiplist and returns true. If num does not exist in the Skiplist, do nothing and return false. If there exist multiple num values, removing any one of them is fine. Note that duplicates may exist in the Skiplist, your code needs to handle this situation.

## 1. 解题思路与分析

- 1. 先创建一个跳表节点 Node，需要包含以下三个参数: 节点值 val, 右边节点 right, 下边节点 down
- 2. 跳表初始化:
  - 每一层设置左右两个哨兵节点，左节点的值要比所有值小，右节点的值要比所有值大。因为  $0 \leq num, target \leq 20000$ ，因此左节点值设为 -1，右节点值设为 20001 即可。
  - 跳表的最大层数为 16。因为题目说了最多调用 50000 次 search, add, 以及 erase 操作，也就是底层链表长度 n 最长为 50000，而我们随机的选  $n/2n/2$  个元素做为一级索引，随机选  $n/4n/4$  个元素作为二级索引，……，随机选  $n/2^k n/2$  作为顶层索引。而  $2^{16} = 65536 > 50000$ ,  $2^{16} = 65536 > 50000$ ，所以 16 层足够为长度 50000 的链表建立索引。
  - 每层的两个哨兵节点中，左节点要指向右节点，并且除了最后一层外左右节点都需要指向下一层节点
  - 跳表起始节点 head 即为 left<sup>1</sup>，即顶层的左哨兵节点
- 3. 查找 (search) 方法。从 head 节点开始查找，如果当前节点 cur 的右边节点值比 target 大，应该往下查找；如果 cur 的右边节点值比 target 小，应该往右查找；如果相等，那就是找到了，返回 True。遍历完了也没找到，返回 False。
- 4. 插入 (add) 方法。相对比较复杂，因为还需要根据概率维护索引。
  - 查找插入位置。也是从 head 开始查找插入的位置，每次往下走的节点都需要保存，因为可能在该节点后面插入新的节点作为索引。换句话说，其实查找的是在每一层应该插入的位置并记录在 stack 中，只是不一定每一层都会插入节点罢了。
  - 插入节点。首先最后一层肯定是要插入节点的，插入完成后，记录一下该层节点，因为如果该节点要加入到索引中，上一层是要指向这一层的。以 1/2 的概率生成索引，在上一层插入节点；1/2 的概率插入结束，不再继续维护索引；如果前面生成了索引，再以 1/2 的概率插入节点直到插入到顶层或中途因概率停止插入。
- 5. 删除 (erase) 方法。删除和查找差不多，只不过是对每一层找到的节点都要进行删除，也就是删除节点的同时要删除由其产生的索引节点。

```
private Node head;
public Skiplist() {
    Node[] left = new Node[16]; // 左侧向下
    Node[] right = new Node[16]; // 右侧向下
    for(int i = 0; i < 16; i++) {
        left[i] = new Node(-1); // 左侧哨兵: -1
        right[i] = new Node(20001); // 右侧哨兵: > 20000
    }
    for(int i = 0; i < 15; i++) {
        left[i].right = right[i]; // 前 15 层: 左右连接起来
        left[i].down = left[i + 1]; // 左侧消兵: 向下连接起来
        right[i].down = right[i + 1]; // 右侧消兵: 向下连接起来
    }
    left[15].right = right[15]; // 最底层: 向右连接起来
    head = left[0]; // 总入口
}
public boolean search(int target) {
    Node r = head;
    while (r != null) {
        if (r.right.val > target)
            r = r.down;
        else if (r.right.val < target)
            r = r.right;
        else return true;
    }
    return false;
}
```

```

public void add(int val) {
    Node r = head;
    ArrayDeque<Node> s = new ArrayDeque<>();
    while (r != null) {
        if (r.right.val >= val) {
            s.offerLast(r);
            r = r.down;
        } else r = r.right;
    }
    Node pre = null;
    while (!s.isEmpty()) {
        r = s.pollLast();
        Node cur = new Node(val);
        cur.right = r.right; // 新建节点放入到 r 与 r.right 之间
        r.right = cur;
        if (pre != null) cur.down = pre; // 如果当前层不是最底层，那么设置新层新插入节点的 down 指向下一层新插入同值节点
        pre = cur; // keep updated
        if (Math.random() < 0.5) break; // 根据这个概率来决定，新插入节点是否继续向上增加节点
    }
}
public boolean erase(int val) {
    Node r = head;
    boolean ans = false;
    while (r != null) {
        if (r.right.val >= val) {
            if (r.right.val == val) { // 把当前层的这一节点删除
                ans = true;
                r.right = r.right.right;
            }
            r = r.down; // 向下遍历，寻找并删除当前层之下可能存在的节点
        } else r = r.right;
    }
    return ans;
}
class Node {
    int val;
    Node right, down;
    public Node(int val) {
        this.val = val;
    }
}

```

## 8.11 不知道归纳到哪里，又很巧妙的题

### 8.11.1 2322. Minimum Score After Removals on a Tree - Hard

- There is an undirected connected tree with n nodes labeled from 0 to n - 1 and n - 1 edges.
- You are given a 0-indexed integer array nums of length n where nums[i] represents the value of the ith node. You are also given a 2D integer array edges of length n - 1 where edges[i] = [ai, bi] indicates that there is an edge between nodes ai and bi in the tree.
- Remove two distinct edges of the tree to form three connected components. For a pair of removed edges, the following steps are defined:
  - Get the XOR of all the values of the nodes for each of the three components respectively.
  - The difference between the largest XOR value and the smallest XOR value is the score of the pair.
- For example, say the three components have the node values: [4,5,7], [1,9], and [3,3,3]. The three XOR values are  $4 \wedge 5 \wedge 7 = 6$ ,  $1 \wedge 9 = 8$ , and  $3 \wedge 3 \wedge 3 = 3$ . The largest XOR value is 8 and the smallest XOR value is 3. The score is then  $8 - 3 = 5$ .
- Return the minimum score of any possible pair of edge removals on the given tree.
- 这里有个详细的分析链接 <https://leetcode.cn/problems/minimum-score-after-removals-on-a-tree/solution/dfs-shi-jian-chuo-chu-li-shu-shang-wen-t-x1kk/>

## 何为时间戳?

我们可以在 DFS 一棵树的过程中, 维护一个全局的时间戳  $clock$ , 每访问一个新的节点, 就将  $clock$  加一。同时, 记录进入节点  $x$  的时间戳  $in[x]$ , 和离开(递归结束)这个节点时的时间戳  $out[x]$ 。

## 时间戳有什么性质?

根据 DFS 的性质, 当我们递归以  $x$  为根的子树时, 设  $y$  是  $x$  的子孙节点, 我们必须先递归完以  $y$  为根的子树, 之后才能递归完以  $x$  为根的子树。

从时间戳上看, 如果  $y$  是  $x$  的子孙节点, 那么区间  $[in[y], out[y]]$  必然被区间  $[in[x], out[x]]$  所包含。

反之, 如果区间  $[in[y], out[y]]$  被区间  $[in[x], out[x]]$  所包含, 那么  $y$  必然是  $x$  的子孙节点(换句话说  $x$  是  $y$  的祖先节点)。因此我们可以得出

$$in[x] < in[y] \leq out[y] \leq out[x]$$

来判断  $x$  是否为  $y$  的祖先节点, 由于  $in[y] \leq out[y]$  恒成立, 上式可以简化为

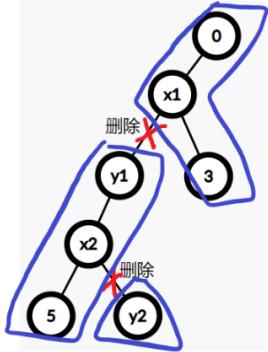
$$in[x] < in[y] \leq out[x]$$

回到本题。由于需要求出子树的异或和, 不妨以 0 为根, DFS 这棵树, 在求出时间戳的同时, 求出每棵以  $x$  为根的子树的异或和  $xor[x]$ 。

由于  $n$  比较小, 我们可以用  $O(n^2)$  的时间枚举要删除的两条边  $x_1-y_1$  和  $x_2-y_2$ , 并假设  $x$  是  $y$  的父节点, 这会产生以下三种情况:

1. 删除的两条边在同一颗子树内, 且  $y_1$  是  $x_2$  的祖先节点(或重合)。

如下图所示, 这三个连通块的异或和分别为  $xor[y_1]$ 、 $xor[y_1] \oplus xor[y_2]$  和  $xor[0] \oplus xor[y_1]$ ( $\oplus$  表示异或运算)。

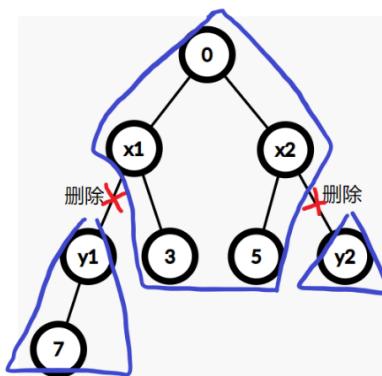


2. 剪除的两条边在同一颗子树内, 且  $y_2$  是  $x_1$  的祖先节点(或重合)。

同上, 这三个连通块的异或和分别为  $xor[y_1]$ 、 $xor[y_2] \oplus xor[y_1]$  和  $xor[0] \oplus xor[y_2]$ 。

3. 剪除的两条边分别属于两颗不相交的子树。

如下图所示, 这三个连通块的异或和分别为  $xor[y_1]$ 、 $xor[y_2]$  和  $xor[0] \oplus xor[y_1] \oplus xor[y_2]$ 。



因此关键之处在于判断这两条边的关系, 这可以用上文提到的时间戳的性质  $O(1)$  地判断出来。

代码实现时, 由于不知道  $edges[i]$  两个点的父子关系, 枚举边的写法需要额外的判断。我们可以改为枚举不是根的两个点, 删掉这两个点及其父节点形成的边, 这样代码更简洁, 效率也略优于枚举边的写法。

## 复杂度分析

- 时间复杂度:  $O(n^2)$ , 其中  $n$  为  $nums$  的长度。
- 空间复杂度:  $O(n)$ 。

Python3 Java C++ Go

```
class Solution {
    List<Integer>[] g;
    int[] nums, xor, in, out;
    int clock;

    public int minimumScore(int[] nums, int[][] edges) {
        var n = nums.length;
        g = new ArrayList[n];
        Arrays.setAll(g, e -> new ArrayList<>());
        for (var e : edges) {
            int u = e[0], v = e[1];
            g[u].add(v);
            g[v].add(u);
        }
        this.a = a;
        xor = new int[n];
        in = new int[n];
        out = new int[n];
        dfs(0, -1);
        var ans = Integer.MAX_VALUE;
        for (int i = 0, x, y, z; i < n; i++) // 就直接把 0 号当作根节点(任何节点作为根节点都无所谓)
            for (int j = 1; j < i; ++j) {
                if (in[i] < in[j] && in[j] <= out[i]) { // i 是 j 的祖先节点
                    x = xor[j];
                    y = xor[i] ^ x;
                    z = xor[0] ^ xor[i];
                } else if (in[j] < in[i] && in[i] <= out[j]) { // j 是 i 的祖先节点
                    x = xor[i];
                    y = xor[j] ^ x;
                    z = xor[0] ^ xor[j];
                } else { // 剪除的两条边分别属于两颗不相交的子树
                    x = xor[i];
                    y = xor[j];
                    z = xor[0] ^ x ^ y;
                }
                ans = Math.min(ans, Math.max(Math.max(x, y), z) - Math.min(Math.min(x, y), z));
                if (ans == 0) return 0; // 提前退出: 这是最小值, 不可能再有更小的了, 大可不必再作无用功
            }
        return ans;
    }

    List<Integer>[] g;
    // xor[i]: 以当前节点为根节点的所有子节点(包括当前根结点)的累积异或结果(所有节点值相异或)
    int[] a, xor, in, out;
    int t;
    void dfs(int u, int p) { // 参数: 当前节点, 与父节点
        in[u] = ++t;
        xor[u] = a[u];
        for (var v : g[u]) { // 遍历与当前节点连通的所有节点
            if (v == p) continue; // 若是父节点, 可以跳过, 因为已经做过了
            dfs(v, u); // 深度优先遍历子节点
            xor[u] ^= xor[v]; // 这里用了极巧妙的递归来求所以子树节点的异或值
        }
    }
}
```

```
    out[u] = t;
}
```

### 8.11.2 2258. Escape the Spreading Fire: 多源头 BFS

You are given a 0-indexed 2D integer array grid of size  $m \times n$  which represents a field. Each cell has one of three values:

```
0 represents grass,  
1 represents fire,  
2 represents a wall that you and fire cannot pass through.
```

You are situated in the top-left cell,  $(0, 0)$ , and you want to travel to the safehouse at the bottom-right cell,  $(m - 1, n - 1)$ . Every minute, you may move to an adjacent grass cell. After your move, every fire cell will spread to all adjacent cells that are not walls.

Return the maximum number of minutes that you can stay in your initial position before moving while still safely reaching the safehouse. If this is impossible, return -1. If you can always reach the safehouse regardless of the minutes stayed, return 109.

Note that even if the fire spreads to the safehouse immediately after you have reached it, it will be counted as safely reaching the safehouse.

A cell is adjacent to another cell if the former is directly north, east, south, or west of the latter (i.e., their sides are touching).

```
public int maximumMinutes(int[][] a) { // 9/55 没有过，不知道是什么原因，改天再写这个  
    m = a.length;  
    n = a[0].length; // mn: kMax  
    r = new int[m * n]; // 火烧到每个方格所需要的最短时间  
    Arrays.fill(r, Integer.MAX_VALUE);  
    buildFireGrid(a); // 把这个方法提出去，就可能让代码各个功能模块更加独立 // <<<<<<<<<<<<<<<<  
  
    int l = 0, h = m*n, ans = -1; // 接下来，再用二分查找法找一个最大的等待时间  
    while (l <= h) { // 判断条件： l <= r 等号狠重要  
        int m = l + (h - l) / 2;  
        if (isSuccessible(m, r, a)) {  
            ans = Math.max(ans, m);  
            l = m + 1;  
        } else h = m-1;  
    }  
    // 再判断一遍，可以永远等吗？【这里，自己最初的想法：是需要再遍历一遍，但是参考别人的，只要答案 ans = m*n 就可以认定，可以永远等!!! 一定要嫁给亲爱的表哥!!!】  
    // 【去掉这个多余步骤之后：】 51 / 55 testcases passed  
    // if (ans == -1) return ans; // 再判断一遍，可以永远等吗？  
    Deque<int> qq = new ArrayDeque<>();  
    qq.offerFirst(new int[] {0, 0});  
    Arrays.stream(vis).forEach(x -> Arrays.fill(x, false));  
    while (!qq.isEmpty()) {  
        for (int size = qq.size()-1; size >= 0; size--) {  
            int [] cur = qq.pollLast();  
            int i = cur[0], j = cur[1];  
            if (i == m-1 && j == n-1) return 1000000000;  
            for (int [] d : dirs) {  
                int x = i + d[0], y = j + d[1];  
                if (x < 0 || x >= m || y < 0 || y >= n || vis[x][y] || a[x][y] != 0 || r[x*n+y] != Integer.MAX_VALUE) continue;  
                vis[x][y] = true;  
                qq.offerFirst(new int[] {x, y});  
            }  
        }  
    }  
    return ans;  
    return ans == m*n ? (int)1e9 : ans;  
}  
int [][] dirs = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};  
int [] r;  
int m, n;  
boolean isSuccessible(int v, int [] r, int [][] a) {  
    Deque<int> q = new ArrayDeque<>();  
    boolean [][] vis = new boolean [m][n];  
    q.offerFirst(new int[] {0, 0});  
    vis[0][0] = true;  
    while (!q.isEmpty()) {  
        v++; // 提到前面来  
        for (int size = q.size()-1; size >= 0; size--) {  
            int [] cur = q.pollLast();  
            int i = cur[0], j = cur[1];  
            // 【不一定：即便最后一个格，也是可能有火烧身的。。。也需要先行判断：因为特权主义，亲爱的表哥在活宝妹这里永远拥有特权!!!】  
            // if (i == m-1 && j == n-1) {  
            //     if (r[i*n+j] < v) continue; // 【不合法解：火及时烧到了终点。。。】  
            //     return true;  
            for (int [] d : dirs) {  
                int x = i + d[0], y = j + d[1];  
                // 这里下面判断用 v+1，也可以把后面 v++ 提前面、循环的前面去  
                // 【看似简洁，漏洞无数：折三段：来判断】：最主要的是，当 r[x*n+y]==v-Continue 时，会过滤掉一个可能存在的正确答案!!! 它给了最后一个终点格特权。。。  
                // if (x < 0 || x >= m || y < 0 || y >= n || vis[x][y] || a[x][y] != 0 || r[x*n+y] <= v+1) continue; // 【折三段：来判断】  
            }  
        }  
    }  
}
```

```

    if (x < 0 || x >= m || y < 0 || y >= n || a[x][y] != 0 || vis[x][y] || r[x*n+y] < v) continue;
    // 【把终止条件结果，放在这里判断】
    if (x == m-1 && y == n-1) { // 【不一定：即便最后一个格，也是可能有火烧身的...】
        if (r[x*n+y] < v) continue;
        return true;
    }
    if (r[x*n+y] == v) continue;

    vis[x][y] = true;
    q.offerFirst(new int [] {x, y});
}
// v++;
}
return false;
}
void buildFireGrid(int [][] a) {
    int time = 0;
    boolean [][] vis = new boolean [m][n];
    Deque<int []> q = new ArrayDeque<()>;
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            if (a[i][j] == 1) {
                q.offerFirst(new int [] {i, j});
                vis[i][j] = true;
            }
    while (!q.isEmpty()) {
        for (int size = q.size()-1; size >= 0; size--) {
            int [] cur = q.pollLast();
            int i = cur[0], j = cur[1];
            r[i*n+j] = time; // 更新纪录：烧到某个方格的最短时间
            for (int [] d : dirs) {
                int x = i + d[0], y = j + d[1];
                if (x < 0 || x >= m || y < 0 || y >= n || vis[x][y] || a[x][y] == 2) continue; // 遍历过了，或是不能穿墙
                vis[x][y] = true;
                q.offerFirst(new int [] {x, y});
            }
        }
        time++;
    }
}

```

### 8.11.3 3311. Construct 2D Grid Matching Graph Layout

You are given a 2D integer array edges representing an undirected graph having n nodes, where edges[i] = [ui, vi] denotes an edge between nodes ui and vi.

Construct a 2D grid that satisfies these conditions:

The grid contains all nodes from 0 to n - 1 in its cells, with each node appearing exactly once.

Two nodes should be in adjacent grid cells (horizontally or vertically) if and only if there is an edge between them in edges.

It is guaranteed that edges can form a 2D grid that satisfies the conditions.

Return a 2D integer array satisfying the conditions above. If there are multiple solutions, return any of them.

```

public int [][] constructGridLayout(int n, int [][] egs) {
    // 【建图】: 无向图
    List<Integer> [] g = new ArrayList[n];
    Arrays.setAll(g, z -> new ArrayList<>());
    for (int [] e : egs) {
        int u = e[0], v = e[1];
        g[u].add(v);
        g[v].add(u);
    }
    // 入度: 【0,4】 每个入度，去找和标记，一个该入度的节点，就可以了。答案的不唯一性
    int [] f = new int [5];
    Arrays.fill(f, -1); // 后结区分，是否存在【0,4】入度的节点
    for (int i = 0; i < n; i++)
        f[g[i].size()] = i;
    // 构建：结果数组的【第一行】
    List<Integer> l = new ArrayList<>();
    if (f[1] != -1) { // 【入度为 1】: 仅只一列
        l.add(f[1]); // 前面，纪录过，入度为 1 的节点、中的一个 f[1]，就用这个
    } else if (f[4] == -1) { // 【入度：最大为 3】: 仅只 2 列
        l.add(f[2]); // 添加：纪录过的【入度为 2】的节点、作为，一个角
        for (int v : g[f[2]]) // 遍历：此角节点的、所有邻居，寻找另一个【入度为 2 的、角节点】
            if (g[v].size() == 2) {
                l.add(v);
                break;
            }
    } else { // 最少三列【至少：三行三列】: 构建【第一行】
        l.add(f[2]); // 添加：纪录过的【入度为 2】的节点、作为，一个角
        int pre = f[2], x = g[f[2]].get(0); // 【至少：三行三列】随便任意一邻居，入度都为 3
        while (g[x].size() == 3) {
            l.add(x);
            for (int y : g[x]) // 遍历：每个连线节点，找一个【入度为 3】的邻居，连一条边

```

```
// if (y != pre && g[y].size() == 3) { // 【写错了】: 这里只需要 == 3 || ==2
if (y != pre && g[y].size() < 4) {
    pre = x;
    x = y;
    break; // break 掉的是: 内嵌 for-loop
}
l.add(x); // 退出 while 前的、最后一个 for 内嵌 x, 它的 g[x].size()==2 , 是第一行边的【另一个角】, 要加上
}
int nn = l.size(), m = n / nn;
int [][] a = new int [m][nn];
boolean [] vis = new boolean [n];
for (int i = 0; i < nn; i++) {
    a[0][i] = l.get(i);
    vis[l.get(i)] = true;
}
for (int i = 1; i < m; i++) // 遍历: 填充, 剩余的行
    for (int j = 0; j < nn; j++) { // 遍历: 列
        for (int v : g[a[i-1][j]]) // 对于: 此格的【此列正上方一格 a[i-1][j】: 其【左上右】三个邻居全填过了
            if (!vis[v]) { // 如果, 没有遍历过, 就填它!
                vis[v] = true;
                a[i][j] = v;
                break;
            }
    }
}
return a;
}
```



# Chapter 9

## 扫描线

### 9.0.1 435. Non-overlapping Intervals - Medium 动态规划贪心

Given an array of intervals intervals where intervals[i] = [starti, endi], return the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

1. 解题思路与分析: 动态规划 O (N^2) tle

#### 方法一：动态规划

##### 思路与算法

题目的要求等价于「选出最多数量的区间，使得它们互不重叠」。由于选出的区间互不重叠，因此我们可以将它们按照端点从小到大的顺序进行排序，并且无论我们按照左端点还是右端点进行排序，得到的结果都是唯一的。

这样一来，我们可以先将所有的  $n$  个区间按照左端点（或者右端点）从小到大进行排序，随后使用动态规划的方法求出区间数量的最大值。设排完序后这  $n$  个区间的左右端点分别为  $l_0, \dots, l_{n-1}$  以及  $r_0, \dots, r_{n-1}$ ，那么我们令  $f_i$  表示「以区间  $i$  为最后一个区间，可以选出的区间数量的最大值」，状态转移方程即为：

$$f_i = \max_{j < i \wedge r_j \leq l_i} \{f_j\} + 1$$

即我们枚举倒数第二个区间的编号  $j$ ，满足  $j < i$ ，并且第  $j$  个区间必须要与第  $i$  个区间不重叠。由于我们已经按照左端点进行升序排序了，因此只要第  $j$  个区间的右端点  $r_j$  没有越过第  $i$  个区间的左端点  $l_i$ ，即  $r_j \leq l_i$ ，那么第  $j$  个区间就与第  $i$  个区间不重叠。我们在所有满足要求的  $j$  中，选择  $f_j$  最大的那一个进行状态转移，如果找不到满足要求的区间，那么状态转移方程中 min 这一项就为 0， $f_i$  就为 1。

最终的答案即为所有  $f_i$  中的最大值。

```
public int eraseOverlapIntervals(int[][] a) { // 刚过去周六早上的比赛简单的当时就想出来 O(NlogN) 的解法了，这里居然还有些不通
    int n = a.length, max = 0;
    Arrays.sort(a, (x, y) -> x[0] != y[0] ? x[0] - y[0] : x[1] - y[1]); // starttime, then end time
    int[] dp = new int[n];
    Arrays.fill(dp, 1);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (a[j][1] > a[i][0]) continue;
            dp[i] = Math.max(dp[i], dp[j] + 1);
        }
        max = Math.max(max, dp[i]);
    }
    return n - max;
}
```

2. 解题思路与分析: 贪心

## 方法二：贪心

### 思路与算法

我们不妨想一想应该选择哪一个区间作为首个区间。

假设在某一种**最优**的选择方法中， $[l_k, r_k]$  是首个（即最左侧的）区间，那么它的左侧没有其它区间，右侧有若干个不重叠的区间。设想一下，如果此时存在一个区间  $[l_j, r_j]$ ，使得  $r_j < r_k$ ，即区间  $j$  的右端点在区间  $k$  的左侧，那么我们将区间  $k$  替换为区间  $j$ ，其与剩余右侧被选择的区间仍然是不重叠的。而当我们把区间  $k$  替换为区间  $j$  后，就得到了另一种**最优**的选择方法。

我们可以不断地寻找右端点在首个区间右端点左侧的新区间，将首个区间替换成该区间。那么当我们无法替换时，**首个区间就是所有可以选择的区间中右端点最小的那个区间**。因此我们将所有区间按照右端点从小到大进行排序，那么排完序之后的首个区间，就是我们选择的首个区间。

如果有多个区间的右端点都同样最小怎么办？由于我们选择的是首个区间，因此在左侧不会有其它的区间，那么左端点在何处是不重要的，我们只要任意选择一个右端点最小的区间即可。

当确定了首个区间之后，所有与首个区间不重合的区间就组成了一个规模更小的子问题。由于我们已经在初始时将所有区间按照右端点排好序了，因此对于这个子问题，我们无需再次进行排序，只要找出其中**与首个区间不重合并且右端点最小的区间**即可。用相同的方法，我们可以依次确定后续的所有区间。

在实际的代码编写中，我们对按照右端点排好序的区间进行遍历，并且实时维护上一个选择区间的右端点  $right$ 。如果当前遍历到的区间  $[l_i, r_i]$  与上一个区间不重合，即  $l_i \geq right$ ，那么我们就可以贪心地选择这个区间，并将  $right$  更新为  $r_i$ 。

```
public int eraseOverlapIntervals(int[][] intervals) {
    if (intervals.length == 0) return 0;
    Arrays.sort(intervals, (a, b) ->a[1] - b[1]);
    int n = intervals.length;
    int right = intervals[0][1];
    int ans = 1;
    for (int i = 1; i < n; ++i) {
        if (intervals[i][0] >= right) {
            ++ans;
            right = intervals[i][1];
        }
    }
    return n - ans;
}
```

## 9.0.2 2054. Two Best Non-Overlapping Events - Medium

You are given a 0-indexed 2D integer array of events where  $events[i] = [startTmei, endTimei, valuei]$ . The  $i$ th event starts at  $startTmei$  and ends at  $endTimei$ , and if you attend this event, you will receive a value of  $valuei$ . You can choose at most two non-overlapping events to attend such that the sum of their values is maximized.

Return this maximum sum.

Note that the start time and end time is inclusive: that is, you cannot attend two events where one of them starts and the other ends at the same time. More specifically, if you attend an event with end time  $t$ , the next event must start at or after  $t + 1$ .

### 1. 解题思路与分析

- 因为最多只有两个事件，联想到股票问题的只有两次交易，很容易想到从头到尾扫一遍，从尾到头扫一遍
- 也是一道扫描线问题，借助 TreeMap, PriorityQueue 等数据结构，或 binary search 算法帮助保证正确性以及提速
- 再熟悉一下 NavigableMap 类中的 floorKey() 和 ceilingKey() 的 API：

类型	方法	描述
K	ceilingKey(K key)	返回大于或等于给定键的最小键，如果没有这样的键，则 null
K	floorKey(K key)	返回小于或等于给定键的最大键，如果没有这样的键，则 null

```
public int maxTwoEvents(int[][] events) {
    int n = events.length, maxSeenVal = 0; // for any timestamp - find max of values to left and max of values to right
    Arrays.sort(events, (a,b) ->a[1] - b[1]); // Sort by end time - to know what events happened previously
```

```

TreeMap<Integer, Integer> leftMaxSeen = new TreeMap<>();
for (int [] event : events) {
    int eventVal = event[2];
    if (eventVal >= maxSeenVal) {
        maxSeenVal = eventVal;
        leftMaxSeen.put(event[1], maxSeenVal); // save maxSeenVal event value seen so far at END timestamp
    }
}
// Sort by start time but in descending order - as we parse events and store max value of future events to current event start timestamp
Arrays.sort(events, (a,b) -> b[0] - a[0]);
int ans = 0, maxSeenRight = 0;
for (int [] event : events) {
    int eventVal = event[2];
    if (eventVal >= maxSeenRight) {
        maxSeenRight = eventVal;
        Integer maxOnLeftKey = leftMaxSeen.floorKey(event[0] - 1);
        if (maxOnLeftKey != null)
            ans = Math.max(ans, leftMaxSeen.get(maxOnLeftKey) + maxSeenRight);
    }
}
return Math.max(maxSeenVal, ans); // 有一种单个事件得最大值的情况不能漏掉
}

```

- 扫描线算法

```

public int maxTwoEvents(int[][] events) {
    int result = 0, maxOfCompletedEvents = 0;
    Arrays.sort(events, (x,y) -> x[0] - y[0]); // Sort by Start time
    PriorityQueue<int[]> inProgressQueue = new PriorityQueue<>((x,y)->x[1]-y[1]); // sorted by end time
    for (int[] currentEvent : events) {
        while (!inProgressQueue.isEmpty() && inProgressQueue.peek()[1] < currentEvent[0])
            maxOfCompletedEvents = Math.max(maxOfCompletedEvents, inProgressQueue.poll()[2]);
        result = Math.max(result, maxOfCompletedEvents + currentEvent[2]);
        inProgressQueue.offer(currentEvent);
    }
    return result;
}

```

- 二分查找: 改天补上

### 9.0.3 2055. Plates Between Candles - Medium

There is a long table with a line of plates and candles arranged on top of it. You are given a 0-indexed string  $s$  consisting of characters '\*' and '|' only, where a '\*' represents a plate and a '|' represents a candle.

You are also given a 0-indexed 2D integer array  $\text{queries}$  where  $\text{queries}[i] = [\text{left}_i, \text{right}_i]$  denotes the substring  $s[\text{left}_i \dots \text{right}_i]$  (inclusive). For each query, you need to find the number of plates between candles that are in the substring. A plate is considered between candles if there is at least one candle to its left and at least one candle to its right in the substring.

For example,  $s = "||**|***|"$ , and a query  $[3, 8]$  denotes the substring " $*|***|$ ". The number of plates between candles in this substring is 2, as each of the two plates has at least one candle in the substring to its left and right. Return an integer array  $\text{answer}$  where  $\text{answer}[i]$  is the answer to the  $i$ th query.

#### 1. 解题思路与分析

- 先把每个调用的最左、以及最右的蜡烛的位置找出来，假如作必要的前置处理的话，可以做到  $O(1)$  时间
- 而要数这最左与最右蜡烛之间的盘子个数的话，如果我们前置数清楚所有位置蜡烛个数，我们也可以做到  $O(1)$  时间
- 所以，使用三个数组，一个记录各个位置蜡烛总个数，另两个分别纪录左右端点

```

public int[] platesBetweenCandles(String t, int[][] queries) {
    int n = t.length();
    int [] sum = new int [n+1];
    int [] pre = new int [n+1], suf = new int [n+1];
    char [] s = t.toCharArray();
    for (int i = 0; i < n; i++) {
        sum[i+1] = sum[i] + (s[i] == '|' ? 1 : 0);
        pre[i+1] = s[i] == '|' ? i : pre[i]; // pre[i] matches i-1
    }
    for (int i = n-1; i >= 0; i--)
        suf[i] = s[i] == '|' ? i : suf[i+1]; // suf[i] matches i
    int [] ans = new int [queries.length];
    for (int i = 0; i < queries.length; i++) {
        int l = suf[queries[i][0]], r = pre[queries[i][1]+1]; // 注意：右蜡烛边界
        if (l < r)
            ans[i] = r - l - (sum[r] - sum[l]);
    }
    return ans;
}

```

## 9.0.4 218. The Skyline Problem - Hard

A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Given the locations and heights of all the buildings, return the skyline formed by these buildings collectively.

The geometric information of each building is given in the array buildings where `buildings[i] = [lefti, righti, heighti]`:

`lefti` is the x coordinate of the left edge of the *i*th building. `righti` is the x coordinate of the right edge of the *i*th building. `heighti` is the height of the *i*th building. You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.

The skyline should be represented as a list of "key points" sorted by their x-coordinate in the form `[[x1,y1],[x2,y2],...]`. Each key point is the left endpoint of some horizontal segment in the skyline except the last point in the list, which always has a y-coordinate 0 and is used to mark the skyline's termination where the rightmost building ends. Any ground between the leftmost and rightmost buildings should be part of the skyline's contour.

Note: There must be no consecutive horizontal lines of equal height in the output skyline. For instance, `[..., [2 3], [4 5], [7 5], [11 5], [12 7], ...]` is not acceptable; the three lines of height 5 should be merged into one in the final output as such: `[..., [2 3], [4 5], [12 7], ...]`

### 1. 解题思路与分析

- 复杂度分析

时间复杂度:  $O(n \log n)$ , 其中  $n$  为建筑数量。每座建筑至多只需要入队与出队一次, 单次时间复杂度为  $O(\log n)$ 。

空间复杂度:  $O(n)$ , 其中  $n$  为建筑数量。数组 `boundaries` 和优先队列的空间占用均为  $O(n)$ 。

```
public List<List<Integer>> getSkyline(int[][] a) {
    List<Integer> pos = new ArrayList<>();
    for (int [] b : a) {
        pos.add(b[0]);
        pos.add(b[1]);
    }
    Collections.sort(pos);
    List<List<Integer>> ans = new ArrayList<>();
    Queue<int []> q = new PriorityQueue<>((x, y) -> y[1] - x[1]);
    int n = a.length, idx = 0;
    for (int v : pos) {
        while (idx < n && a[idx][0] <= v) {
            q.offer(new int [] {a[idx][1], a[idx][2]});
            idx++;
        }
        while (!q.isEmpty() && q.peek()[0] <= v) q.poll();
        int maxHiCur = q.isEmpty() ? 0 : q.peek()[1];
        if (ans.size() == 0 || maxHiCur != ans.get(ans.size() - 1).get(1))
            ans.add(Arrays.asList(v, maxHiCur));
    }
    return ans;
}
```

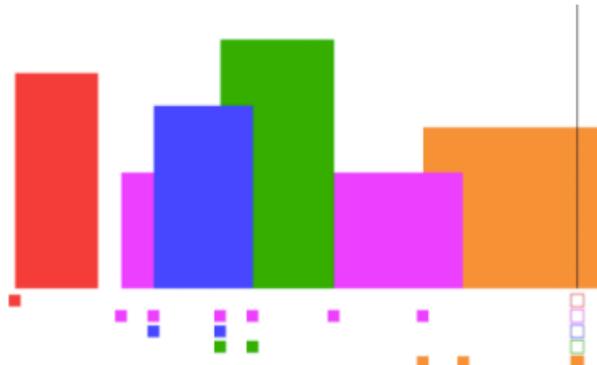
### 2. 解题思路与分析

使用扫描线, 从左至右扫过。如果遇到左端点, 将高度入堆, 如果遇到右端点, 则将高度从堆中删除。使用 `last` 变量记录上一个转折点。 $O(N \log N)$

#### 扫描线法

使用扫描线, 从左至右扫过。如果遇到左端点, 将高度入堆, 如果遇到右端点, 则将高度从堆中删除。使用 `last` 变量记录上一个转折点。

可以参考下面的图, 扫描线下方的方格就是堆。



```

public List<List<Integer>> getSkyline(int[][] buildings) { // 从左向右扫一遍过去
    List<List<Integer>> ans = new ArrayList<>();
    Map<Integer, List<Integer>> map = new TreeMap<>(); // TreeMap, 取其根据键的大小自动排序之意, 可以得到 logN 的效率
    for (int [] v : buildings) {
        map.computeIfAbsent(v[0], z -> new ArrayList<>()).add(-v[2]); // 插入左节点: 高度为负
        map.computeIfAbsent(v[1], z -> new ArrayList<>()).add(v[2]); // 插入右节点: 高度为正
    }
    Map<Integer, Integer> heights = new TreeMap<>((a, b) -> b-a); // 保留当前位置的所有高度 重定义排序: 从大到小
    int [] last = {0, 0}; // 保留上一个位置的横坐标及高度
    for (Integer key : map.keySet()) {
        List<Integer> ys = map.get(key);
        Collections.sort(ys);
        for (int y : ys) {
            if (y < 0) // 左端点, 高度入队
                heights.put(-y, heights.getOrDefault(-y, 0) + 1);
            else { // 右端点移除高度
                heights.put(y, heights.getOrDefault(y, 1) - 1);
                if (heights.get(y) == 0) heights.remove(y);
            }
        }
        Integer maxHeight = 0; // 获取 heights 的最大值: 就是第一个值
        if (!heights.isEmpty())
            maxHeight = heights.keySet().iterator().next();
        if (last[1] != maxHeight) { // 如果当前最大高度不同于上一个高度, 说明其为转折点
            last[0] = key; // 更新 last, 并加入结果集
            last[1] = maxHeight;
            ans.add(Arrays.asList(key, maxHeight));
        }
    }
    return ans;
}

```

### 3. 解题思路与分析: 比较慢

思路: 类似于 meeting room 的方法, 用 hashmap 存《start, +1》和《end, -1》。在这里就是先用一个 List 《int[]》来存储《start, height》和《end, -height》, 然后把这个 list 按照从小到大排序, 最后用一个 queue 来走一遍每个点, 每次遇到正值的 height, 就 offer, 遇到负值就 pop 此值, 然后在每一个点, 都 peek heap 中的最大 height, 如果这个 height 和之前存在 result 里的值不同, 将此点和值存进 result, 否则更新 pre, 继续向后走, 直到走完。

```

public List<List<Integer>> getSkyline(int[][] buildings) {
    List<List<Integer>> ans = new ArrayList<>();
    List<int[]> hite = new ArrayList<>();
    for (int [] v : buildings) {
        hite.add(new int [] {v[0], v[2]});
        hite.add(new int [] {v[1], -v[2]});
    }
    Collections.sort(hite, (a, b)-> b[0] != a[0] ? a[0] - b[0] : b[1] - a[1]);
    Queue<Integer> q = new PriorityQueue<>((a, b)-> b - a);
    q.offer(0);
    int cur = 0, pre = 0;
    for (int [] h : hite) {
        if (h[1] > 0) q.offer(h[1]);
        else q.remove(-h[1]);
        cur = q.peek();
        if (pre != cur) {
            ans.add(List.of(h[0], cur));
            pre = cur;
        }
    }
    return ans;
}

```

### 4. 解题思路与分析: Java divide and conquer solution beats 96% todo

The basic idea is divide the buildings into two subarrays, calculate their skylines respectively, then merge two skylines together.

- <https://leetcode.com/problems/the-skyline-problem/discuss/61281/Java-divide-and-conquer-solution-best-time-complexity>

```

public List<List<Integer>> getSkyline(int[][] buildings) { // todo; divide and conque 代码被我改乱了
    return merge(buildings, 0, buildings.length-1); // 今天这个题看累了, 改天再补这个
}
private List<List<Integer>> merge(int[][] buildings, int lo, int hi) {
    List<List<Integer>> res = new LinkedList<>();
    if (lo > hi) {
        return res;
    } else if (lo == hi) {
        res.add(List.of(buildings[lo][0], buildings[lo][2]));
        res.add(List.of(buildings[lo][1], 0));
        return res;
    }
    int mid = lo+(hi-lo)/2;
    List<List<Integer>> left = merge(buildings, lo, mid);
    List<List<Integer>> right = merge(buildings, mid+1, hi);
    int lefth = 0, righth = 0;

```

```

while(!left.isEmpty() || !right.isEmpty()) {
    long x1 = left.isEmpty()? Long.MAX_VALUE: left.peekFirst()[0];
    long x2 = right.isEmpty()? Long.MAX_VALUE: right.peekFirst()[0];
    int x = 0;
    if(x1 < x2) {
        int[] temp = left.pollFirst();
        x = temp[0];
        leftH = temp[1];
    } else if(x1 > x2) {
        int[] temp = right.pollFirst();
        x = temp[0];
        rightH = temp[1];
    } else {
        x = left.peekFirst()[0];
        leftH = left.pollFirst()[1];
        rightH = right.pollFirst()[1];
    }
    int h = Math.max(leftH, rightH);
    if(res.isEmpty() || h != res.peekLast()[1])
        res.add(List.of(x, h));
}
return res;
}

```

### 9.0.5 715. Range Module - Hard

A Range Module is a module that tracks ranges of numbers. Design a data structure to track the ranges represented as half-open intervals and query about them.

A half-open interval  $[left, right)$  denotes all the real numbers  $x$  where  $left \leq x < right$ .

Implement the RangeModule class:

RangeModule() Initializes the object of the data structure. void addRange(int left, int right) Adds the half-open interval  $[left, right)$ , tracking every real number in that interval. Adding an interval that partially overlaps with currently tracked numbers should add any numbers in the interval  $[left, right)$  that are not already tracked. boolean queryRange(int left, int right) Returns true if every real number in the interval  $[left, right)$  is currently being tracked, and false otherwise. void removeRange(int left, int right) Stops tracking every real number currently being tracked in the half-open interval  $[left, right)$ .

```

class Range {
    int left, right;
    public Range(int left, int right) {
        this.left = left;
        this.right = right;
    }
}
TreeSet<Range> ts;
public RangeModule() {
    ts = new TreeSet<Range>((a, b)->(a.left != b.left ? a.left - b.left : a.right - b.right));
}
public void addRange(int left, int right) {
    int nl = left, nr = right;
    Range high = new Range(right, Integer.MAX_VALUE);
    while (true) {
        Range r = ts.lower(high);
        if (r == null || r.right < left) break;
        if (r.right > right) nr = r.right;
        if (r.left < left) nl = r.left;
        ts.remove(r);
    }
    ts.add(new Range(nl, nr));
}
public boolean queryRange(int left, int right) {
    Range target = ts.floor(new Range(left, Integer.MAX_VALUE));
    return target != null && target.left <= left && target.right >= right;
}
public void removeRange(int left, int right) {
    Range high = new Range(right, right);
    while (true) {
        Range r = ts.lower(high);
        if (r == null || r.right <= left) break;
        if (r.right > right)
            ts.add(new Range(right, r.right));
        if (r.left < left)
            ts.add(new Range(r.left, left));
        ts.remove(r);
    }
}

```

### 9.0.6 352. Data Stream as Disjoint Intervals - Hard

Given a data stream input of non-negative integers  $a_1, a_2, \dots, a_n$ , summarize the numbers seen so far as a list of disjoint intervals.

Implement the SummaryRanges class:

SummaryRanges() Initializes the object with an empty stream. void addNum(int val) Adds the integer val to the stream. int[][] getIntervals() Returns a summary of the integers in the stream currently as a list of disjoint intervals [starti, endi].

```
class Range implements Comparable<Range> {
    int bgn, end;
    public Range(int bgn, int end) {
        this.bgn = bgn;
        this.end = end;
    }
    @Override public int compareTo(Range other) {
        return this.bgn - other.bgn;
    }
}
TreeSet<Range> ts;
public SummaryRanges() {
    ts = new TreeSet<Range>();
}
public void addNum(int val) {
    Range cur = new Range(val, val);
    Range bef = ts.floor(cur);
    Range aft = ts.ceiling(cur);
    if (bef != null && bef.end + 1 >= val) {
        cur.bgn = bef.bgn;
        cur.end = Math.max(val, bef.end);
        ts.remove(bef);
    }
    if (aft != null && aft.bgn == val + 1) {
        cur.end = aft.end;
        ts.remove(aft);
    }
    ts.add(cur);
}
public int[][][] getIntervals() {
    int [][] ans = new int [ts.size()][2];
    int i = 0;
    for (Range cur : ts) {
        ans[i][0] = cur.bgn;
        ans[i][1] = cur.end;
        i++;
    }
    return ans;
}
```

### 9.0.7 1419. Minimum Number of Frogs Croaking - Medium

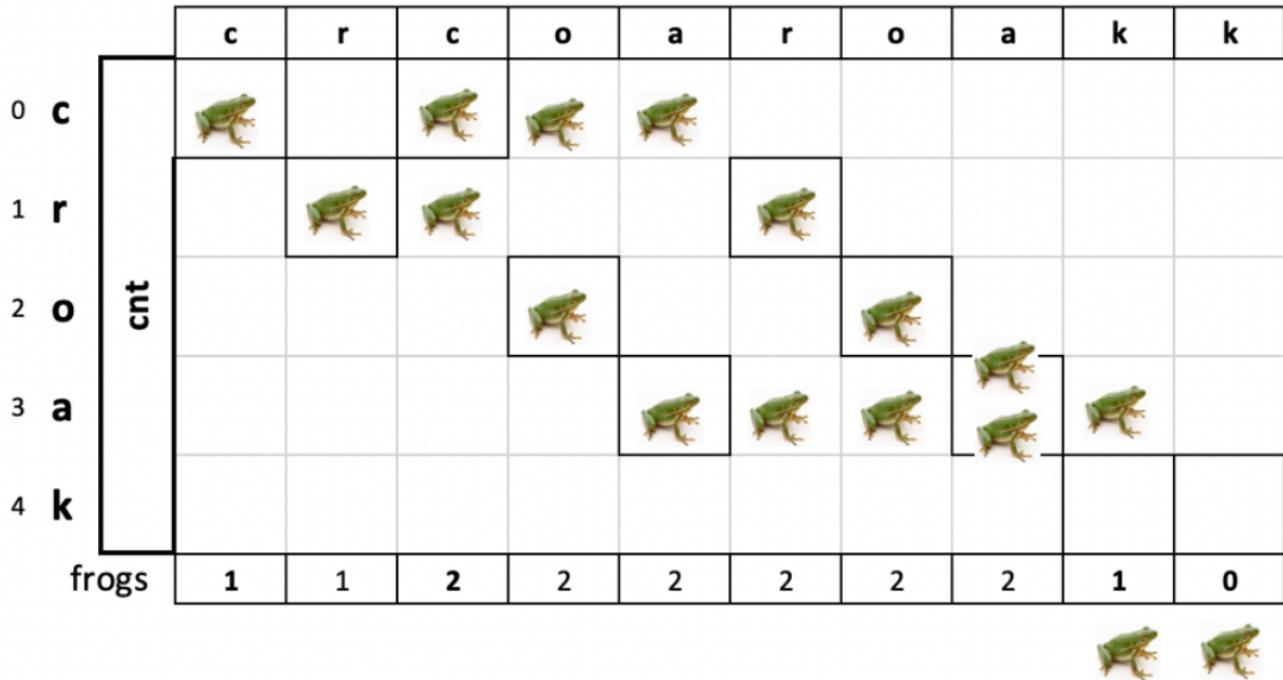
Given the string croakOfFrogs, which represents a combination of the string "croak" from different frogs, that is, multiple frogs can croak at the same time, so multiple "croak" are mixed. Return the minimum number of different frogs to finish all the croak in the given string.

A valid "croak" means a frog is printing 5 letters 'c', 'r', 'o', 'a', 'k' sequentially. The frogs have to print all five letters to finish a croak. If the given string is not a combination of valid "croak" return -1.

```
public int minNumberOfFrogs(String s) { // 写得有点儿乱
    int n = s.length();
    int cc = 0, cr = 0, co = 0, ca = 0, cnt = 0, max = 0;
    for (int i = 0; i < n; i++) {
        char c = s.charAt(i);
        if (c == 'c') {
            ++cc;
            ++cnt;
            max = Math.max(max, cnt);
        } else if (c == 'r') {
            if (cc == 0) return -1;
            --cc;
            ++cr;
        } else if (c == 'o') {
            if (cr == 0) return -1;
            --cr;
            ++co;
        } else if (c == 'a') {
            if (co == 0) return -1;
            --co;
            ++ca;
        } else if (c == 'k') {
            if (ca == 0) return -1;
            --ca;
            --cnt;
        }
    }
    if (cc + cr + co + ca > 0) return -1;
    return max;
}
```

We can track how many frogs are 'singing' each letter in `cnt`:

- Increase number of frogs singing this letter, and decrease number singing previous letter.
- When a frog sings 'c', we increase the number of (simultaneous) frogs .
- When a frog sings 'k', we decrease the number of (simultaneous) frogs .
- If some frog is singing a letter, but no frog sang the previous letter, we return -1 .



Track and return the maximum number of frogs ever singing together.

Catch: if some frog hasn't finished croaking, we need to return -1 .

```
public int minNumberOfFrogs(String s) {
    int n = s.length();
    int cnt [] = new int [5];
    int cur = 0, max = 0;
    for (int i = 0; i < n; i++) {
        char c = s.charAt(i);
        int idx = "croak".indexOf(c);
        ++cnt[idx];
        if (idx == 0)
            max = Math.max(max, ++cur);
        else if (--cnt[idx-1] < 0)
            return -1;
        else if (idx == 4)
            --cur;
    }
    return cur == 0 ? max : -1; // 如果最后所有的青蛙都叫完了的话
}
```

### • 解法三:

这个跟上面的解法差不多，优化了查询位置的时间。直接对比每一步的结果。如果当前字符位的个数比上一位多，那么说明当前位没有办法被匹配成功。

```
public int minNumberOfFrogs(String croakOfFrogs) {
    char[] ch = croakOfFrogs.toCharArray();
    int curr = 0;
    int res = 0;
    int c = 0, r = 0, o = 0, a = 0, k = 0;
    for(int i = 0; i < ch.length; i++) {
        if(ch[i] == 'c') {
            c++;
            curr++;
        } else if(ch[i] == 'r')
            r++;
        else if(ch[i] == 'o')
            o++;
        else if(ch[i] == 'a')
            a++;
        else {
            k++;
            curr--;
        }
    }
    if(c != res || r != res || o != res || a != res)
        return -1;
    return curr;
}
```

```
    }
    res = Math.max(res, curr);
    if (c < r || r < o || o < a || a < k) // 要满足所有的条件
        return -1;
}
if ((c == r) && (r == o) && (o == a) && (a == k))
    return res;
return -1;
}
```



# Chapter 10

## 单调栈

### 10.0.1 975. Odd Even Jump - Hard

You are given an integer array arr. From some starting index, you can make a series of jumps. The (1st, 3rd, 5th, ...) jumps in the series are called odd-numbered jumps, and the (2nd, 4th, 6th, ...) jumps in the series are called even-numbered jumps. Note that the jumps are numbered, not the indices.

You may jump forward from index i to index j (with  $i < j$ ) in the following way:

During odd-numbered jumps (i.e., jumps 1, 3, 5, ...), you jump to the index j such that  $\text{arr}[i] \leq \text{arr}[j]$  and  $\text{arr}[j]$  is the smallest possible value. If there are multiple such indices j, you can only jump to the smallest such index j. During even-numbered jumps (i.e., jumps 2, 4, 6, ...), you jump to the index j such that  $\text{arr}[i] \geq \text{arr}[j]$  and  $\text{arr}[j]$  is the largest possible value. If there are multiple such indices j, you can only jump to the smallest such index j. It may be the case that for some index i, there are no legal jumps. A starting index is good if, starting from that index, you can reach the end of the array (index  $\text{arr.length} - 1$ ) by jumping some number of times (possibly 0 or more than once).

Return the number of good starting indices.

#### 1. 解题思路与分析: 树映射 (Tree Map)

在方法一中, 原问题简化为: 奇数次跳跃时, 对于一些索引 i, 下一步应该跳到哪里去 (如果有的话)。

我们可以使用 TreeMap, 一个维护有序数据的绝佳数据结构。我们将索引 i 映射到  $v = A[i]$  上。

从  $i = N-2$  到  $i = 0$  的遍历过程中, 对于  $v = A[i]$ , 我们想知道比它略大一点和略小一点的元素是谁。TreeMap.lowerKey 与 TreeMap.higherKey 函数就是用来做这样一件事情的。

了解这一点之后, 解法接下来的内容就非常直接了: 我们使用动态规划来维护 odd[i] 和 even[i]: 从索引 i 出发奇数次跳跃与偶数次跳跃是否能到达数组末尾。

- 复杂度分析

时间复杂度:  $O(N \log N)$ , 其中 NN 是数组 A 的长度。

空间复杂度:  $O(N)$

```
public int oddEvenJumps(int[] a) { // 写出这种状况: 就该强调, 选对数据结构, 事半功倍!
    int n = a.length, ans = 1; // n-1 算答案里的一个
    boolean[] odd = new boolean[n];
    boolean[] evn = new boolean[n];
    TreeMap<Integer, Integer> m = new TreeMap<>(); // 这里用 treemap 就比用 ArrayDeque 好用
    odd[n-1] = evn[n-1] = true;
    m.put(a[n-1], n-1);
    for (int i = n-2; i >= 0; i--) {
        Integer higher = m.ceilingKey(a[i]); // higherKey() 返回的是键 key
        Integer lower = m.floorKey(a[i]); // lowerKey()
        if (higher != null) odd[i] = evn[m.get(higher)]; // 仍需取值, 而非用键
        if (lower != null) evn[i] = odd[m.get(lower)];
        if (odd[i]) ans++;
        m.put(a[i], i);
    }
    return ans;
}
```

#### 2. 解题思路与分析: 单调栈

首先, 我们可以发现下一步应该跳到哪里只与我们当前的位置与跳跃次数的奇偶性有关系。

对于每一种状态, 接下来可以跳到的状态一定只有一种 (或者接下来不能跳跃了)。如果我们使用某种方法知道了不同状态之间的转移关系, 我们就可以通过一次简单的遍历解决这个问题了。

于是, 问题就简化为了: 从索引 i 进行奇数次跳跃时, 下一步应该跳到哪里去 (如果有的话)。偶数次跳跃也是类似的。

假设当前是奇数次跳跃，让我们来搞清楚在索引  $i$  的位置接下来应该跳到哪里去。

我们从小到大考虑数组  $A$  中的元素。假设当前我们正在考虑  $A[j] = v$ ，在我们已经处理过但是还未确定下一步跳跃位置的索引中（也就是  $<= v$  的那些）进行搜索。如果我们找到了某些已经处理过的值  $v_0 = A[i]$  且  $i < j$ ，那么我们就可以知道从索引  $i$  下一步应该跳到索引  $j$  的位置。

这种朴素的方法有一点点慢，然而我们可以使用一个很常见的技巧单调栈来加速这个过程。

我们在栈中保存所有已经处理过的索引  $i$ ，并且时时刻刻维护这个栈中的元素是递减的。当我们增加一个新的索引  $j$  的时候，我们弹出栈顶比较小的索引  $i < j$ ，并且记录这些索引下一步全都会跳跃到索引  $j$ 。

然后，我们就知道所有的  $\text{oddnext}[i]$ ，也就是位于索引  $i$  在奇数次跳跃时将会跳到的位置。使用类似的方法，我们也可以求出  $\text{evennext}[i]$ 。有了这些信息，我们就可以使用动态规划的技巧快速建立所有可达状态。

上面的方法在时间复杂度上并没有优化，而是使用单调栈结构（Monotonic Stack）去解决问题，问题的本质还是求从某一点出发，当进行奇数跳时，他能到达什么位置；然而进行偶数跳时，他又能到达什么位置，将这些信息记录在数组当中。那么问题的关键是如何利用单调栈来求得每一个能到达的相应位置。

```
public int oddEvenJumps(int[] a) {
    int n = a.length, ans = 1;
    ArrayDeque<Integer> s = new ArrayDeque<>();
    Integer[] idx = new Integer[n]; // idx 存储的都是 A 数组的下标
    for (int i = 0; i < n; i++) idx[i] = i;
    Arrays.sort(idx, (x, y) -> a[x] - a[y]); // 将 A 数组的下标按 A 中的元素的大小进行排序
    int[] odIdx = new int[n];
    for (int i = 0; i < n; i++) {
        while (!s.isEmpty() && s.peekLast() < idx[i]) // 利用单调栈获取每一个元素进行奇数跳时所到达的位置
            odIdx[s.pollLast()] = idx[i];
        s.offerLast(idx[i]);
    }
    Arrays.sort(idx, (x, y) -> a[y] - a[x]);
    s.clear();
    int[] enIdx = new int[n];
    for (int i = 0; i < n; i++) {
        while (!s.isEmpty() && s.peekLast() < idx[i])
            enIdx[s.pollLast()] = idx[i];
        s.offerLast(idx[i]);
    }
    boolean[] odd = new boolean[n], evn = new boolean[n];
    odd[n-1] = evn[n-1] = true;
    for (int i = n-2; i >= 0; i--) {
        odd[i] = evn[odIdx[i]];
        evn[i] = odd[enIdx[i]];
        if (odd[i]) ans++;
    }
    return ans;
}
```

### 10.0.2 1793. Maximum Score of a Good Subarray - Hard

You are given an array of integers  $\text{nums}$  (0-indexed) and an integer  $k$ .

The score of a subarray  $(i, j)$  is defined as  $\min(\text{nums}[i], \text{nums}[i+1], \dots, \text{nums}[j]) * (j - i + 1)$ . A good subarray is a subarray where  $i \leq k \leq j$ .

Return the maximum possible score of a good subarray.

#### 1. 解题思路与分析: 左右两个单调栈

```
public int maximumScore(int[] a, int k) {
    int n = a.length, ans = 0;
    List<Integer> li = new ArrayList<>();
    List<Integer> ri = new ArrayList<>();
    li.add(k);
    ri.add(k);
    for (int i = k-1; i >= 0; i--)
        if (a[i] < a[li.get(li.size()-1)]) li.add(i);
    for (int i = k+1; i < n; i++)
        if (a[i] < a[ri.get(ri.size()-1)]) ri.add(i);
    int i = 0, j = 0, nl = li.size(), nr = ri.size();
    while (i < nl || j < nr) {
        int l = (i == nl - 1) ? -1 : li.get(i+1);
        int r = (j == nr - 1) ? n : ri.get(j+1);
        ans = Math.max(ans, Math.min(a[li.get(i)], a[ri.get(j)]) * (r - l - 1));
        if (i == nl - 1 && j == nr - 1) break;
        else if (i == nl - 1) j++;
        else if (j == nr - 1) i++;
        else if (a[l] <= a[r]) j++;
        else i++;
    }
    return ans;
}
```

#### 2. 解题思路与分析: 用真实的栈

- 上面的解法是用数组来代替栈以提高效率，真正用栈的实现如下：

```

public int maximumScore(int[] a, int k) { // 这个方法速度很慢，上面一个比较快
    ArrayDeque<Integer> s = new ArrayDeque<>();
    int ans = 0;
    for (int i = 0; i < a.length; i++) {
        while (!s.isEmpty() && a[s.peekLast()] > a[i]) {
            int idx = s.pollLast();
            int l = -1;
            if (!s.isEmpty()) l = s.peekLast();
            int cur = (i - l - 1) * a[idx];
            if (l + 1 <= k && i - 1 >= k) ans = Math.max(ans, cur);
        }
        s.offerLast(i);
    }
    while (!s.isEmpty()) {
        int idx = s.pollLast();
        int l = -1;
        if (!s.isEmpty()) l = s.peekLast();
        int cur = (a.length - l - 1) * a[idx];
        if (l <= k) ans = Math.max(ans, cur);
    }
    return ans;
}

```

### 3. 解题思路与分析: two pointers

```

public int maximumScore(int[] a, int k) { // O(N) Two Pointers
    int n = a.length, ans = a[k];
    int l = k, r = k, min = a[k];
    while (true) {
        while (r+1 < n && a[r+1] >= min) r++;
        while (l-1 >= 0 && a[l-1] >= min) l--;
        ans = Math.max(ans, min*(r - l + 1));
        if (l == 0 && r == n-1) break;
        if (l == 0) min = a[r+1];
        else if (r == n-1) min = a[l-1];
        else min = Math.max(a[r+1], a[l-1]);
    }
    return ans;
}

```



# Chapter 11

## 字符串

### 11.1 总结

- Rabin-Karp Rolling Hash 算法

### 11.2 KMP 算法

#### 11.2.1 1392. Longest Happy Prefix - Hard : Use Longest Prefix Suffix (KMP-table) or String Hashing.

A string is called a happy prefix if it is a non-empty prefix which is also a suffix (excluding itself).

Given a string s, return the longest happy prefix of s. Return an empty string "" if no such prefix exists.

##### 1. 解题思路与分析: KMP 算法

给定一个字符串, 其长度为  $k$  的前缀和长度为  $k$  后缀有可能相等, 问最大的  $k$  对应的那个前缀是什么。

题目本质上是求 KMP 算法中的 next 数组 (KMP 有个原始版本和优化版本, 这里指的是原始版本。对于原始版本,  $n[i]$  表示  $s[0:i-1]$  中最长相等前后缀的长度, 规定  $n[0] = -1$ )。

我们考虑  $n[i]$  和  $n[i-1]$  的递推关系。

首先  $n[1] = 0$ , 我们是可以确定的。对于  $n[i-1]$ , 假设  $n[i-1] = t$ , 意味着  $s[0:t-1] = s[i-t-1:i-2]$ :

1、若  $s[i-1] = s[t]$ , 那么就有  $s[0:t] = s[i-t-1:i-1]$ , 所以  $n[i] \geq t+1$ 。但如果  $n[i] > t+1$  的话, 就会导致  $s[0:t+1] = s[i-t-2:i-1]$ , 从而有  $s[0:t] = s[i-t-2:i-2]$ , 这与  $n[i-1] = t$  矛盾了, 所以有  $n[i] = t+1$ 。

2、若  $s[i-1] \neq s[t]$ , 根据 next 数组的定义,  $s[i-1]$  要继续和  $s[n[t]]$  进行比较, 如果不等, 则继续和  $s[n[n[t]]]$  进行比较, 如此这样下去, 直到与  $s[i-1] = s[n^k[t]]$  相等为止, 相等后,  $n[i] = n^k[t] + 1$ ; 若一直都不等, 则  $n[i] = n[0] + 1 = -1 + 1 = 0$ , 也是对的。证明与 1 类似。代码如下:

```
public String longestPrefix(String ss) {
    int n = ss.length();
    char [] s = ss.toCharArray();
    int [] lps = new int [n];
    for (int i = 1, j = 0; i < n; i++) {
        while (j > 0 && s[i] != s[j])
            j = lps[j-1];
        if (s[i] == s[j])
            lps[i] = ++j;
    }
    return ss.substring(0, lps[n-1]);
}
```

##### 2. rolling hash

Time complexity: O(n) / worst case: O(n^2)

Space complexity: O(1)

```
public String longestPrefix(String s) { // 容易出错, KMP 写起来比较简单
    int n = s.length(), hashPre = 0, hashSuf = 0;
    int left = 0, right = n-1, pow = 1, maxLen = 0;
    String res = "";
    while (left < n-1) {
        hashPre = hashPre * 31 + s.charAt(left);
```

```

        hashSuf = hashSuf + s.charAt(right)*pow;
        if (hashPre == hashSuf) maxLen = left + 1;
        left++;
        right--;
        pow *= 31;
    }
    return maxLen == 0 ? "" : s.substring(0, maxLen);
}

```

### 11.2.2 1910. Remove All Occurrences of a Substring - Medium 可用 KMP 算法

Given two strings  $s$  and  $part$ , perform the following operation on  $s$  until all occurrences of the substring  $part$  are removed:

Find the leftmost occurrence of the substring  $part$  and remove it from  $s$ . Return  $s$  after removing all occurrences of  $part$ .

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

```

public String removeOccurrences(String s, String part) {
    if (!s.contains(part)) return s;
    int n = s.length();
    int m = part.length();
    while (s.contains(part)) {
        int idx = s.indexOf(part);
        s = s.substring(0, idx) + (idx+m-1 == n-1 ? "" : s.substring(idx+m));
    }
    return s;
}

```

- 有人用了 KMP: 字符串匹配可以用 KMP 算法, 由于  $p$  始终不变, 可以先算一下  $p$  的 next 数组, 然后每次从  $s$  中找  $p$  的第一次出现, 删去之, 再重复进行这个过程

```

private int[] buildNext(String s) { // 找与每个位置字符不同的下一个字母的 idx
    int[] next = new int[s.length()];
    for (int i = 0, j = next[0] = -1; i < s.length()-1; ) {
        if (j == -1 || s.charAt(i) == s.charAt(j)) {
            i++;
            j++;
            next[i] = s.charAt(i) != s.charAt(j) ? j : next[j];
        } else j = next[j];
    }
    return next;
}
private int kmp(String s, String p, int[] next) { // 像是夹生饭, 半生不熟的
    for (int i = 0, j = 0; i < s.length(); ) {
        if (j == -1 || s.charAt(i) == p.charAt(j)) {
            i++;
            j++;
        } else j = next[j];
        if (j == p.length()) return i-j;
    }
    return -1;
}
public String removeOccurrences(String s, String part) {
    int[] next = buildNext(part);
    int idx = -1;
    while ((idx = kmp(s, part, next)) != -1)
        s = s.substring(0, idx) + s.substring(idx + part.length());
    return s;
}

```

### 11.2.3 1316. Distinct Echo Substrings - Hard

Return the number of distinct non-empty substrings of  $text$  that can be written as the concatenation of some string with itself (i.e. it can be written as  $a + a$  where  $a$  is some string).

- 解题思路与分析: 枚举我们在  $text$  中枚举位置  $i$  和  $j$ , 若字符串  $text[i:j]$  和  $text[j:j*2-i]$  相等, 那么字符串  $text[i:j*2-i]$  就是一个满足条件的子串, 其中  $text[x:y]$  表示字符串  $text$  中以位置  $x$  开始, 位置  $y$  结束并且不包含位置  $y$  的子串。

由于题目需要求出不同的子串数目, 因此我们还需要使用哈希集合 (HashSet) 对所有满足条件的子串进行去重操作。

```

public int distinctEchoSubstrings(String t) {
    int n = t.length(), j = 0;
    Set<String> ss = new HashSet<>();
    for (int d = 1; d <= n/2; d++) {
        for (int i = 0; i+d <= n-d; i++) {
            if (t.substring(i, i+d).equals(t.substring(i+d, i+d+d)))
                ss.add(t.substring(i, i+d));
        }
    }
    return ss.size();
}

```

## 复杂度分析

- 时间复杂度:  $O(N^3)$ , 其中  $N$  是字符串 `text` 的长度。我们需要两重循环枚举位置 `i` 和 `j`, 时间复杂度为  $O(N^2)$ , 而在比较字符串 `text[i:j]` 和 `text[j:j*2-i]` 时, 最坏的时间复杂度为  $O(N)$ , 并且将字符串加入哈希集合中的时间复杂度也为  $O(N)$ , 因此总时间复杂度为  $O(N^3)$ 。但由于本题的测试数据较弱, 它可以在规定时间内通过所有的测试数据。
- 空间复杂度:  $O(N^2)$  或  $O(N^3)$ , 在最坏情况下, 哈希集合中会存储  $O(N^2)$  个字符串, 如果我们在哈希集合中存储的是字符串视图 (例如 C++ 中的 `std::string_view`), 那么每个字符串视图使用的空间为  $O(1)$ , 总空间复杂度为  $O(N^2)$ ; 如果我们在哈希集合中存储的是字符串本身 (例如 C++ 中的 `std::string` 和 Python3 中的 `str`), 那么每个字符串使用的空间为  $O(N)$ , 总空间复杂度为  $O(N^3)$ 。

## 2. 解题思路与分析: 滚动哈希 + 前缀和

本方法需要一些关于「滚动哈希」或「Rabin-Karp 算法」的预备知识, 其核心是将字符串看成一个  $k$  进制的整数, 其中  $k$  是字符串中可能出现的字符种类, 本题中字符串只包含小写字母, 即  $k = 26$  (也可以取比  $k$  大的整数, 一般来说可以取一个质数, 例如 29 或 31)。这样做的好处是绕开了字符串操作, 将字符串看成整数进行比较, 并可以在常数时间内将字符串加入哈希集合中。

关于「滚动哈希」或「Rabin-Karp 算法」的知识, 可以参考 1044. 最长重复子串的官方题解或使用搜索引擎, 这里对算法本身的流程不再赘述。

## 使用滚动哈希

我们仍然在 `text` 中枚举位置 `i` 和 `j`，并判断字符串 `text[i:j]` 和 `text[j:j*2-i]` 是否相等。但与方法一不同的是，我们比较这两个字符串的哈希值，而并非字符串本身。如果仅使用 Rabin-Karp 算法计算这两个字符串的哈希值，我们仍然需要使用  $O(N)$  的时间，与直接比较字符串的时间复杂度相同。那么我们如何在更短的时间内计算出字符串的哈希值呢？

我们可以使用类似前缀和的方法。记 `prefix[i]` 表示字符串 `text[0:i]` 的哈希值。特别地，`prefix[0]` 的值为 `0`，为空串的哈希值。我们可以在  $O(N)$  的时间内计算出数组 `prefix` 中的所有元素，即：

$$\text{prefix}[i] = \text{prefix}[i - 1] * k + \text{text}[i]$$

当我们得到了前缀和数组 `prefix` 之后，如何计算任意字符串 `text[i:j]` 的哈希值呢？观察 `prefix[i]` 和 `prefix[j]` 这两项，它们的表达式为：

$$\begin{aligned}\text{prefix}[i] &= \text{text}[0] * k^{i-1} + \text{text}[1] * k^{i-2} + \cdots + \text{text}[i-1] \\ \text{prefix}[j] &= \text{text}[0] * k^{j-1} + \text{text}[1] * k^{j-2} + \cdots + \text{text}[j-1]\end{aligned}$$

如果将 `prefix[i]` 乘以  $k^{j-i}$ ，那么等式两侧会变为：

$$k^{j-i} * \text{prefix}[i] = \text{text}[0] * k^{j-1} + \text{text}[1] * k^{j-2} + \cdots + \text{text}[i-1] * k^{j-i}$$

将上式与 `prefix[j]` 的表达式相减，得到：

$$\text{prefix}[j] - k^{j-i} * \text{prefix}[i] = \text{text}[i] * k^{j-i-1} + \cdots + \text{text}[j-1]$$

与 `text[i:j]` 的哈希值相等，因此我们可以在  $O(1)$  的时间计算出任意字符串 `text[i:j]` 的哈希值。在此之前，我们还需要预处理计算出所有 `k` 的次幂，不然在进行乘法操作时，仍然需要超过  $O(1)$  的时间。

注意：上面的所有等式都省略了取模操作（如果你不知道为什么要取模，那么你对「Rabin-Karp 算法」的预备知识还没有掌握透彻），但在实际的代码编写中不能忽略，并且在取模的意义下，做减法时会产生负数。

通用的写法如下所示，它同时考虑了取模和负数（在 C++ 等语言中，还需要注意乘法可能产生的溢出）：

$$\text{hash\_value}(\text{text}[i:j]) = (\text{prefix}[j] - k^{j-i} * \text{prefix}[i] \% \text{mod} + \text{mod}) \% \text{mod}$$

回到我们原来的问题，当我们用字符串 `text[i:j]` 和 `text[j:j*2-i]` 的哈希值代替其本身判断是否相等后，如果两者相等，字符串 `text[i:j*2-i]` 就是一个满足条件的子串。但我们还需要进行去重操作，才能得到最终的答案。

在「判断」这一步中，由于我们只对两个字符串进行比较，因此引入哈希冲突（在下方的注意事项中也有提及）的概率极小。然而在「去重」这一步中，最坏情况下字符串的数量为  $O(N^2)$ ，大量的字符串造成哈希冲突的概率极大。为了减少字符串的数量以降低冲突的概率，我们可以使用 `N` 个哈希集合，分别存放不同长度的字符串，即第 `m` 个哈希集合存放长度为 `m + 1` 的字符串的哈希值。这样每个哈希集合只对某一固定长度的字符串进行去重操作，并且其中最多只有 `N` 个字符串，冲突概率非常低。

```
static final int mod = (int)1e9 + 7;
public int distinctEchoSubstrings(String t) {
    int n = t.length(), ans = 0;
    char[] s = t.toCharArray();
    int base = 31;
    int[] pre = new int[n+1], mul = new int[n+1];
    mul[0] = 1;
    for (int i = 1; i <= n; i++) {
        pre[i] = (int)((long)pre[i-1] * base + s[i-1]) % mod;
        mul[i] = (int)((long)mul[i-1] * base % mod);
    }
    Set<Integer>[] vis = new HashSet[n];
    for (int i = 0; i < n; i++) vis[i] = new HashSet<>();
```

```

    for (int i = 0; i < n; i++) {
        for (int j = i+1; j < n; j++) {
            int l = j - i;
            if (j + l <= n) {
                int hash_left = gethash(pre, mul, i, j-1);
                if (!vis[l-1].contains(hash_left) && hash_left == gethash(pre, mul, j, j+l-1)) {
                    ++ans;
                    vis[l-1].add(hash_left);
                }
            }
        }
    }
    return ans;
}
private int gethash(int [] pre, int [] mul, int l, int r) {
    return (int)((pre[r+l] - (long)pre[l] * mul[r-l+1] % mod + mod) % mod);
}

```

- 注意事项

由于 Rabin-Karp 算法会将字符串对应的整数值进行取模，那么：

如果字符串 S1 和 S2 对应的整数值 I1 和 I2 不相等，那么 S1 和 S2 一定不相等；

如果字符串 S1 和 S2 对应的整数值 I1 和 I2 相等，并不代表 S1 和 S2 一定相等；

这与实际应用中使用的哈希算法也是一致的，即先判断两个实例的哈希值是否相等，再判断它们本质上是否相等。而在竞赛题目中，由于数据量较少，几乎不会产生哈希冲突，因此我们可以直接用 I1 和 I2 的相等代替 S1 和 S2 的相等，减少时间复杂度。但需要牢记在实际应用中，这样做是不严谨的。

#### 11.2.4 467. Unique Substrings in Wraparound String - Medium Rabin-Karp Rolling Hash 算法

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;
}

```

#### 11.2.5 1044. Longest Duplicate Substring - Hard

Given a string s, consider all duplicated substrings: (contiguous) substrings of s that occur 2 or more times. The occurrences may overlap.

Return any duplicated substring that has the longest possible length. If s does not have a duplicated substring, the answer is "".

```

public static int base = 26; // 256
public static int mod = (1 << 31) - 1;
public static boolean match(String str1, String str2) {
    assert str1.length() == str2.length();
    for (int i = 0; i < str1.length(); i++)
        if (str1.charAt(i) != str2.charAt(i))

```

```

        return false;
    return true;
}

private String search(String s, int v) { // v: substring length
    int n = s.length();
    long hash = 0L, mp = 1L; // to avoid overflow, long long long
    Map<Long, List<Integer>> map = new HashMap<>();
    for (int j = 0; j < v; j++) {
        hash = ((hash * base) + s.charAt(j)) % mod;
        if (j >= 1)
            mp = mp * base % mod; // 先乘好准备好了，准备着备用
    }
    map.computeIfAbsent(hash, k->new ArrayList<>()).add(0);
    for (int i = 1; i+v <= n; i++) {
        hash = ((hash - s.charAt(i-1) * mp % mod + mod) % mod * base % mod + s.charAt(i+v-1)) % mod; // mod
        if (map.containsKey(hash))
            for (int idx : map.get(hash))
                if (match(s.substring(i, i+v), s.substring(idx, idx+v)))
                    return s.substring(i, i+v);
        map.computeIfAbsent(hash, k->new ArrayList<>()).add(i);
    }
    return null;
}

public String longestDupSubstring(String s) {
    int n = s.length();
    int l = 0, r = n;
    String res = "";
    while (l <= r) {
        int m = l + (r-l) / 2;
        String tmp = search(s, m);
        if (tmp == null) r = m-1;
        else {
            if (tmp.length() > res.length())
                res = tmp;
            l = m+1;
        }
    }
    return res;
}
}

```

## 11.2.6 1156. Swap For Longest Repeated Character Substring - Medium

You are given a string `text`. You can swap two of the characters in the text.

Return the length of the longest substring with repeated characters.

给你一个字符串，如何找最长的重复子串，博主会数连续相同的字符，若此时有一个不同字符出现了，只要后面还有相同的字符，就会继续数下去，因为有一次交换的机会，什么时候停止呢，当再次出现不同字符的时候就停止，或者是当前统计个数等于该字符出现的总个数时也停止，因为得到的结果不可能超过某个字符出现的总个数。所以可以先统计每个字符的出现次数，然后开始遍历字符，对于每个遍历到的字符，都开始数之后跟其相等的字符，新建变量 `j`, `cnt`, 和 `diff`, 当 `j` 小于 `n`, 且当前字符和比较字符相同，或者 `diff` 等于 0，且 `cnt` 小于比较字符出现的总个数时进行遍历，若当前遍历到的字符和要比较的字符不相等，说明该使用交换操作了，`diff` 自增 1，此时将 `i` 更新为 `j-1`，这是一个优化操作，可以避免一些不必要的计算，下次从这个位置往后统计，也相当于重置了 `diff`。还有就是这个 `cnt` 小于字符出现总个数这个条件卡的非常好，即便下一个还是相同字符，也不能再统计了，因为最后的这个相同字符可能是用来交换前面的断点位置的。每次用统计出来的 `cnt` 更新结果 `res`，但是一个方向的遍历可能无法应对所有情况，比如“`acbaaa`”，若只是从前往后遍历，那么最终只能得到 3，而正确的答案是 4，因为可以将 `b` 和第一个 `a` 交换，所以还需要从后往前进行一次相同的操作，这样才能得到正确的答案，参见代码如下：

```

public int maxRepOpt1(String s) { // O(n^2)
    int n = s.length(), ans = 0;
    Map<Character, Integer> charCnt = new HashMap<>();
    for (char c : s.toCharArray())
        charCnt.put(c, charCnt.getOrDefault(c, 0) + 1);
    for (int i = 0; i < n; i++) {
        char cur = s.charAt(i);
        int j = i, cnt = 0, dif = 0;
        while (j < n && (cur == s.charAt(j) || dif == 0) && cnt < charCnt.get(cur)) {
            if (cur != s.charAt(j)) {
                ++dif;
                i = j-1; // exchanged once, i moves to be the repeated sequence tail
            }
            ++cnt;
            ++j;
        }
        ans = Math.max(ans, cnt);
    }
    for (int i = n-1; i >= 0; i--) {
        char cur = s.charAt(i);
        int j = i, cnt = 0, dif = 0;
        while (j >= 0 && (cur == s.charAt(j) || dif == 0) && cnt < charCnt.get(cur)) {
            if (cur != s.charAt(j)) {
                ++dif;
                i = j+1;
            }
            ++cnt;
            --j;
        }
        ans = Math.max(ans, cnt);
    }
}

```

```

        }
        ++cnt;
        --j;
    }
    ans = Math.max(ans, cnt);
}
return res;
}

```

- O(N) 解法

上面的解法严格来说还是平方级的，再来看一种线性时间的解法，可能比较难想，由于这里需要关注的是相同字符的出现位置，所以可以将所有相同的字符的位置都放到一个数组中，那么这里就建立一个字符和其出现位置数组之间的映射。由于题目中限制了只有英文字母，所以可以按照每个字母进行遍历，直接遍历每个字符的位置数组，这里新建变量 `cnt`, `cnt2`, 和 `mx`, 其中 `cnt` 统计的是连续字母的个数，`cnt2` 相当于一个临时变量，当使用交换操作时，保存之前的 `cnt` 值，`mx` 为二者之和。在遍历完某个字母位置数组之后，最后看一下若该字母出现总个数大于 `mx`，则说明交换后的字母还没有统计进去，不管之前有没有使用交换操作，都需要加上这个额外的一个，参见代码如下：

```

public int maxRepOpt1(String s) { // O(n^2)
    int n = s.length(), ans = 0;
    Map<Character, List<Integer>> idxMap = new HashMap<>();
    for (int i = 0; i < n; i++)
        idxMap.computeIfAbsent(s.charAt(i), k -> new ArrayList<>()).add(i);
    for (char c = 'a'; c <= 'z'; c++) {
        if (!idxMap.containsKey(c)) continue;
        int cnt = 1, cntb = 0, max = 0;
        List<Integer> idxs = idxMap.get(c);
        for (int i = 1; i < idxs.size(); i++) {
            if (idxs.get(i) == idxs.get(i-1) + 1) // aa
                ++cnt;
            else {
                cntb = (idxs.get(i) == idxs.get(i-1) + 2) ? cnt : 0; // aba ?
                cnt = 1;
            }
            max = Math.max(max, cnt + cntb);
        }
        ans = Math.max(ans, max + (idxs.size() > max ? 1 : 0)); // aaaaabaaaaaca 多于两个重复子段，中间替换字符可是是相同的
    }
    return ans;
}

```

## 算法

(动态规划)  $O(n)$

1. 我们分别考察每种字符所能得到的最大答案。
2. 假设我们当前考察的是字符 `c`。我们通过遍历一次数组的方式，找到单字符就是 `c` 的情况下的最长单重复子串。
3. 设 `cnt` 为字符串中 `c` 出现的次数；`f(i)` 表示以 `i` 结尾且没有替换过不是 `c` 的字符时的最长但重复子串；`g(i)` 表示以 `i` 结尾且替换过不是 `c` 的字符时的最长但重复子串。
4. 当 `c == text[i]` 时，累加 `cnt`，`f(i) = f(i - 1) + 1`, `g(i) = g(i - 1) + 1`，也就是都可以沿着上一次的结果往后累加 1 的长度。
5. 如果 `c != text[i]`，则 `g(i) = f(i - 1) + 1`，表示我们强制替换 `text[i]` 的字符为 `c`（注意不是交换），所以要从没有替换过的 `f` 数组转移；`f(i) = 0`，表示如果不替换，那么长度只能归 0。交换只是替换的一种特殊形式，我们都先假设总能交换。如果不能交换，如 `aaabaaa` 我们无法把中间的 `b` 用一个新的 `a` 交换时，可以通过答案不能超过 `cnt` 来特判。
6. 答案为  $\min(cnt, \max(f(i), g(i)))$ ，也就是所有的 `f` 和 `g` 的最大值，但结果不能超过 `cnt`。
7. 这里的 `f` 和 `g` 数组因为之和上一位有关，所以可以化简为变量。

## 时间复杂度

- 枚举每种字符时，只需要遍历数组一次，故时间复杂度为  $O(n)$ 。

## 空间复杂度

- 动态规划    • 优化后，只需要常数个变量，故空间复杂度为  $O(1)$ 。

```

private int solve(char c, String s) {
    int n = s.length(), max = 0;
    int f = 0, g = 0, cnt = 0;
    for (int i = 0; i < n; i++) {
        if (c == s.charAt(i)) {

```

```

        f++;
        g++;
        cnt++;
    } else {
        g = f + 1;
        f = 0;
    }
    max = Math.max(max, Math.max(f, g));
}
return Math.min(max, cnt);
}
public int maxRepOpt1(String s) {
    int n = s.length(), ans = 0;
    for (char i = 'a'; i <= 'z'; i++)
        ans = Math.max(ans, solve(i, s));
    return ans;
}

```

### 11.2.7 395. Longest Substring with At Least K Repeating Characters - Medium

Given a string  $s$  and an integer  $k$ , return the length of the longest substring of  $s$  such that the frequency of each character in this substring is greater than or equal to  $k$ .

```

// 由于字母只有 26 个，而整型 mask 有 32 位，足够用了,
// 每一位代表一个字母，如果为 1，表示该字母不够 k 次，如果为 0 就表示已经出现了 k 次，这种思路真是太聪明了,
// 隐约记得这种用法在之前的题目中也用过，但是博主并不能举一反三（沮丧脸:(），还得继续努力啊。
// 遍历字符串，对于每一个字符，都将其视为起点，然后遍历到末尾，增加 HashMap 中字母的出现次数，如果其小于 k，将 mask 的对应位改为 1，如果大于等于 k，将 mask 对应位改为 0。然后看 mask 是否为 0，是的话就更新 res 结果，然后把当前满足要求的子字符串的起始位置 j 保存到 max_idx 中，等内层循环结束后，将外层循环变量 i 赋值为 max_idx+1，继续遍历。
public int longestSubstring(String s, int k) { // O(N^2)
    int n = s.length(), res = 0, i = 0;
    while (i + k <= n) {
        int[] m = new int[26];
        int mask = 0, maxIdx = i;
        for (int j = i; j < n; j++) {
            int t = s.charAt(j) - 'a';
            m[t]++;
            if (m[t] < k) mask |= (1 << t);
            else mask &= (~(1 << t));
            if (mask == 0) {
                res = Math.max(res, j - i + 1);
                maxIdx = j;
            }
        }
        i = maxIdx + 1;
    }
    return res;
}

```

- 双指针 sliding window O(N)

```

public int longestSubstring(String s, int k) {
    int n = s.length(), res = 0;
    for (int cnt = 1; cnt <= 26; cnt++) {
        int start = 0, i = 0, uniqueCnt = 0;
        int[] charCntr = new int[26];
        while (i < n) {
            boolean valid = true;
            if (charCntr[s.charAt(i++) - 'a']++ == 0) ++ uniqueCnt;
            while (uniqueCnt > cnt) {
                if (--charCntr[s.charAt(start++) - 'a'] == 0) --uniqueCnt;
                for (int j = 0; j < 26; j++)
                    if (charCntr[j] > 0 && charCntr[j] < k) valid = false;
                if (valid) res = Math.max(res, i - start);
            }
        }
    }
    return res;
}

```

- 分治：分而治之

```

public int longestSubstring(String s, int k) { // str.split("[dkfldjf]")
    int n = s.length();
    if (n < k) return 0;
    if (n == k && s.chars().distinct().count() == 1) return k;
    int[] cnt = new int[26];
    for (int i = 0; i < n; i++)
        cnt[s.charAt(i) - 'a']++;
    if (Arrays.stream(cnt).max().getAsInt() < k) return 0;
    StringBuilder sb = new StringBuilder("[");
    for (int i = 0; i < 26; i++)
        if (cnt[i] < k && cnt[i] != 0)
            sb.append((char)(i + 'a'));
    sb.append(']');
    if (sb.length() == 2) return n;
    String[] sa = s.split(sb.toString()); // str.split("[-*/=]") pay attention to the format

```

```

System.out.println(Arrays.toString(sa));
int max = 0;
for (int i = 0; i < sa.length; i++)
    max = Math.max(max, longestSubstring(sa[i], k));
return max;
}
public int longestSubstring(String s, int k) { // 人工手动拆分
    int n = s.length(), maxIdx = 0, res = 0;
    int[] cnt = new int[128];
    boolean valid = true;
    for (char c : s.toCharArray())
        cnt[c]++;
    for (int i = 0; i < n; i++) {
        if (cnt[s.charAt(i)] < k) {
            res = Math.max(res, longestSubstring(s.substring(maxIdx, i), k));
            valid = false;
            maxIdx = i+1;
        }
    }
    return valid ? n : Math.max(res, longestSubstring(s.substring(maxIdx, n), k));
}

```

### 11.2.8 1830. Minimum Number of Operations to Make String Sorted - Hard 排列组合费小马快速幂

You are given a string  $s$  (0-indexed). You are asked to perform the following operation on  $s$  until you get a sorted string:

Find the largest index  $i$  such that  $1 \leq i < s.length$  and  $s[i] < s[i - 1]$ . Find the largest index  $j$  such that  $i \leq j < s.length$  and  $s[k] < s[i - 1]$  for all the possible values of  $k$  in the range  $[i, j]$  inclusive. Swap the two characters at indices  $i - 1$  and  $j$ . Reverse the suffix starting at index  $i$ . Return the number of operations needed to make the string sorted. Since the answer can be too large, return it modulo  $10^9 + 7$ .

- 解题思路与分析 题中每次对字符串  $s$  执行的操作，是将其变为由当前字母组成的前一字典序的字符串。因此求最少操作次数，等价于求解该字符串在由当前字母组成的所有排列中的字典序；

求比当前字符串  $s$  小的排列个数，可通过排列组合公式计算得到；

排列组合公式中的阶乘逆元取模，可通过费马小定理，转化为对模数的乘方进行计算；

可通过快速乘方算法，进一步提高对乘方的计算效率。

```

private int quickmul(int base, int exp) { // 快速乘方算法
    long ans = 1L;
    while (exp > 0) {
        if ((exp & 1) == 1) // 指数是奇数，就先乘一次 base
            ans = ans * base % mod;
        base = (int)((long)base * base % mod); // 底平方（指数变偶数之后）
        exp >>= 1; // 指数除 2，快速计算
    }
    return (int)ans;
}
int mod = (int)1e9 + 7;
public int makeStringSorted(String t) {
    int n = t.length();
    char[] s = t.toCharArray();
    int[] cnt = new int[26]; // 记录剩余字符串中各字母个数
    Arrays.fill(cnt, 0);
    for (int i = 0; i < n; i++)
        cnt[s[i] - 'a']++;
    int[] fact = new int[n+1];
    int[] finv = new int[n+1];
    fact[0] = 1;
    finv[0] = 1;
    for (int i = 1; i <= n; i++) {
        fact[i] = (int)((long)fact[i-1] * i % mod); // fact[i] = i! % mod
        finv[i] = quickmul(fact[i], mod - 2); // 费马小定理计算乘法逆元，finv[i] = (i!) ^ -1 % mod
    }
    long ans = 0L;
    for (int i = 0; i < n-1; i++) {
        int lessCnt = 0; // 比当前位置小的字母总数
        for (int j = 0; j < s[i] - 'a'; j++)
            lessCnt += cnt[j];
        long upper = (long)lessCnt * fact[n-1-i] % mod; // 排列公式分子
        for (int j = 0; j < 26; j++)
            upper = upper * finv[cnt[j]] % mod;
        ans = (ans + upper) % mod;
        cnt[s[i] - 'a']--; // 指针右移
    }
    return (int)ans;
}

```

### 11.2.9 1960. Maximum Product of the Length of Two Palindromic Substrings - Hard 马拉车算法 todo

You are given a 0-indexed string  $s$  and are tasked with finding two non-intersecting palindromic substrings of odd length such that the product of their lengths is maximized.

More formally, you want to choose four integers  $i, j, k, l$  such that  $0 \leq i \leq j < k \leq l < s.length$  and both the substrings  $s[i\dots j]$  and  $s[k\dots l]$  are palindromes and have odd lengths.  $s[i\dots j]$  denotes a substring from index  $i$  to index  $j$  inclusive.

Return the maximum possible product of the lengths of the two non-intersecting palindromic substrings.

A palindrome is a string that is the same forward and backward. A substring is a contiguous sequence of characters in a string.

#### 1. 解题思路与分析: rolling hash 算法

```
public long maxProduct(String t) { // rolling hash 算法: Time complexity - O(n), space complexity - O(n). NO collision detection
    int n = t.length();
    char [] s = t.toCharArray();
    long [] p = new long[n + 1]; // pow[i] = BASE^i
    p[0] = 1;
    for (int i = 1; i <= n; i++) p[i] = p[i-1] * base % mod;
    long [] lh = new long [n+1], rh = new long [n+1]; // 从左向右、从右向左遍历的 hash 值
    for (int i = 1; i <= n; i++) // s[0, i]: 从左向右遍历
        lh[i] = (lh[i-1] * base + s[i-1] - 'a') % mod;
    for (int i = n-1; i >= 0; i--) // s(i, n-1): 从右向左遍历
        rh[i] = (rh[i+1] * base + s[i] - 'a') % mod;
    int left [] = new int [n], right [] = new int [n];
    for (int i = 0, max = 1; i < n; i++) { // 以 i 为 右 端点, 回文片段在其左侧的最大回文长度
        if (max < i && isPalindromic(i-max-1, i+1, lh, rh, p)) max += 2;
        left[i] = max;
    }
    for (int i = n-1, max = 1; i >= 0; i--) { // 以 i 为 左 端点, 回文片段在其右侧的最大回文长度
        if (i + max + 2 <= n && isPalindromic(i, i+max+2, lh, rh, p)) max += 2;
        right[i] = max;
    }
    long ans = 1;
    for (int i = 1; i < n; i++)
        ans = Math.max(ans, (long)left[i-1] * right[i]);
    return ans;
}
static final long mod = Integer.MAX_VALUE;
static final long base = 29L;
static boolean isPalindromic(int l, int r, long [] left, long [] right, long [] p) {
    return lh(l, r, left, p) == rh(l, r, right, p);
}
static long lh(int l, int r, long [] hash, long [] p) {
    long ans = (hash[r] - hash[l] * p[r-l]) % mod;
    if (ans < 0) ans += mod;
    return ans;
}
static long rh(int l, int r, long [] hash, long [] p) {
    long ans = (hash[l] - hash[r] * p[r-l]) % mod;
    if (ans < 0) ans += mod;
    return ans;
}
```

#### 2. 解题思路与分析: 马拉车

[https://leetcode.com/problems/maximum-product-of-the-length-of-two-palindromic-substrings/discuss/1393288/Java-100-O\(n\)-time-O\(n\)-space-using-Manacher's-algorithm](https://leetcode.com/problems/maximum-product-of-the-length-of-two-palindromic-substrings/discuss/1393288/Java-100-O(n)-time-O(n)-space-using-Manacher's-algorithm)

```
public long maxProduct(String s) {
    int n = s.length();
    int d [] = manacher(s); // 以 i 为中心的, 包括当前字符的单侧最大回文长度
    int l [] = new int [n], r [] = new int [n]; // 转化为: 以当前坐标为最右、最左极限的全回文片段最大长度
    for (int i = 0; i < n; i++) {
        l[i+d[i]-1] = Math.max(l[i+d[i]-1], 2 * d[i]-1); // 用第一个当前下标, 来更新所有右端所能企及、以右端点为结尾的最大片段长度, 从左向右
        r[i-d[i]+1] = 2 * d[i] - 1; // 用当前回文片段全长, 更新以左端点为起始点的最大长度, 从右向左???
    }
    // if maximum palindrome has end at i and its length is L, then there's a palindrome of length L-2 with end at position i-1
    for (int i = n-2, j = n-1; i >= 0; i--, j--) // 从右向左
        l[i] = Math.max(l[i], l[j]-2);
    for (int i = 1, j = 0; i < n; i++, j++) // 从左向右
        r[i] = Math.max(r[i], r[j]-2);
    // Same fact for beginnings of palindromes
    for (int i = 1, j = 0; i < n; i++, j++)
        l[i] = Math.max(l[i], l[j]);
    for (int i = n-2, j = n-1; i >= 0; i--, j--)
        r[i] = Math.max(r[i], r[j]);
    long ans = 1;
    for (int i = 1; i < n; i++)
        ans = Math.max(ans, (long)l[i-1] * r[i]);
    return ans;
}
```

```

static int[] manacher(String t) { // 这个写法可能并不标准: modifiedOddManacher
    int n = t.length();
    char[] s = t.toCharArray();
    int[] ans = new int[n];
    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int len = i > r ? 1 : Math.min(ans[l+r-i], r-i+1); // 起始单侧长度只有 1
        int maxLen = Math.min(i, n-1-i); // 就算两边都能对称延伸至两端点, 以现 i 为中心的回文的 单侧最大长度 (包括当前下标的字符计算在内)
        int x = i - len, y = i + len;
        while (len <= maxLen && s[x--] == s[y++]) len++; // 向两侧扩散延伸
        ans[i] = len--;
        if (i + len > r) {
            l = i - len;
            r = i + len;
        }
    }
    return ans;
}

```

### 3. 解题思路与分析: 根据题意提速版的马拉车算法

```

public long maxProduct(String s) { // 这个方法是: 目前自己能够理解的解法里最快的
    int n = s.length();
    StringScanner sb = new StringScanner(s);
    int[] l = new int[n], r = new int[n];
    modifiedOddManacher(sb.toString(), l);
    modifiedOddManacher(sb.reverse().toString(), r);
    long ans = 1;
    for (int i = 0; i < n-1; i++) {
        ans = Math.max(ans, (l[i]-1) * 2l) * (1 + (r[n-(i+1)-1]-1) * 2l));
    }
    return ans;
}
void modifiedOddManacher(String t, int[] d) {
    int n = t.length();
    char[] s = t.toCharArray();
    int[] idx = new int[n]; // idx: center
    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int radius = (i > r) ? 1 : Math.min(idx[l+(r-i)], r-i+1);
        while (i - radius >= 0 && i + radius < n && s[i-radius] == s[i+radius]) { // 因为半径没有限制, 所以要检查两边是否会出界
            d[i+radius] = radius + 1;
            radius++;
        }
        idx[i] = radius--;
        if (i + radius > r) {
            l = i - radius;
            r = i + radius;
        }
    }
    for (int i = 0, max = 1; i < n; i++) {
        max = Math.max(max, d[i]);
        d[i] = max;
    }
}
// Just compare with basic odd Manacher for better understanding.
void oddManacher(String s) { // 是需要对比理解一下
    int n = s.length();
    int[] idx = new int[n];
    for (int i = 0, l = 0, r = -1; i < n; i++) {
        int radius = (i > r) ? 1 : Math.min(idx[l+r-i], r-i+1);
        while (i - radius >= 0 && i + radius < n && s[i-radius] == s[i+radius]) radius++;
        idx[i] = radius--;
        if (i + radius > r) {
            l = i - radius;
            r = i + radius;
        }
    }
}

```

### 4. DP 解: 要再好好理解消化一下

```

public long maxProduct(String s) { // 这个方法稍慢, 但思路相对简洁
    final int n = s.length();
    long ans = 1;
    // l[i] := max length of palindromes in s[0..i]
    int[] l = manacher(s, n);
    // r[i] := max length of palindromes in s[i..n)
    int[] r = manacher(new StringScanner(s).reverse().toString(), n);
    reverse(r, 0, n - 1);
    for (int i = 0; i + 1 < n; ++i)
        ans = Math.max(ans, (long) l[i] * r[i + 1]);
    return ans;
}
private int[] manacher(final String s, int n) {
    int[] maxExtends = new int[n];
    int[] l2r = new int[n];
    Arrays.fill(l2r, 1);
    int center = 0;
    for (int i = 0; i < n; ++i) {
        final int r = center + maxExtends[center] - 1;

```

```
final int mirrorIndex = center - (i - center);
int extend = i > r ? 1 : Math.min(maxExtends[mirrorIndex], r - i + 1);
while (i - extend >= 0 && i + extend < n && s.charAt(i - extend) == s.charAt(i + extend)) {
    l2r[i + extend] = 2 * extend + 1;
    ++extend;
}
maxExtends[i] = extend;
if (i + maxExtends[i] >= r)
    center = i;
}
for (int i = 1; i < n; ++i)
    l2r[i] = Math.max(l2r[i], l2r[i - 1]);
return l2r;
}
private void reverse(int[] A, int l, int r) {
    while (l < r)
        swap(A, l++, r--);
}
private void swap(int[] A, int i, int j) {
    final int temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}
```

# Chapter 12

## PreSum 差分数组

### 12.1 1674. Minimum Moves to Make Array Complementary - Medium 差分数组

You are given an integer array `nums` of even length  $n$  and an integer `limit`. In one move, you can replace any integer from `nums` with another integer between 1 and `limit`, inclusive.

The array `nums` is complementary if for all indices  $i$  (0-indexed),  $\text{nums}[i] + \text{nums}[n - 1 - i]$  equals the same number. For example, the array [1,2,3,4] is complementary because for all indices  $i$ ,  $\text{nums}[i] + \text{nums}[n - 1 - i] = 5$ .

Return the minimum number of moves required to make `nums` complementary.

1. 解题思路与分析 Sweep Line / Prefix Sum Let  $a = \min(\text{nums}[i], \text{nums}[n-i-1])$ ,  $b = \max(\text{nums}[i], \text{nums}[n-i-1])$

The key to this problem is how many moves do we need to make  $a + b == T$ .

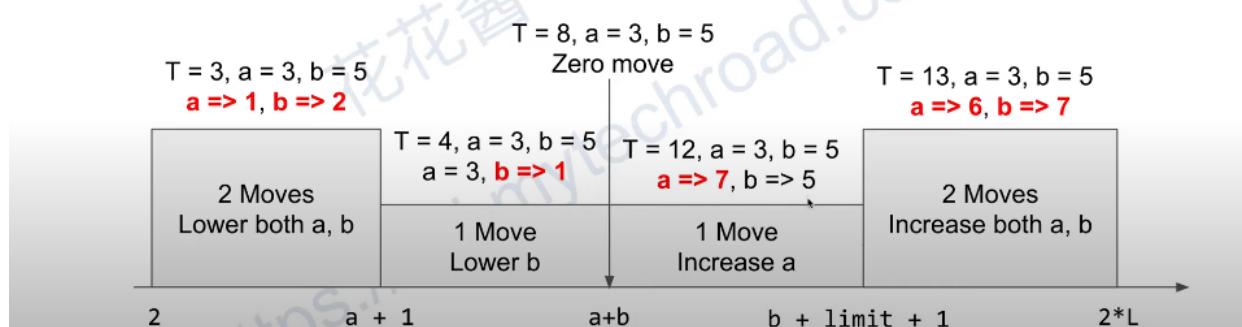
```
if 2 <= T < a + 1, two moves, lower both a and b.  
if a + 1 <= T < a + b, one move, lower b  
if a + b == T, zero move  
if a + b + 1 <= T < b + limit + 1, one move, increase a  
if b + limit + 1 <= T <= 2*limit, two moves, increase both a and b.
```

- 关键是找到五个关键点，并利用扫描线、差分数组来暴搜所有数对和，并最全局最优解

To better understand this problem, we can start with the brute force method:  
Try all possible pair sum  $T$  from 2 to  $\text{limit} * 2$ , and check how many moves we need.

Let  $a = \min(\text{nums}[i], \text{nums}[n - i - 1])$ ,  $b = \max(\text{nums}[i], \text{nums}[n - i - 1])$   
*How many moves do we need to sum up to  $T$ ? It depends on  $(a, b, T, \text{limit})$*

Time complexity:  $O(\text{limit} * n)$  TLE 72/110 passed, Space complexity:  $O(1)$



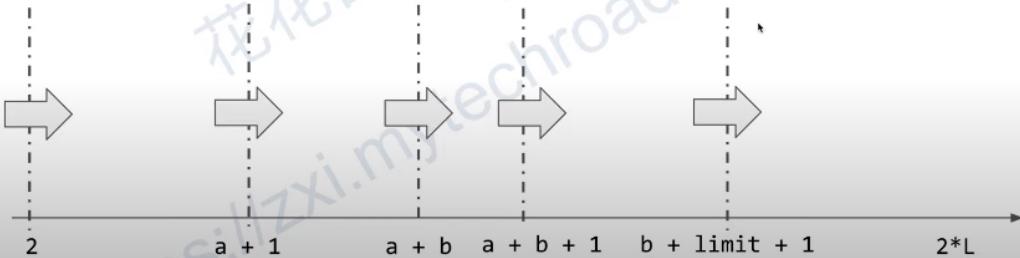
Optimization: If we move T in the x-axis, whenever T cross one boundary, we need to perform some action.

We can record the delta for each pair at each boundary.

Then we only need to do a prefix sum to determine how many moves we need at the current pair sum T in O(1). Similar to LeetCode 731. My Calendar II.

Time complexity: O(n + limit), Space complexity: O(limit)

+2 (lower a, lower b)   -1 (lower a)   -1 (no-op)   +1 (increase a)   +1 (increase a, increase b)



- 用一个实例来看。这里强调一下：不可以使用二分搜索，因为这里的答案并不唯一。

a, b	2 / +2	a+1 / -1	a+b / -1	a+b+1 / +1	a+b+L+1 / +1	delta
1, 3	2 / +2	2 / -1	4 / -1	5 / +1	9 / +1	{2:1, 4:-1, 5:1, 9:1}
2, 4	2 / +2	3 / -1	6 / -1	7 / +1	11 / +1	{2:3, 3:-1, 4:-1, 5:1, 6:-1, 7:1, 9:1, 11:1}

T	delta	Prefix sum / ans
2	3	3
3	-1	2
4	-1	1
5	1	2
6	-1	1
7	1	2
9	1	3
11	1	4

[1, 2, 2, 3]

[3, 2, 4, 3]

Since there are at most O(n) unique boundaries, we can sort them to obtain O(nlogn) time complexity and O(n) space complexity. If limit >> n.

```
public int minMoves(int[] nums, int limit) {
    int n = nums.length;
    int [] delta = new int [2 * limit + 2]; // 差分数组
    for (int i = 0; i < n/2; i++) {
        int a = Math.min(nums[i], nums[n-1-i]);
        int b = Math.max(nums[i], nums[n-1-i]); // 把各区间所需要的操作简化到五行代码中去了
        delta[2] += 2; // dec a, decreasing b [2, a] 区间的和 所需要的变换次数都是 2 次
        --delta[a+1]; // dec a [a + 1, a+b-1] 区间中的和 所需要变的次数都是 1 次
        --delta[a+b]; // no operations
        ++delta[a+b+1]; // inc a [a+b+1, b+limit] 区间中的和 只需要变动 1 次;
        ++delta[b+limit+1]; // inc a, inc b
    }
    int ans = n, sum = 0;
    for (int t = 2; t < 2 * limit + 2; t++) {
        sum += delta[t]; // 差分数组的前缀和等于 (共需要的操作次数)
        ans = Math.min(ans, sum);
    }
    return ans;
}
```

## 12.2 798. Smallest Rotation with Highest Score

Hard

308

19

Add to List

**Share** You are given an array `nums`. You can rotate it by a non-negative integer `k` so that the array becomes `[nums[k], nums[k + 1], ... , nums[nums.length - 1], nums1, nums2, ..., nums[k-1]]`. Afterward, any entries that are less than or equal to their index are worth one point.

For example, if we have `nums = [2,4,1,3,0]`, and we rotate by `k = 2`, it becomes `[1,3,0,2,4]`. This is worth 3 points because  $1 > 0$  [no points],  $3 > 1$  [no points],  $0 \leq 2$  [one point],  $2 \leq 3$  [one point],  $4 \leq 4$  [one point]. Return the rotation index `k` that corresponds to the highest score we can achieve if we rotated `nums` by it. If there are multiple answers, return the smallest such index `k`.

### 1. 解题思路与分析

答案的思路也十分巧妙，并没有采用 **brute force** 那种直接求每一个 `K` 值的得分，而是反其道而行之，对于每个数字，探究其跟 `K` 值之间的联系。首先我们要讨论一下边界情况，那么就是当 `A[i] = 0` 或 `N` 的情况，首先如果 `A[i] = 0` 的话，那么 0 这个数字在任何位置都会小于等于坐标值，所以在任何位置都会得分的，那么其实可以忽略之，因为其不会对最大值产生任何影响，同理，如果 `A[i] = N` 的时候，由于长度为 `N` 的数组的坐标值范围是 `[0, N-1]`，所以数字 `N` 在任何位置都不得分，同样也不会对最大值产生任何影响，可以忽略之。那么我们关心的数字的范围其实是 `[1, N-1]`。在这个范围内的数字在旋转数组的过程中，从位置 0 变到 `N-1` 位置的时候，一定会得分，因为此范围的数字最大就是 `N-1`。这个一定得的分我们在最后统一加上，基于上面的发现，我们再来分析下题目中的例子 `[2, 3, 1, 4, 0]`，其中红色数字表示不得分的位置：

```

A: 2 3 1 4 0 (K = 0)
A: 3 1 4 0 2 (K = 1)
A: 1 4 0 2 3 (K = 2)
A: 4 0 2 3 1 (K = 3)
A: 0 2 3 1 4 (K = 4)
idx: 0 1 2 3 4

```

对于某个数字 `A[i]`，我们想知道其什么时候能旋转到坐标位置为 `A[i]` 的地方，这样就可以得分了。比如上面博主标记了红色的数字 3，最开始时的位置为 1，此时是不得分的，我们想知道其什么时候能到位置 3，答案是当 `K=3` 的时候，其刚好旋转到位置 3，`K` 再增加的时候，其又开始不得分了。所以这个最后能得分的临界位置是通过  $(i - A[i] + N) \% N$  得到，那么此时如果 `K` 再增加 1 的话，`A[i]` 就开始不得分了（如果我们 suppose 每个位置都可以得分，那么不得分的地方就可以当作是失分了），所以我们可以在这个刚好开始不得分的地方标记一下，通过-1 进行标记，这个位置就是  $(i - A[i] + 1 + N) \% N$ 。我们用一个长度为 `N` 的 `change` 数组，对于每个数字，我们都找到其刚好不得分的地方，进行-1 操作，那么此时 `change[i]` 就表示数组中的数字在 `i` 位置会不得分的个数，如果我们仔细观察上面红色的数字，可以发现，由于是左移，坐标在不断减小，所以原先失分的地方，在 `K+1` 的时候还是失分，除非你从开头位置跑到末尾去了，那会得分，所以我们要累加 `change` 数组，并且 `K` 每增加 1 的时候，要加上额外的 1，最后 `change` 数组中最大数字的位置就是要求的 `K` 值了。

**Key point**

Don't calculate the score for `K=0`, we don't need it at all.

(I see almost all other solutions did)

The key point is to find out how score changes when `K++`

**Time complexity:**

"A will have length at most 20000."

I think it means you should find a  $O(N)$  solution.

**How does score change when `K++` ?**

- **Get point**

Each time when we rotate, we make index `0` to index `N-1`, then we get one more point.

We know that for sure, so I don't need to record it.

- **Lose point**

`(i - A[i] + N) % N` is the value of `K` making `A[i]`'s index just equal to `A[i]`.

For example, If `A[6] = 1`, then `K = (6 - A[6]) % 6 = 5` making `A[6]` to index `1` of new array.

So when `K=5`, we get this point for `A[6]`

Then if `K` is bigger when `K = (i - A[i] + 1) % N`, we start to lose this point, making our score  $-= 1$

All I have done is record the value of `K` for all `A[i]` where we will lose points.

- **`A[i]=0`**

Rotation makes no change for it, because we always have `0 <= index`.

However, it is covered in "get point" and "lose point".

**Explanation of codes**

1. Search the index where score decrease and record this change to a list `change`.

2. A simple for loop to calculate the score for every `K` value.

```
score[K] = score[K-1] + change[K]
```

In my codes I accumulated `changes` so I get the changed score for every `K` value compared to `K=0`

3. Find the index of best score.

时间复杂度:  $O(n)$

空间复杂度:  $O(n)$

```
public int bestRotation(int[] a) {
    int n = a.length, ans = 0;
    int [] change = new int [n];
    for (int i = 0; i < n; ++i) change[(i - a[i] + n + 1) % n] -= 1;
    for (int i = 1; i < n; ++i) {
        change[i] += change[i-1] + 1;
        ans = change[i] > change[ans] ? i : ans;
    }
    return ans;
}
```

### 12.3 1074. Number of Submatrices That Sum to Target - Hard 二维数组转化为一维数组: 数组中 target Sum 的 2D 版、presum 思想的 2d 版

Given a matrix and a target, return the number of non-empty submatrices that sum to target.

A submatrix  $x_1, y_1, x_2, y_2$  is the set of all cells  $\text{matrix}[x][y]$  with  $x_1 \leq x \leq x_2$  and  $y_1 \leq y \leq y_2$ .

Two submatrices  $(x_1, y_1, x_2, y_2)$  and  $(x'_1, y'_1, x'_2, y'_2)$  are different if they have some coordinate that is different: for example, if  $x_1 \neq x'_1$ .

本题也需要使用到 presum 思路来解题, 只不过这个前缀和 presum 的计算对象是二维数组。对于任意一个点, `presum[i][j]` 代表了从  $\text{Matrix}^{1,1}$  到  $\text{Matrix}[i][j]$  之间的和。

有了前缀和之后，我们可以将二维数组拆解为多个一维数组，再用一维数组的思路去解题。

拆分数组时可以以列拆分，也可以以行拆，无论如何都可以达到遍历所有子矩阵的效果。本题以列拆分为例，对于任意两列 col1 和 col2，我们可以得到所有行的前缀和

这样，二维数组就转化为了一维数组。接下来只要遍历所有列的组合即可。

```
public int numSubmatrixSumTarget(int[][] matrix, int target) {
    int m = matrix.length, n = matrix[0].length, rowSum = 0;
    int [][] sum = new int [m][n]; // sums of row sum only
    for (int i = 0; i < m; i++) {
        rowSum = 0;
        for (int j = 0; j < n; j++) {
            rowSum += matrix[i][j];
            sum[i][j] = rowSum + (i == 0 ? 0 : sum[i-1][j]);
        }
    }
    int cnt = 0, cur = 0;
    for (int j = 0; j < n; j++) // col1
        for (int k = j; k < n; k++) // col2
            Map<Integer, Integer> map = new HashMap<>(); // 用于记录前缀和个数
            for (int i = 0; i < m; i++) {
                cur = sum[i][k] - (j == 0 ? 0 : sum[i][j-1]);
                if (cur == target) ++cnt;
                cnt += map.getOrDefault(cur - target, 0);
                map.put(cur, map.getOrDefault(cur, 0) + 1);
            }
    }
    return cnt;
}
```

- 对 corner case 的处理相对简洁的写法

```
public int numSubmatrixSumTarget(int [][] matrix, int target) {
    int res = 0, m = matrix.length, n = matrix[0].length;
    int [][] sum = new int [m+1][n+1];
    for (int i = 1; i <= m; ++i)
        for (int j = 1; j <= n; ++j) // 对全数组纵横求和
            sum[i][j] = sum[i][j - 1] + sum[i - 1][j] - sum[i - 1][j - 1] + matrix[i - 1][j - 1];
    for (int i = 1; i <= m; ++i)
        for (int j = 1; j <= n; ++j)
            for (int p = 1; p <= i; ++p)
                for (int q = 1; q <= j; ++q) {
                    int t = sum[i][j] - sum[i][q - 1] - sum[p - 1][j] + sum[p - 1][q - 1];
                    if (t == target) ++res;
                }
    return res;
}

public int numSubmatrixSumTarget(int[][] matrix, int target) {
    int m = matrix.length, n = matrix[0].length;
    int [][] sum = new int [m][n+1]; // sums of row sum only
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++)
            sum[i][j+1] = sum[i][j] + matrix[i][j]; // row sums only
    }
    int cnt = 0, cur = 0;
    for (int j = 0; j < n; j++) // col1
        for (int k = j; k < n; k++) // col2
            cur = 0;
            Map<Integer, Integer> map = new HashMap<>(); // 用于记录前缀和个数
            map.put(0, 1);
            for (int i = 0; i < m; i++) {
                cur += sum[i][k+1] - sum[i][j];
                cnt += map.getOrDefault(cur - target, 0);
                map.put(cur, map.getOrDefault(cur, 0) + 1);
            }
    }
    return cnt;
}
```

### 解题思路分析

首先对每一行，计算一个 presum

然后对任意两个列， $[j,k]$ ，计算  $j$  和  $k$  之间的所有数的和，因为对于每一行，我们已经计算了 presum，所以，对于  $[j,k]$  之间的数，可以通过  $\text{presum}[i][k] - (j == 0 ? 0 : \text{presum}[i][j-1])$  得到某一行在  $[j,k]$  之间的值。这样，对于任意两列  $[j,k]$  之间的数，我们可以形成一个临时的一维数组，表示从第 0 行到最后一行的数，那么，问题就转换成在这个一维数组里面，找一个连续子数组，使得这些数的和是 target，那么，后面这个问题就转换成 leetcode 560

<http://www.noteanddata.com/leetcode-560-Subarray-Sum-Equals-K-javascript-solution-note.html>

所以，这个是 presum 思想的 2d 版，类似的题目还有 leetcode 304 Range Sum Query 2D - Immutable

```
public int numSubmatrixSumTarget(int [][] matrix, int target) {
    int res = 0, m = matrix.length, n = matrix[0].length;
    for (int i = 0; i < m; ++i)
```

```

for (int j = 1; j < n; ++j)
    matrix[i][j] += matrix[i][j - 1];
for (int i = 0; i < n; ++i)
    for (int j = i; j < n; ++j) {
        Map<Integer, Integer> cntMap = new HashMap<>();
        cntMap.put(0, 1);
        int cur = 0;
        for (int k = 0; k < m; ++k) {
            cur += matrix[k][j] - (i > 0 ? matrix[k][i - 1] : 0);
            res += cntMap.getOrDefault(cur - target, 0);
            cntMap.put(cur, cntMap.getOrDefault(cur, 0) + 1);
        }
    }
return res;
}

private int cntTarget(int[] arr, int target) { // 比较一下与上面解法、写法的不同!!!
    int cnt = 0, sum = 0;
    Map<Integer, Integer> cntMap = new HashMap<>();
    cntMap.put(0, 1);
    for (int i = 0; i < arr.length; i++) {
        sum += arr[i];
        cnt += cntMap.getOrDefault(sum - target, 0);
        cntMap.compute(sum, (k, v) -> { //
            if (v == null) v = 0;
            return v + 1;
        });
    }
    return cnt;
}
}

public int numSubmatrixSumTarget(int[][] matrix, int target) {
    int m = matrix.length, n = matrix[0].length, cur = 0;
    int[][] sum = new int[m][n]; // sums of row sum only
    for (int i = 0; i < m; i++) {
        cur = 0;
        for (int j = 0; j < n; j++) {
            cur += matrix[i][j];
            sum[i][j] = cur;
        }
    }
    int cnt = 0;
    for (int j = 0; j < n; j++) {
        for (int k = j; k < n; k++) {
            int[] tmp = new int[m];
            for (int i = 0; i < m; i++)
                tmp[i] = sum[i][k] - (j == 0 ? 0 : sum[i][j - 1]);
            int oneCnt = cntTarget(tmp, target);
            cnt += oneCnt;
        }
    }
    return cnt;
}
}

```

### 12.3.1 Java O(NlgN) optimized Brute Force with Fenwick Tree O(NlogN)

我们假设 A 的大小为 n，如果采用暴力法逐个测试，则时间复杂度为  $O(n^2)$ ，应该过不了大数据测试。

我采取的方法是：首先计算使得每个元素  $A[i]$  要符合条件，需要 rotate 的 K 的集合，用线段表示；然后再扫描一遍，求出这些线段中重合最多的点，那么这个点对应的 rotate 次数就是题目所要计算的 K。对应  $A[i]$  来讲，如果  $A[i] \leq i$ ，那么它向左移动到 j 也可能维持  $A[i] \leq j$ ，所以我们计算出此时它向左移动的合法区间  $[0, i - A[i]]$ 。那么  $A[i]$  向右移动的合法区间是多少呢？我们知道它向右移动最多移动到  $n - 1$ ，即移动  $n - 1 - i$  步；而最少需要移动  $\max(1, A[i] - i)$  步，其中 1 表示  $A[i] \leq i$  的情况。那么如果这个区间合法，就可以同样构成了一个合法的移动区间  $[i + 1, n - \max(1, A[i] - i)]$ 。

得到多个线段构成的合法移动区间之后，我们的任务就是求出这些区间的最大重合点。首先对 segment 中的各个点进行排序，然后采用扫描线的方法计算最大重合处。为了便于区分某个点是起点还是终点，我们定义一个  $pair<int, bool>$  来表示点，并且让起点的 bool 值为 false，终点的 bool 值为 true，这样就可以在扫描到某个点之后，先处理起点，再处理终点。

由于每个  $A[i]$  最多对应 2 个合法移动区间，所以 segments 大小也是  $O(n)$  量级的。这样可以得知，本算法的时间复杂度是  $O(n\lg n)$ ，空间复杂度是  $O(n)$ 。

<https://www.codeleading.com/article/62063257456/>

- 不是我的代码风格，需要再学习一下

```

public int bestRotation(int[] A) {
    int LEN = A.length;
    int score, ans, ansInd, k; score = k = ans = ansInd = 0;
    for (int i = 0; i < LEN; i++) A[i] -= i;
    int[] bit = new int[3 * LEN + 1];
    for (int i = k; i < LEN; i++) bitAdd(bit, 3 * LEN + 1, vToIndex(A[i], LEN), 1);
    while (k < LEN) {
        score = bitPreSum(bit, vToIndex(-k, LEN));
        if (score > ans) {
            ansInd = k;
            ans = score;
        }
        k++;
    }
}

```

### 12.3. 1074. NUMBER OF SUBMATRICES THAT SUM TO TARGET - HARD

```
    }
    bitAdd(bit, 3 * LEN + 1, vToIndex(A[k], LEN), -1);
    bitAdd(bit, 3 * LEN + 1, vToIndex(A[k] - LEN, LEN), 1);
    k++;
}
return ansInd;
}
void bitAdd(int[] bit, int bLEN, int ind, int delta) {
    for(; ind < bLEN; ind = (ind | (ind + 1))) {
        bit[ind] += delta;
    }
}
int bitPreSum(int[] bit, int ind) {
    int ans = 0;
    for(; ind >= 0; ind = (ind & (ind + 1)) - 1) {
        ans += bit[ind];
    }
    return ans;
}
int vToIndex(int v, int LEN) {
    return v + 2 * LEN;
}
```



# Chapter 13

## Greedy

### 13.1 1585. Check If String Is Transformable With Substring Sort Operations - Hard

Given two strings s and t, you want to transform string s into string t using the following operation any number of times:

Choose a non-empty substring in s and sort it in-place so the characters are in ascending order. For example, applying the operation on the underlined substring in "14234" results in "12344".

Return true if it is possible to transform string s into string t. Otherwise, return false.

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

```
public boolean isTransformable(String s, String t) {
    ArrayDeque<Integer> [] q = new ArrayDeque[10];
    for (int i = 0; i < 10; i++)
        q[i] = new ArrayDeque<Integer>();
    for (int i = 0; i < s.length(); i++)
        q[s.charAt(i)-'0'].offerLast(i);
    for (char c : t.toCharArray()) {
        int d = c - '0';
        if (q[d].isEmpty()) return false;
        for (int i = 0; i < d; i++)
            if (!q[i].isEmpty() && q[i].peekFirst() < q[d].peekFirst())
                return false;
        q[d].pollFirst();
    }
    return true;
}
```

### 13.2 1330. Reverse Subarray To Maximize Array Value - Hard

You are given an integer array nums. The value of this array is defined as the sum of  $|nums[i]-nums[i+1]|$  for all  $0 \leq i < \text{nums.length}-1$ .

You are allowed to select any subarray of the given array and reverse it. You can perform this operation only once.

Find maximum possible value of the final array.

## 解题思路

给出一个整数数组，定义这个数组的和为相邻元素之差的和。现在允许将数组中的某个子数组进行整体翻转，然后计算对应的和。返回数组可得到的最大和。

这个题分析一下可知。有如下几种情况。

如果最优的最大和是不翻转的情况，那么就等于是翻转整个数组。

如果最优的情况下，翻转子串的边界是数组边界，那么用 $O(n)$ 的时间遍历两次，分别找出是右边界和左边界的情况就可以了。

如果最优的情况下，翻转子串的边界在数组内部，这是最复杂的情况，那么该如何考虑呢？假设我们已经计算出来数组在不翻转的情况下和 $S$ ，那么翻转之后其实只是改变了子串和整体数组的交界处的情况，其它地方均没有改变。因此假设边界左端点的值是 $b$ ，接着的前一个是 $a$ ，右端点是 $c$ ，接着的后一个是 $d$ 。那么变换之后的结果其实就是 $S + \text{abs}(d-b) + \text{abs}(c-a) - \text{abs}(b-a) - \text{abs}(d-c) = S + x$ 。只要 $x > 0$ ，就意味着增量为正，变换会导致结果优化。我们再假设 $a <= b, c <= d$ ，那么增量就是 $2 * (c - b)$ 。而分析可知我们要让 $c$ 尽可能大， $b$ 尽可能小。因此我们用两个 $O(n)$ 的时间。第一次找 $b$ ，怎么找呢？就是每两个相邻的，选大的那一个，然后在所有对中选对应 $b$ 最小的那一对。第二次找 $c$ 就是每两个相邻的，选小的那个，然后在所有的对中选最大的 $c$ 的那一对。

这里假设了 $abcd$ 的关系，其实也可以假设其他情况，最后都是一样的。

## 时空复杂度

代码如下，时间上， $O(n)$ ，空间上， $O(1)$ 。

```
public int maxValueAfterReverse(int[] a) {
    int n = a.length, ans = 0, sum = 0;
    if (n == 1) return 0;
    for (int i = 0; i < n-1; i++) // 不发生交换情况下的解
        sum += Math.abs(a[i] - a[i+1]);
    for (int i = 0; i < n-1; i++) // 发生交换、但交换的左端点为数组头的 case, 交换的区间左端点是数组的左边界
        ans = Math.max(ans, sum + Math.abs(a[i+1]-a[0]) - Math.abs(a[i+1]-a[i]));
    for (int i = n-1; i > 0; i--) // 发生交换、但交换的左端点为数组尾的 case, 交换的区间右端点是数组的右边界
        ans = Math.max(ans, sum + Math.abs(a[n-1] - a[i-1]) - Math.abs(a[i] - a[i-1]));
    // 交换区间左右端点在数组内部
    int l = 1; // 1. 找出左端点 b: 每两个相邻的，选最大的那一个数；在所有相邻对中，选最小的
    for (int i = 2; i < n; i++)
        if (Math.max(a[i], a[i-1]) < Math.max(a[l], a[l-1])) l = i;
    int r = 0; // 2. 找出右端点 c: 每两个相邻的，先最小的那一个数；在所有相邻对中，选最大的
    for (int i = 1; i < n-1; i++)
        if (Math.min(a[i], a[i+1]) > Math.min(a[r], a[r+1])) r = i;
    ans = Math.max(ans, sum + 2 * (Math.min(a[r], a[r+1]) - Math.max(a[l], a[l-1])));
    return ans;
}
```

# Chapter 14

## others

### 14.1 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., `nums1` 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;
}
```

### 14.2 Rectangle Area II

We are given a list of (axis-aligned) rectangles. Each `rectangle[i] = [xi1, yi1, xi2, yi2]`, where  $(xi_1, yi_1)$  are the coordinates of the bottom-left corner, and  $(xi_2, yi_2)$  are the coordinates of the top-right corner of the  $i$ th rectangle. Find the total area covered by all rectangles in the plane. Since the answer may be too large, return it modulo  $10^9 + 7$ .

```
public int rectangleArea(int[][][] rectangles) {
    ll = new ArrayList<>();
    long ans = 0;
    for (int [] r : rectangles)
        addRect(r, 0);
    for (int [] r : ll)
        ans = (ans + (long)(r[2] - r[0]) * (long)(r[3] - r[1])) % mod;
    return (int)ans;
}
static final int mod = (int)1e9 + 7;
List<int []> ll;
void addRect(int [] a, int i) { // i: idx
    if (i >= ll.size()) {
        ll.add(a);
        return ;
    }
    int [] r = ll.get(i);
    if (a[2] <= r[0] || a[0] >= r[2] || a[1] >= r[3] || a[3] <= r[1]) { // 被添加的，与现遍历的矩形，完全没有交集，直接往后遍历
        addRect(a, i+1);
        return ;
    }
    // 找出所有有交集的、交集部分——与现遍历矩形相比，多出来的部分，的四个顶点，再往后遍历
    if (a[0] < r[0]) // 左
        addRect(new int [] {a[0], a[1], r[0], a[3]}, i+1);
    if (a[2] > r[2]) // 右
        addRect(new int [] {r[2], a[1], a[2], a[3]}, i+1);
    if (a[1] < r[1]) // 下：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
        addRect(new int [] {Math.max(a[0], r[0]), a[1], Math.min(a[2], r[2]), r[1]}, i+1);
    if (a[3] > r[3]) // 上：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
        addRect(new int [] {Math.max(a[0], r[0]), r[3], Math.min(a[2], r[2]), a[3]}, i+1);
}
```

## 14.3 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;
}
```

## 14.4 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);
}
```

## 14.5 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  $\text{connections}[i] = [a_i, b_i]$  represents a connection between servers  $a_i$  and  $b_i$ . 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] = ++recdn;
    dfn[u] = recdn;
    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 节点更早出现的节点
                // --- 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);
    recdn = 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);
    recdn = 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;
}

```

## 14.6 891. Sum of Subsequence Widths - Hard 考 sorting 和对 subsequence 的理解

The width of a sequence is the difference between the maximum and minimum elements in the sequence.

Given an array of integers nums, return the sum of the widths of all the non-empty subsequences of nums. Since the answer may be very large, return it modulo 109 + 7.

A subsequence is a sequence that can be derived from an array by deleting some or no elements without changing the order of the remaining elements. For example, [3,6,2,7] is a subsequence of the array [0,3,1,6,2,2,7].

### 1. 解题思路与分析

- 这道题的最优解法相当的 tricky，基本有点脑筋急转弯的感觉了。在解题之前，我们首先要知道的是一个长度为 n 的数组，共有多少个子序列，如果算上空集的话，共有  $2^n$  个。

那么在给数组排序之后，对于其中任意一个数字 A[i]，其前面共有 i 个数是小于等于 A[i] 的，这 i 个数字共有  $2^i$  个子序列，它们加上 A[i] 都可以组成一个新的非空子序列，并且 A[i] 是这里面最大的数字，那么在宽度计算的时候，就要加

上  $A[i] \times (2^i)$ ,

同理,  $A[i]$  后面还有  $n-1-i$  个数字是大于等于它的, 后面可以形成  $2^{(n-1-i)}$  个子序列, 每个加上  $A[i]$  就都是一个新的非空子序列, 同时  $A[i]$  是这些子序列中最小的一个, 那么结果中就要减去  $A[i] \times (2^{(n-1-i)})$ 。对于每个数字都这么计算一下, 就是最终要求的所有子序列的宽度之和了。

可能你会怀疑虽然加上了  $A[i]$  前面  $2^i$  个子序列的最大值, 那些子序列的最小值减去了么? 其实是减去了的, 虽然不是在遍历  $A[i]$  的时候减去, 在遍历之前的数字时已经将所有该数字是子序列最小值的情况减去了, 同理,  $A[i]$  后面的那些  $2^{(n-1-i)}$  个子序列的最大值也是在遍历到的时候才加上的, 所以不会漏掉任何一个数字。

在写代码的时候有几点需要注意的地方, 首先, 结果  $res$  要定义为 `long` 型, 因为虽然每次会对  $1e9+7$  取余, 但是不能保证不会在取余之前就已经整型溢出, 所以要定义为长整型。

其次, 不能直接算  $2^i$  和  $2^{(n-1-i)}$ , 很容易溢出, 即便是长整型, 也有可能溢出。那么解决方案就是, 在累加  $i$  的同时, 每次都乘以一个 2, 那么遍历到  $i$  的时候, 也就乘到  $2^i$  了, 防止溢出的诀窍就是每次乘以 2 之后就立马对  $1e9+7$  取余, 这样就避免了指数溢出, 同时又不影响结果。

最后, 由于这种机制下的  $2^i$  和  $2^{(n-1-i)}$  不方便同时计算, 这里又用了一个 trick, 就是将  $A[i] \times (2^{(n-1-i)})$  转换为了  $A[n-1-i] \times 2^i$ , 其实二者最终的累加和是相等的:

```
sum(A[i] * 2^(n-1-i)) = A[0]*2^(n-1) + A[1]*2^(n-2) + A[2]*2^(n-3) + ... + A[n-1]*2^0
sum(A[n-1 - i] * 2^i) = A[n-1]*2^0 + A[n-2]*2^1 + ... + A[1]*2^(n-2) + A[0]*2^(n-1)
```

```
public int sumSubseqWidths(int[] a) {
    long mod = (int)1e9 + 7, c = 1;
    long ans = 0;
    Arrays.sort(a);
    for (int i = 0; i < a.length; i++) {
        ans = (ans + (long)a[i] * c - a[a.length-1-i] * c) % mod;
        c = (c << 1) % mod;
    }
    return (int)ans;
}
```

## 14.7 335. Self Crossing

Hard

225

433

Add to List

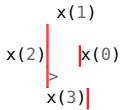
Share You are given an array of integers  $distance$ .

You start at point  $(0,0)$  on an X-Y plane and you move  $distance^1$  meters to the north, then  $distance^2$  meters to the west,  $distance^3$  meters to the south,  $distance^4$  meters to the east, and so on. In other words, after each move, your direction changes counter-clockwise.

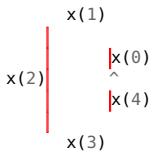
Return true if your path crosses itself, and false if it does not.

### 1. 解题思路与分析

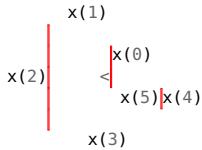
- 参考的网上大神们的解法, 实际上相交的情况只有以下三种情况:



第一类是第四条边和第一条边相交的情况, 需要满足的条件是第一条边大于等于第三条边, 第四条边大于等于第二条边。同样适用于第五条边和第二条边相交, 第六条边和第三条边相交等等, 依次向后类推的情况...



第二类是第五条边和第一条边重合相交的情况, 需要满足的条件是第二条边和第四条边相等, 第五条边大于等于第三条边和第一条边的差值, 同样适用于第六条边和第二条边重合相交的情况等等依次向后类推...



第三类是第六条边和第一条边相交的情况，需要满足的条件是第四条边大于等于第二条边，第三条边大于等于第五条边，第五条边大于等于第三条边和第一条边的差值，第六条边大于等于第四条边和第二条边的差值，同样适用于第七条边和第二条边相交的情况等等依次向后类推...

```
public boolean isSelfCrossing(int [] d) {
    int n = d.length;
    if (n < 4) return false;
    for (int i = 3; i < n; i++) { // 4 >= 2 && 1 >= 3
        // if (i % 3 == 0 && d[i] >= d[i-2] && d[i-3] >= d[i-1]) return true;
        // else if (i % 4 == 0 && d[i-1] == d[i-3] && d[i] >= d[i-2] - d[i-4]) return true; // 2 == 4 && 5 >= 3-1
        // else if (i % 5 == 0 && d[i-2] >= d[i-4] && d[i-3] >= d[i-1] && d[i-1] >= d[i-3] - d[i-5] && d[i] >= d[i-2] - d[i-4])
            if (d[i] >= d[i-2] && d[i-3] >= d[i-1]) return true;
        else if (i >= 4 && d[i-1] == d[i-3] && d[i] >= d[i-2] - d[i-4]) return true; // 2 == 4 && 5 >= 3-1
        else if (i >= 5 && d[i-2] >= d[i-4] && d[i-3] >= d[i-1] && d[i-1] >= d[i-3] - d[i-5] && d[i] >= d[i-2] - d[i-4])
            return true;
        // else if (i % 6 == 0 && d[i-4] + d[i] >= d[i-2] && d[i-1] <= d[i-3] && d[i-5] + d[i-1] >= d[i-3]) return true; // 这个条件不对
    }
    return false;
}
```

## 14.8 391. Perfect Rectangle - Hard

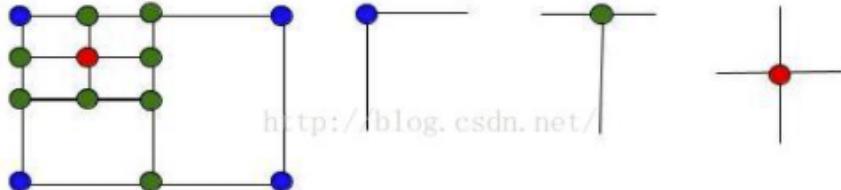
Given an array rectangles where rectangles[i] = [xi, yi, ai, bi] represents an axis-aligned rectangle. The bottom-left point of the rectangle is (xi, yi) and the top-right point of it is (ai, bi).

Return true if all the rectangles together form an exact cover of a rectangular region.

1. 解题思路与分析: 数顶点个数: 掩码 O(N)

思路：一种naive的做法是先计算总共小矩阵的面积和四个边界的面积，如果不相等则返回false。否则就要计算是否有重叠，计算重叠的方式是两两检查即可。这样时间复杂度是 $O(n^2)$ ，并不能过掉所有数据，所以我们需要一种更聪明的做法。

如果一堆小矩形要组成一个完美的大矩形需要满足以下条件：



1. 蓝色的点一个位置只能出现一次，并且只有四个这样的点
2. 绿色的点一个位置只能出现两次
3. 红色的点一个位置只能出现四次

也就是只能存在以上类型的点，并且我们给一个矩阵的四个角编号分别为1, 2, 3, 4，那么出现在同一个位置的点必须为一个矩阵不同编号的点。如此条件都满足即可构成一个完美矩形。

算法实现步骤如下：

1. 先遍历每一个矩阵，对每一个点进行处理，对于每一个位置的点用hash表来存储这个位置出现了几个corner，并且为了区分出现在这个位置的点是什么编号，我们可以用一个掩码来表示这个点的编号，这样也好判断是否这个位置出现了重复的编号的corner。
2. 对矩阵做好处理之后接下来就好遍历hash表查看对于每个位置来说是否满足以上规定的点的形式。任意一个点不满足条件即可返回false。

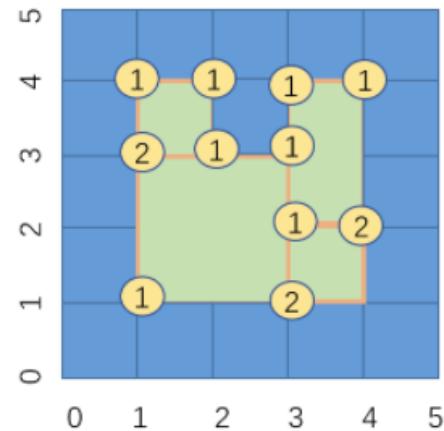
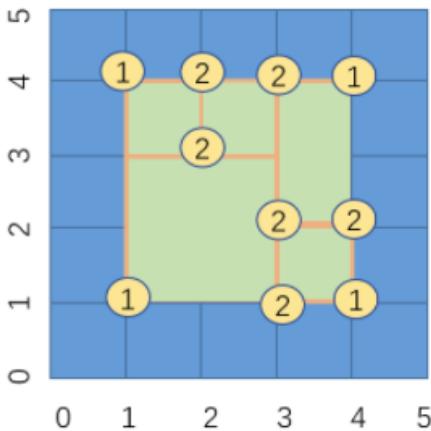
在实现的过程中为了简化代码用了一些技巧，比如为了在hash表中保存一个坐标，可以用字符串形式来保存。还有判断一个数是否是2的倍数可以用 $val \& (-val)$ 是否为0来判断。

这样时间复杂度即可降为 $O(n)$

```
public boolean isRectangleCover(int[][] arr) {
    Map<String, Integer> m = new HashMap<>();
    for (int[] a : arr)
        for (int i = 0; i < 4; i++) { // [0 1 2 3] 四种类型的顶点，第i个位置每个类型最多出现一次
            String cur = a[i / 2 * 2] + "-" + a[i % 2 * 2 + 1]; // 记录从左下角开始逆时针的四个顶点 [0 1 2 3]: 左下, 右下, 右上, 左上
            if ((m.getOrDefault(cur, 0) & (1 << i)) > 0) return false; // 同一形状、同一个角型出现了重叠
            m.put(cur, m.getOrDefault(cur, 0) | (1 << i));
        }
    int cnt = 0;
    for (Map.Entry<String, Integer> en : m.entrySet()) {
        int v = en.getValue();
        if ((v & (v - 1)) == 0 && cnt++ > 4) return false; // 只出现一次的顶点最多只有四个
        // if ((v & (v - 1)) > 0 && !(v == 15 || v == 10 || v == 12 || v == 5 || v == 3)) // 是 test case 太弱吗？检测不出来？
        if ((v & (v - 1)) > 0 && !(v == 15 || v == 12 || v == 10 || v == 5 || v == 9 || v == 3)) // 所有合法的子集除外
            return false;
    }
    return true; // 所有的点都合法了，就不用检测面积了
}
```

## 2. 解题思路与分析

因此，我们还需要从点的角度进行进一步判断。下面给出一个合法的例子如左图所示，不合法的例子如右图所示，同时用数字标记了所有小矩形包包含的顶点出现的次数。可以发现，合法的例子中，出现次数为1的顶点就是最后完美矩形的四个顶点，其他的顶点都出现了两次。而不合法的例子中，出现次数为1的顶点不只是完美矩形的四个顶点。

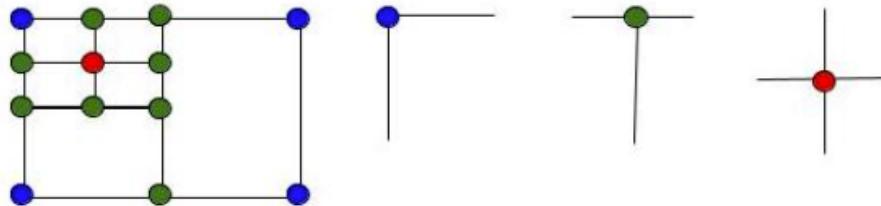


因此，我们可以在遍历小矩形的同时记录顶点出现的情况。如果当前顶点没有遍历过，则将其加入到集合中，否则将其从集合中删除。如果最后集合中只包含可能的完美矩形的四个顶点，那么说明例子是合法的，否则说明无法构成完美矩形。

总结，如果解该题需要遍历所有的小矩形，遍历过程中需要等到完美矩形的左下角和右上角坐标、矩形面积之和，以及顶点的出现情况，最后，判断小矩形面积之和和完美矩形面积之和是否相等，集合中是否只包含4个顶点且是完美矩形的四个顶点，只有这两个条件同时满足才能构成完美矩形。

```
public boolean isRectangleCover(int[][] arr) {
    Set<String> s = new HashSet<>();
    int i = Integer.MAX_VALUE, j = Integer.MAX_VALUE, x = Integer.MIN_VALUE, y = Integer.MIN_VALUE, sum = 0;
    for (int[] a : arr) {
        i = Math.min(i, a[0]);
        j = Math.min(j, a[1]);
        x = Math.max(x, a[2]);
        y = Math.max(y, a[3]);
        sum += (a[2] - a[0]) * (a[3] - a[1]);
        String bl = a[0] + "-" + a[1], tl = a[2] + "-" + a[1];
        String br = a[0] + "-" + a[3], tr = a[2] + "-" + a[3];
        if (!s.contains(bl)) s.add(bl); else s.remove(bl);
        if (!s.contains(br)) s.add(br); else s.remove(br);
        if (!s.contains(tl)) s.add(tl); else s.remove(tl);
        if (!s.contains(tr)) s.add(tr); else s.remove(tr);
    }
    String a = i + "-" + j, b = i + "-" + y;
    String c = x + "-" + j, d = x + "-" + y;
    if (!s.contains(a) || !s.contains(b) || !s.contains(c) || !s.contains(d) || s.size() != 4)
        return false;
    return sum == (x - i) * (y - j);
}
```

博主能力有限，只能去论坛中找各位大神的解法，发现下面两种方法比较fancy，也比较好理解。首先来看第一种方法，这种方法的设计思路很巧妙，利用了mask，也就是位操作Bit Manipulation的一些技巧，下面这张图来自这个帖子：



所有的矩形的四个顶点只会有下面蓝，绿，红三种情况，其中蓝表示该顶点周围没有其他矩形，T型的绿点表示两个矩形并排相邻，红点表示四个矩形相邻，那么在一个完美矩形中，蓝色的点只能有四个，这是个很重要的判断条件。我们再来看矩形的四个顶点，我们按照左下，左上，右上，右下的顺序来给顶点标号为1, 2, 4, 8，为啥不是1, 2, 3, 4呢，我们注意它们的二进制1(0001), 2(0010), 4(0100), 8(1000)，这样便于我们与或的操作，我们还需要知道的一个判定条件是，当一个点是某一个矩形的左下顶点时，这个点就不能是其他矩形的左下顶点了，这个条件对于这四种顶点都要成立，那么对于每一个点，如果它是某个矩形的四个顶点之一，我们记录下来，如果在别的矩形中它又是相同的顶点，那么直接返回false即可，这样就体现了我们标记为1, 2, 4, 8的好处，我们可以按位检查1。如果每个点的属性没有冲突，那么我们来验证每个点的mask是否合理，通过上面的分析，我们知道每个点只能是蓝，绿，红三种情况的一种，其中蓝的情况是mask的四位中只有一个1，分别就是1(0001), 2(0010), 4(0100), 8(1000)，而且蓝点只能有四个；那么对于T型的绿点，mask的四位中有两个1，那么就有六种情况，分别是12(1100), 10(1010), 9(1001), 6(0110), 5(0101), 3(0011)；而对于红点，mask的四位都是1，只有一种情况15(1111)，那么我们可以通过直接找mask是1, 2, 4, 8的个数，也可以间接通过找不是绿点和红点的个数，看是否是四个。最后一个判定条件是每个矩形面积累加和要等于最后的大矩形的面积，那么大矩形的面积我们通过计算最小左下点和最大右上点来计算出来即可，参见代码如下：

```
bool isRectangleCover(vector<vector<int>>& rectangles) {
    unordered_map<string, int> m;
    int min_x = INT_MAX, min_y = INT_MAX, max_x = INT_MIN, max_y = INT_MIN, area = 0, cnt = 0;
    for (auto rect : rectangles) {
        min_x = min(min_x, rect[0]);
        min_y = min(min_y, rect[1]);
        max_x = max(max_x, rect[2]);
        max_y = max(max_y, rect[3]);
        area += (rect[2] - rect[0]) * (rect[3] - rect[1]);
        if (!isValid(m, to_string(rect[0]) + "_" + to_string(rect[1]), 1)) return false; // bottom-left
        if (!isValid(m, to_string(rect[0]) + "_" + to_string(rect[3]), 2)) return false; // top-left
        if (!isValid(m, to_string(rect[2]) + "_" + to_string(rect[3]), 4)) return false; // top-right
        if (!isValid(m, to_string(rect[2]) + "_" + to_string(rect[1]), 8)) return false; // bottom-right
    }
    for (auto it = m.begin(); it != m.end(); ++it) {
        int t = it->second;
        if (t != 15 && t != 12 && t != 10 && t != 9 && t != 6 && t != 5 && t != 3) {
            ++cnt;
        }
    }
    return cnt == 4 && area == (max_x - min_x) * (max_y - min_y);
}
bool isValid(unordered_map<string, int>& m, string corner, int type) {
    int& val = m[corner];
    if (val & type) return false;
    val |= type;
    return true;
}
```

# Chapter 15

## HashMap

### 15.1 数有多少个数组数不清楚

### 15.2 1392. Longest Happy Prefix - Hard

A string is called a happy prefix if it is a non-empty prefix which is also a suffix (excluding itself).

Given a string s, return the longest happy prefix of s. Return an empty string "" if no such prefix exists.

```
// 频繁的字符串操作 (substring 和 equals 操作) 会大幅消耗执行时间，也会导致 TLE 时间超时。  
// 因此我们可以使用字符串的 hash 值方式来比较前后缀是否相同。这里我们需要普及一个知识点，任意一个字符串的 Hash 值的计算公式为：  
// int hash=s[0]31^(n1)+s[1]31^(n2) +...+s[n2]31^1+s[n1]31^0  
// 对于前缀 hash，每次长度加一后 hash 的变化应该是：  
// hash = hash*31 + 新添头字符 ch  
// 对于后缀 hash，每次长度加一后 hash 的变化应该是：  
// hash = hash + 新添尾字符 *31^t (t 为后缀长度减 1)  
// 这样我们每次只需要比较前缀 hash 与后缀 hash 是否相同即可。  
public String longestPrefix(String s) {  
    int n = s.length(), hashPre = 0, hashSuf = 0;  
    int left = 0, right = n-1, pow = 1, maxLen = 0;  
    String res = "";  
    while (left < n-1) {  
        hashPre = hashPre * 31 + s.charAt(left);  
        hashSuf = hashSuf + s.charAt(right)*pow;  
        if (hashPre == hashSuf) maxLen = left + 1;  
        left++;  
        right--;  
        pow *= 31;  
    }  
    return maxLen == 0 ? "" : s.substring(0, maxLen);  
}
```

### 15.3 336. Palindrome Pairs - Hard

Given a list of unique words, return all the pairs of the distinct indices (i, j) in the given list, so that the concatenation of the two words words[i] + words[j] is a palindrome.

#### 15.3.1 解题思路与分析

- 利用字典 wmap 保存单词 -> 下标的键值对
- 遍历单词列表 words, 记当前单词为 word, 下标为 idx:

- 若当前单词 word 本身为回文，并且 words 中存在空串，则将空串下标 bidx 与 idx 加入答案
- 若当前单词的逆序串在 words 中，则将逆序串下标 ridx 与 idx 加入答案
- 将当前单词 word 拆分为左右两半 left, right。
  - 若 left 为回文，并且 right 的逆序串在 words 中，则将 right 的逆序串下标 rridx 与 idx 加入答案
  - 若 right 为回文，并且 left 的逆序串在 words 中，则将 left 的逆序串下标 idx 与 rliidx 加入答案

情况分析清楚以后，针对每个情况逐个判断，然后提前建好 hashmap 以后判断就好。

细节部分是不要忘记空字符串的情况，然后针对整个字符串反过来的情况，不要有重复。

```
private boolean isPalindrome(String s, int i, int j) {  
    while (i < j)  
        if (s.charAt(i++) != s.charAt(j--)) return false;  
    return true;  
}
```

```

public List<List<Integer>> palindromePairs(String[] words) {
    Map<String, Integer> map = new HashMap<>();
    List<List<Integer>> ans = new ArrayList<>();
    for (int i = 0; i < words.length; i++)
        map.put(words[i], i);
    for (int i = 0; i < words.length; i++) {
        if (words[i].equals("")) {
            for (int j = 0; j < words.length; j++) {
                String w = words[j];
                if (isPalindrome(w, 0, w.length()-1) && j != i) {
                    ans.add(List.of(i, j));
                    ans.add(List.of(j, i));
                }
            }
            continue;
        }
        StringBuilder sb = new StringBuilder(words[i]);
        sb.reverse();
        String bw = sb.toString();
        if (map.containsKey(bw)) {
            int res = map.get(bw);
            if (res != i) ans.add(List.of(i, res));
        }
        for (int j = 1; j < bw.length(); j++) {
            if (isPalindrome(bw, 0, j-1)) {
                String s = bw.substring(j);
                if (map.containsKey(s))
                    ans.add(List.of(i, map.get(s)));
            }
            if (isPalindrome(bw, j, bw.length()-1)) {
                String s = bw.substring(0, j);
                if (map.containsKey(s))
                    ans.add(List.of(map.get(s), i));
            }
        }
    }
    return ans;
}

```

## 15.4 KMP 算法

1	2
i 01234567890123456789012	
s ABC ABCDAB ABCDABCDABDE	
p ABCDABD\$	next=0
j 01234567	

j	0	1	2	3	4	5	6	7
p	A	B	C	D	A	B	D	\$
nxt	0	0	0	0	0	1	2	0

1	2
i 01234567890123456789012	
s ABC ABCDAB ABCDABCDABDE	
p ABCDABD\$	next=0
j 01234567	

1	2
i 01234567890123456789012	
s ABC ABCDAB ABCDABCDABDE	
p ABCDABD\$	next=2
j 01234567	

1	2
i 01234567890123456789012	
s ABC ABCDAB ABCDABCDABDE	
w ABCDABD\$	next=0
j 01234567	

1	2
i 01234567890123456789012	
s ABC ABCDAB ABCDABCDABDE	
w ABCDABD\$	next=2
j 01234567	

1	2
i 01234567890123456789012	
s ABC ABCDAB ABCDABCDABDE	
p ABCDABD\$	next=0
j 01234567	

j	0	1	2	3
p	A	A	A	\$
nxt	0	0	1	2

i 01234567  
s AAAA~~AAAAA~~  
p AAA\$  
j 0123      next=2

i 01234567  
s A~~AAA~~AAAA  
p AA~~A~~\$  
j 0123      next=2

i 01234567  
s AA~~AAA~~AAA  
p AA~~A~~\$  
j 0123      next=2

i 01234567  
s AAAA~~AAA~~A  
p A~~AA~~\$  
j 0123      next=2

i 01234567  
s AAAA~~AA~~A  
p AA~~A~~\$  
j 0123      next=2

i 01234567  
s AAAA~~AA~~A  
p AA~~A~~\$  
j 0123      next=2

### KMP::Match

- Case 1:  $s[i] == p[j]$ , match
  - Action:  $++i$ ,  $++j$ : try next pair
    - ↓                  ↓                  ↓
    - ABCABCD   =>   ABCABCD   =>   ABCABCD
    - ABC...            ABC...            ABC...
- Case 2:  $s[i] != p[j]$ , a mismatch happened
  - Key idea: use partial matching information
  - Let  $q = p[0 \sim j-1]$  be the partial matched string
  - Action:  $i$  remains the same, if  $j == 0$ :  $++i$  else:  $j = \text{next}[j]$ ,
- Case 2.1:  $q$  is empty (e.g.  $j = 0$ )
  - $++i$ , try next one
    - ↓                  ↓
    - ABCABCD            ABCABCD
    - CA                =>    CA

- Case 2.2:  $\text{suffix}(q) \cap \text{prefix}(q) = \emptyset$ 
  - $\text{next}[j]$  must be 0
  - skip the partial match (no potential matches in between)
 
$$\begin{array}{ccc} \text{ABC}\text{A}\text{BCD} & \quad & \text{ABC}\text{ABC}\text{D} \\ \downarrow & \quad & \downarrow \\ \text{ABCD} & => & \text{ABCD} \end{array}$$
- Case 2.3:  $\text{suffix}(q) \cap \text{prefix}(q) \neq \emptyset$ 
  - $\text{next}[j] = \text{len}(\text{LPS}(q))$
  - e.g. ABCDAB, AB is the longest suffix that is also a prefix
  - Skip the prefix AB and start matching from 'C'
 
$$\begin{array}{ccc} \text{ABC}\text{DAB}\text{C}\text{DABE} & \quad & \text{ABC}\text{DAB}\text{C}\text{DABE} \\ \downarrow & \quad & \downarrow \\ \text{ABCDABD} & => & \text{ABCDABD} \end{array}$$
- Case 2.4:  $q = p$ , we had a full match
  - Action:  $++i$ ,  $j = \text{next}[j]$ , treat it as a mismatch

```
// straight forward KMP algorithm - 8ms - beats 94% time - O(N) time and O(N) space
public String longestPrefix(String s) {
    int i, x, N = s.length();
    int [] LPS = new int[N];
    LPS[0] = 0;
    for (i = 1; i < N; i++){
        x = LPS[i - 1];
        while (s.charAt(i) != s.charAt(x)){
            if (x == 0){
                x = -1;
                break;
            }
            x = LPS[x - 1];
        }
        LPS[i] = x + 1;
    }
    return s.substring(0, LPS[N - 1]);
}

string longestPrefix(string s) {
    vector<int> lps(s.size(), 0);
    size_t i = 0, j = 1;
    while (j < s.size()){
        if (s[i] == s[j]) lps[j++] = (i++) + 1; // situ.1
        else if (i != 0) i = lps[i - 1]; // situ.2
        else lps[j++] = 0; // situ.3
    }
    return s.substr(0, lps.back());
}
```

## 15.5 1172. Dinner Plate Stacks - Hard

You have an infinite number of stacks arranged in a row and numbered (left to right) from 0, each of the stacks has the same maximum capacity.

Implement the DinnerPlates class:

DinnerPlates(int capacity) Initializes the object with the maximum capacity of the stacks capacity.  
void push(int val) Pushes the given integer val into the leftmost stack with a size less than capacity.  
int pop() Returns the value at the top of the rightmost non-empty stack and removes it from that stack, and returns -1 if all the stacks are empty.  
int popAtStack(int index) Returns the value at the top of the stack with the given index index and removes it from that stack or returns -1 if the stack with that given index is empty.

```
Stack<Stack<Integer>> stacks = new Stack<>();
TreeSet<Integer> set = new TreeSet<>(); // set:
int capacity;
public DinnerPlates(int capacity) {
    this.capacity = capacity;
    stacks = new Stack<>();
}
public void push(int val) {
    if (set.size() != 0) {
        int idx = set.iterator().next();
        stacks.get(idx).push(val);
    }
}
```

```

        if (stacks.get(idx).size() == capacity)
            set.remove(idx);
    } else {
        if (stacks.isEmpty() || stacks.peek().size() == capacity) {
            stacks.add(new Stack<>()); // 更高效一点儿?
            // stacks.push(new Stack<>());
            stacks.peek().add(val);
        } else stacks.peek().add(val);
    }
}
public int pop() {
    if (!stacks.isEmpty()) {
        int k = stacks.peek().pop();
        while (!stacks.isEmpty() && stacks.peek().isEmpty())
            set.remove(stacks.size()-1);
        stacks.pop();
    }
    return k;
}
return -1;
}
public int popAtStack(int index) {
    if (index >= stacks.size() || stacks.get(index).size() == 0)
        return -1;
    if (index == stacks.size()-1)
        return this.pop();
    set.add(index);
    return stacks.get(index).pop();
}
}

```

- 用了双端队列的一个方法

```

List<Deque<Integer>> stackList = new ArrayList<>();
TreeSet<Integer> pushIdxSet = new TreeSet<>();
TreeSet<Integer> popIdxSet = new TreeSet<>();
int capacity;
public DinnerPlates(int capacity) {
    stackList = new ArrayList<>();
    pushIdxSet = new TreeSet<>();
    popIdxSet = new TreeSet<>();
    this.capacity = capacity;
    stackList.add(new ArrayDeque<>());
    pushIdxSet.add(0);
}
public void push(int val) {
    int idx = pushIdxSet.first();
    if (stackList.get(idx).isEmpty())
        popIdxSet.add(idx);
    stackList.get(idx).offerLast(val);
    if (stackList.get(idx).size() == capacity) {
        if (idx == stackList.size() - 1) {
            stackList.add(new ArrayDeque<>());
            pushIdxSet.add(idx + 1);
        }
        pushIdxSet.remove(idx);
    }
}
public int pop() {
    if (popIdxSet.isEmpty()) return -1;
    int idx = popIdxSet.last();
    if (stackList.get(idx).size() == capacity)
        pushIdxSet.add(idx);
    int res = stackList.get(idx).pollLast();
    if (stackList.get(idx).isEmpty())
        popIdxSet.remove(idx);
    return res;
}
public int popAtStack(int index) {
    if (index >= stackList.size()) return -1;
    if (stackList.get(index).isEmpty()) return -1;
    if (stackList.get(index).size() == capacity)
        pushIdxSet.add(index);
    int res = stackList.get(index).pollLast();
    if (stackList.get(index).isEmpty())
        popIdxSet.remove(index);
    return res;
}
}

```

## 15.6 2080. Range Frequency Queries - Medium

Design a data structure to find the frequency of a given value in a given subarray.

The frequency of a value in a subarray is the number of occurrences of that value in the subarray.

Implement the RangeFreqQuery class:

RangeFreqQuery(int[] arr) Constructs an instance of the class with the given 0-indexed integer array arr. int query(int left, int right, int value) Returns the frequency of value in the subarray arr[left...right]. A subarray is a contiguous sequence of elements within an array. arr[left...right] denotes the subarray that contains the elements of nums between indices left and right (inclusive).

### 15.6.1 解题思路与分析: 最简单粗暴的

```
Map<Integer, TreeMap<Integer, Integer>> m = new HashMap<>();
public RangeFreqQuery(int[] arr) {
    for (int i = 0; i < arr.length; i++)
        m.computeIfAbsent(arr[i], z -> new TreeMap<>()).put(i, m.get(arr[i]).size()); // 这里写得好 tricky, 考试的时候紧张就想不到!!!
}
public int query(int left, int right, int value) {
    if (!m.containsKey(value)) return 0;
    TreeMap<Integer, Integer> map = m.get(value);
    Integer a = map.ceilingKey(left), b = map.floorKey(right);
    if (a == null || b == null) return 0;
    return map.get(b) - map.get(a) + 1;
}
```

### 15.6.2 解题思路与分析: binary Search

```
Map<Integer, List<Integer>> map = new HashMap<>();
public RangeFreqQuery(int[] arr) {
    for (int i = 0; i < arr.length; i++)
        map.computeIfAbsent(arr[i], z -> new ArrayList<>()).add(i);
}
public int query(int left, int right, int value) {
    if (!map.containsKey(value)) return 0;
    List<Integer> l = map.get(value);
    int s = Collections.binarySearch(l, left);
    int e = Collections.binarySearch(l, right);
    if (s < 0) s = (s + 1) * (-1);
    if (e < 0) e = (e + 2) * (-1);
    return e - s + 1;
}
```

- 与上面相同的想法，但自己实现函数的

```
Map<Integer, List<Integer>> map = new HashMap();
public RangeFreqQuery(int[] arr) {
    int n = arr.length;
    for(int i = 0; i < n; i++)
        map.computeIfAbsent(arr[i], z -> new ArrayList<>()).add(i);
}
public int query(int left, int right, int value) {
    List<Integer> A = map.get(value);
    if (A == null || left > A.get(A.size()-1) || right < A.get(0))
        return 0;
    int i = ceil(A, left), j = floor(A, right);
    return j-i+1;
}
public int ceil(List<Integer> A, int x){
    int left = 0, right = A.size()-1;
    if (x < A.get(0))
        return 0;
    while (left < right) {
        int mid = (left+right)/2;
        if (A.get(mid) < x)
            left = mid + 1;
        else
            right = mid;
    }
    return left;
}
public int floor(List<Integer> A, int x){
    int left = 0, right = A.size ()-1;
    if (x > A.get (right))
        return right;
    while (left < right) {
        int mid = (left+right)/2+1;
        if (A.get (mid) > x)
            right = mid - 1;
        else
            left = mid;
    }
    return left;
}
```

### 15.6.3 解题思路与分析: segment tree

```
Map<Integer, Integer>[] freq;
int n;
```

```
public RangeFreqQuery(int[] arr) {
    n = arr.length;
    freq = new HashMap[4 * n];
    for(int i = 0; i < 4 * n; i++)
        freq[i] = new HashMap<>();
    build(1, 0, n - 1, arr);
}

private void build(int id, int start, int end, int[] arr){
    if(start == end)
        freq[id].put(arr[start], 1);
    else{
        int mid = (start + end) / 2, left = 2 * id, right = 2 * id + 1;
        build(left, start, mid, arr);
        build(right, mid + 1, end, arr);
        for(int i : freq[left].keySet())
            freq[id].put(i, freq[id].getOrDefault(i, 0) + freq[left].get(i));
        for(int i : freq[right].keySet())
            freq[id].put(i, freq[id].getOrDefault(i, 0) + freq[right].get(i));
    }
}
public int query(int left, int right, int value) {
    return find(1, 0, n - 1, left, right, value);
}
private int find(int id, int start, int end, int l, int r, int value){
    if(r < start || end < l)
        return 0;
    else if(start == end)
        return freq[id].getOrDefault(value, 0);
    else if(l <= start && end <= r)
        return freq[id].getOrDefault(value, 0);
    else{
        int mid = (start + end) / 2;
        int left = find(2 * id, start, mid, l, r, value);
        int right = find(2 * id + 1, mid + 1, end, l, r, value);
        return left + right;
    }
}
```



# Chapter 16

## binary Search

### 16.1 LeetCode Binary Search Summary 二分搜索法小结

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

#### 16.1.1 标准二分查找

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

循环终止的条件包括：

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

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

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

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

#### 16.1.2 二分查找左边界

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

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

##### 1. 左边界查找类型 1

类型 1 包括了上面说的第一种, 第二种情况。

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

```

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

```

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

与标准的二分查找不同:

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

其次, 这里的循环条件是 `left < right`。

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

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

```

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

```

## 2. 左边界查找类型 2

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

```

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

```

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

### 16.1.3 二分查找右边界

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

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

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

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

右边界的查找一般来说不会单独使用，如有需要，一般是需要同时查找左右边界。

### 16.1.4 二分查找左右边界

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

### 16.1.5 二分查找极值

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

Find Peak Element

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

Given an input array  $nums$ , where  $nums[i] < nums[i+1]$ , find a peak element and return its index.

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

You may imagine that  $nums[-1] = nums[n] = -$ .

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

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

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

```

int mid = left + (right - left) / 2;
if (nums[mid] == target) return mid;
else if (nums[mid] < target)
    left = mid + 1;
else
    right = mid - 1;
}
// 如果没找到就只能返回-1
return -1;
}

// 模板二，适合判断当前 index 和 index + 1 之间的关系。
// right 指针一开始的定义是在数组下标范围外的, [left, right), 所以在需要移动 right 指针的时候不能写成 right = mid。这样会遗漏掉一些下标的判断。
public int binarySearch3(int[] nums, int target) {
    // right 不在下标范围内
    // [left, right)
    int left = 0;
    int right = nums.length;
    // while 循环跳出的条件是 left == right
    // 这个模板比较适合判断当前 index 和 index + 1 之间的关系
    // left < right, example, left = 0, right = 1
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) return mid;
        else if (nums[mid] < target)
            left = mid + 1;
        else
            // 因为搜索范围是左闭右开所以这里不能-1
            right = mid;
    }
    // 最后的特判
    if (left != nums.length && nums[left] == target)
        return left;
    return -1;
}

// while 条件不满足的时候, left + 1 == right, 两下标应该指向某个下标 i 和 i + 1。这样如果有什么特殊的值需要判断, 应该不是 left 就是 right 了。
public int binarySearch1(int[] nums, int target) {
    // left 和 right 都在数组下标范围内
    // [left, right]
    int left = 0;
    int right = nums.length - 1;
    // 举例, start - 0, end = 3
    // 中间隔了起码有 start + 1 和 start + 2 两个下标
    // 这样跳出 while 循环的时候, start + 1 == end
    // 才有了最后的两个判断
    while (left + 1 < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == target) return mid;
        else if (nums[mid] < target)
            left = mid;
        else
            right = mid;
    }
    // 特判
    if (nums[left] == target) return left;
    if (nums[right] == target) return right;
    // 如果没找到就只能返回-1
    return -1;
}

```

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

### 16.1.6 第一类：需查找和目标值完全相等的数

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

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

会返回 3，也就是 target 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

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

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

```

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

```

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

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

第二类应用实例:

[Heaters](#), [Arranging Coins](#), [Valid Perfect Square](#), [Max Sum of Rectangle No Larger Than K](#), [Russian Doll Envelopes](#)

第二类变形应用: [Valid Triangle Number](#)

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

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

```

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

```

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

第三类应用实例:

[Kth Smallest Element in a Sorted Matrix](#)

第三类变形应用示例:

[Sqrt\(x\)](#)

### 16.1.9 第四类: 用子函数当作判断关系 (通常由 mid 计算得出)

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

第四类应用实例:

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

### 16.1.10 第五类: 其他 (通常 target 值不固定)

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

类似的还有一道 [H-Index II](#), 这道题的 `target` 也不是一个固定值, 而是 `len-mid`, 这就很意思了, 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

## 16.2 793. Preimage Size of Factorial Zeroes Function

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

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

```
private long numberOfTrailingZeros(long v) {
    long cnt = 0;
    for (; v > 0; v /= 5)
        cnt += v / 5;
    return cnt;
}

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

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

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

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

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

## 16.3 2040. Kth Smallest Product of Two Sorted Arrays - Hard

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

### Thinking Process

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

### Algorithm

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

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

- 另一种相当于换汤不换药的写法

```

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

```



# Chapter 17

## Tree 树结构：各种新型数据结构

### 17.1 最大深度到树的最大直径（不一定经过根节点）

- 【亲爱的表哥的活宝妹，任何时候，亲爱的表哥的活宝妹，就是一定要、一定会嫁给活宝妹的亲爱的表哥!!! 爱表哥，爱生活!!!】
- 【亲爱的表哥的活宝妹，现在再看、最大直径，很简单】文件要删除一些不必要的简单题目去，把文件弄小【TODO】:

#### 17.1.1 104. Maximum Depth of Binary Tree

- Given the root of a binary tree, return its maximum depth.
- A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

##### 1. 深度优先搜索

```
// DFS: 深度优先遍历
public int maxDepth(TreeNode r) {
    if (r == null) return 0;
    // 下面这个 CASE: 仔细想也是被上面的包括的，所以可以不用
    if (r.left == null && r.right == null) return 1; // 这句可以不用要
    int left = maxDepth(r.left);
    int right = maxDepth(r.right);
    // 下面这一行包括上面那一句，所以上面那一句可以不用
    return Math.max(left, right) + 1;
}
```

##### 2. 广度优先搜索

```
// 广度优先遍历
public int maxDepth(TreeNode r) {
    if (r == null) return 0;
    int cnt = 0;
    Deque<TreeNode> q = new ArrayDeque<>();
    q.offerFirst(r);
    while (!q.isEmpty()) {
        int qsize = q.size();
        for (int size = qsize-1; size >= 0; size--) {
            TreeNode cur = q.pollLast();
            if (cur.left != null) q.offerFirst(cur.left);
            if (cur.right != null) q.offerFirst(cur.right);
        }
        cnt++;
    }
    return cnt;
}
```

#### 17.1.2 543. Diameter of Binary Tree

- Given the root of a binary tree, return the length of the diameter of the tree.
- The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root.
- The length of a path between two nodes is represented by the number of edges between them.

```

public int diameterOfBinaryTree(TreeNode r) {
    traverse(r);
    return max;
}
int max = 0;
int traverse(TreeNode r) {
    if (r == null) return 0; // 叶子节点会是 1, 成了数节点个数
    int left = traverse(r.left);
    int right = traverse(r.right);
    max = Math.max(max, left + right);
    return Math.max(left, right) + 1;
}

```

### 17.1.3 2385. Amount of Time for Binary Tree to Be Infected

- You are given the root of a binary tree with unique values, and an integer start. At minute 0, an infection starts from the node with value start.
- Each minute, a node becomes infected if:
  - The node is currently uninfected.
  - The node is adjacent to an infected node.
- Return the number of minutes needed for the entire tree to be infected.

#### 1. 提示的思路如下：

- Convert the tree to an undirected graph to make it easier to handle.
- Use BFS starting at the start node to find the distance between each node and the start node. The answer is the maximum distance.
- 感觉因为树的题目简单，做题的时候只埋头做题了，但是没有很好地总结；思路比较死，可能理解得不是很透彻，所以解题的时候还不能够灵活应用。

#### 2. BFS 简洁精炼版的

```

public int amountOfTime(TreeNode r, int start) {
    buildGraph(r, null);
    // 下面这里注意：两个数据结构的初始化方式
    return maxDistance(new ArrayDeque<>(List.of(start)), new HashSet<>(Set.of(start)));
}
Map<Integer, List<Integer>> m = new HashMap<>();
int maxDistance(Deque<Integer> q, HashSet<Integer> vis) {
    int cnt = 0;
    while (!q.isEmpty()) {
        int size = q.size();
        while (size-- > 0) {
            int u = q.pollLast();
            vis.add(u);
            if (m.get(u) == null) continue;
            for (int v : m.get(u)) {
                if (vis.contains(v)) continue;
                q.offerFirst(v);
            }
        }
        cnt++;
    }
    return cnt-1;
}
void buildGraph(TreeNode r, TreeNode p) { // 我还是比较喜欢自己这种相对简洁的方法
    if (r == null) return;
    // 这里是，从当前节点，加父节点，加左右两个子节点；简洁，但是需要父节点作为参数
    m.computeIfAbsent(r.val, z -> new ArrayList<>());
    if (p != null) m.get(r.val).add(p.val);
    if (r.left != null) m.get(r.val).add(r.left.val);
    if (r.right != null) m.get(r.val).add(r.right.val);
    buildGraph(r.left, r);
    buildGraph(r.right, r);
}

```

#### 3. DFS 简洁精炼版的

```

public int amountOfTime(TreeNode r, int start) {
    buildGraph(r, null);
    maxDistance(new HashSet<>(), start, 0);
    return max;
}
Map<Integer, List<Integer>> m = new HashMap<>();
int max = 0;
void maxDistance(Set<Integer> vis, int u, int d) {

```

```

if (vis.contains(u)) return ;
vis.add(u);
max = Math.max(max, d);
for (int v : m.get(u))
    maxDistance(vis, v, d+1);
}
void buildGraph(TreeNode r, TreeNode p) { // 我还是比较喜欢自己这种相对简洁的方法
    if (r == null) return ;
    // 这里是, 从当前节点, 加父节点, 加左右两个子节点; 简洁, 但是需要父节点作为参数
    m.computeIfAbsent(r.val, z -> new ArrayList<>());
    if (p != null) m.get(r.val).add(p.val);
    if (r.left != null) m.get(r.val).add(r.left.val);
    if (r.right != null) m.get(r.val).add(r.right.val);
    buildGraph(r.left, r);
    buildGraph(r.right, r);
}
void buildGraph(TreeNode r) {
    if (r == null) return ;
    m.computeIfAbsent(r.val, z -> new ArrayList<>());
    // 从左右两个确定存在的子节点, 加父节点
    if (r.left != null) {
        m.computeIfAbsent(r.left.val, z -> new ArrayList<>()).add(r.val);
        m.get(r.val).add(r.left.val); // 从当前节点, 加左右两个子节点;
        buildGraph(r.left);
    }
    if (r.right != null) {
        m.computeIfAbsent(r.right.val, z -> new ArrayList<>()).add(r.val);
        m.get(r.val).add(r.right.val); // 从当前节点, 加左右两个子节点;
        buildGraph(r.right);
    }
}

```

### 17.1.4 979. Distribute Coins in Binary Tree

You are given the root of a binary tree with  $n$  nodes where each node in the tree has  $\text{node.val}$  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.

```

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;
}

```

### 17.1.5 1719. Number Of Ways To Reconstruct A Tree - Hard

You are given an array pairs, where  $\text{pairs}[i] = [\text{xi}, \text{yi}]$ , and:

There are no duplicates.  $\text{xi} < \text{yi}$  Let ways be the number of rooted trees that satisfy the following conditions:

The tree consists of nodes whose values appeared in pairs. A pair  $[\text{xi}, \text{yi}]$  exists in pairs if and only if  $\text{xi}$  is an ancestor of  $\text{yi}$  or  $\text{yi}$  is an ancestor of  $\text{xi}$ . Note: the tree does not have to be a binary tree. Two ways are considered to be different if there is at least one node that has different parents in both ways.

Return:

$0$  if  $\text{ways} = 0$   $1$  if  $\text{ways} = 1$   $2$  if  $\text{ways} > 1$  A rooted tree is a tree that has a single root node, and all edges are oriented to be outgoing from the root.

An ancestor of a node is any node on the path from the root to that node (excluding the node itself). The root has no ancestors.

#### 1. 解题思路与分析

```

public int checkWays(int[][] pairs) { // 自顶向下
    int max = 0; // [1, 500]
    for (int[] p : pairs) // 求出节点的最大值
        max = Math.max(max, Math.max(p[0], p[1]));
    int[] cnt = new int[max+1]; // 记录每个节点的祖先关系数量
    int[][] adj = new int[max+1][max+1]; // 是否存在祖孙关系的图
    for (int[] p : pairs) {
        cnt[p[0]]++;
        cnt[p[1]]++;
        adj[p[0]][p[1]] = 1;
    }
}

```

```

        adj[p[1]][p[0]] = 1;
    }
    Integer[] nodes = new Integer[max+1]; // 创建一个新的数组，可以方便后面的按祖先关系数量大小将节点排序，和将零散的节点集中到前面。
    int n = 0; // 使用包装整数类型，方便后面调用 API 排序
    for (int i = 1; i <= max; i++) {
        if (cnt[i] > 0) nodes[n++] = i;
    }
    Arrays.sort(nodes, 0, n, (a, b) -> cnt[b] - cnt[a]); // 按照祖先关系数量从大到小排序
    if (cnt[nodes[0]] != n-1) return 0; // 当根节点不满足要求
    int[] par = new int[max+1];
    int[][] allPar = new int[max+1][max+1];
    for (int i = 0; i < n; i++) {
        for (int j = i-1; j >= 0; j--) {
            if (adj[nodes[i]][nodes[j]] == 1) {
                par[nodes[i]] = nodes[j]; // 记录父节点
                for (int f = nodes[j]; f != 0; f = par[f]) // 自底向上：向祖先节点遍历，记录祖先节点，循环遍历直到根节点
                    allPar[nodes[i]][f] = 1;
                break; // 父节点只有一个，已经找到一个合法父节点，并且更新了所有的父节点，就可以不用再遍历了
            }
        }
        int ans = 1;
        for (int i = 1; i <= max; i++) {
            for (int j = i+1; j <= max; j++) {
                if (adj[i][j] == 1 && cnt[i] == cnt[j]) ans = 2; // 可以调换位置，有多个解
                if (adj[i][j] != (allPar[i][j] | allPar[j][i]))
                    return 0; // 有冲突，无解，出现在已经记录了当前节点和祖先节点的关系，但是 pairs 中没有该关系
            }
        }
    }
    return ans;
}

```

## 2. 解题思路与分析: dfs: 这个方法好慢

```

public int checkWays(int[][] pairs) { // 这个方法好慢
    for (int[] p : pairs) {
        adj.computeIfAbsent(p[0], z -> new HashSet<>()).add(p[1]);
        adj.computeIfAbsent(p[1], z -> new HashSet<>()).add(p[0]);
    }
    return helper(adj.keySet());
}
Map<Integer, Set<Integer>> adj = new HashMap<>();
int helper(Set<Integer> nodes) {
    Map<Integer, List<Integer>> lenMap = new HashMap<>();
    for (Integer v : nodes)
        lenMap.computeIfAbsent(adj.get(v).size(), z -> new ArrayList<>()).add(v);
    if (!lenMap.containsKey(nodes.size()-1)) return 0; // 不存在合法的根节点
    Integer root = lenMap.get(nodes.size()-1).get(0); // 这个任命为根的节点是否带有随机性？lenMap 里 key 为 nodes.size()-1 的值应该只有一个
    for (Integer v : adj.get(root)) // 因为需要 dfs 自顶向下深度遍历，这些东西需要移掉
        adj.get(v).remove(root);
    Set<Integer> vis = new HashSet<>();
    Set<Set<Integer>> group = new HashSet<>(); // 以每个节点作为根节点的子树子节点集合
    for (Integer v : nodes)
        if (!v.equals(root) && !vis.contains(v)) {
            Set<Integer> cur = new HashSet<>();
            dfs(vis, v, cur);
            group.add(cur);
        }
    int ans = lenMap.get(nodes.size()-1).size() > 1 ? 2 : 1; // 如果根节点不止一个，就可能有并行答案
    for (Set<Integer> g : group) { // 自顶向下：遍历根节点下每个节点的建树是否合法、是否唯一
        int tmp = helper(g);
        if (tmp == 0) return 0; // 不存在合法的根节点
        if (tmp == 2) ans = 2;
    }
    return ans;
}
private void dfs(Set<Integer> vis, int node, Set<Integer> cur) {
    vis.add(node);
    cur.add(node);
    for (int next : adj.get(node))
        if (!vis.contains(next))
            dfs(vis, next, cur);
}

```

## 3. 解题思路与分析

```

public int checkWays(int[][] pairs) {
    Map<Integer, Integer> cnt = new HashMap<>(); // 统计结点对中各个结点出现的次数
    Map<Integer, List<Integer>> adj = new HashMap<>();
    for (int[] pair : pairs) {
        int from = pair[0], to = pair[1];
        cnt.put(from, cnt.getOrDefault(from, 0) + 1);
        cnt.put(to, cnt.getOrDefault(to, 0) + 1);
        adj.computeIfAbsent(from, x -> new ArrayList<>()).add(to);
        adj.computeIfAbsent(to, x -> new ArrayList<>()).add(from);
    }
    List<Integer> list = new ArrayList<>(cnt.keySet()); // list of ori nodes 将结点对中的结点存储在 List 集合中
    list.sort((a, b) -> cnt.get(b) - cnt.get(a)); // 对 list 集合进行排序
    // pairs 中给出了树中所有具有祖孙关系的结点对，很显然，根节点是其他所有结点的祖先
    // 所以根节点在 pairs 出现的次数应该为总结点数-1，找不到符合这个关系的结点，那就不符合题目中构树的要求
    if (cnt.get(list.get(0)) != list.size() - 1) return 0;
    // 判断已排序后的结点集合是否有两个结点具有相同出现次数，如果存在，那么这两个结点可以互换，即为两颗树
}

```

```

int ans = 1;
for (int [] p : pairs) {
    if (cnt.get(p[0]).equals(cnt.get(p[1]))) {
        ans = 2;
        break;
    }
}
// 将所有结点的父结点置为出现结点最多的结点, 即根结点
// 在没有确定除根结点之外的其它结点真正父结点之前, 根结点就是它们的祖先
Map<Integer, Integer> farMap = new HashMap<>();
Set<Integer> set = new HashSet<>(); // 存储所有父结点
set.add(list.get(0));
for (Integer i : list) //
    farMap.put(i, list.get(0));
// 处理除最大结点数外, 按着构树规则处理其它结点
for (int i = 1; i < list.size(); ++i) {
    for (Integer s : adj.get(list.get(i)))
        // 判断当前结点是否为父结点
        if (!set.contains(s)) {
            // 如果 s 不是父结点, 那么就是当前 list.get(i) 结点的子结点
            // 在没有更新父结点之前, s 的父结点和 list.get(i) 的父结点是相同的 (父子在一条链上)
            // 如果父结点不相同, 可以理解为 s 的父结点 list.get(i) 有多个父结点, 显然是不合理的
            // 同样也可以把树理解为图, 除根结点之外, 所有结点的入度都为 1, 而上边的情况表示存在一个入度为 2 的结点
            // 显显与树的构建原理相悖
            if (farMap.get(s) != farMap.get(list.get(i)))
                return 0;
            farMap.put(s, list.get(i));
        }
    set.add(list.get(i));
}
return ans;
}

```

### 17.1.6 1766. Tree of Coprimes - Hard

There is a tree (i.e., a connected, undirected graph that has no cycles) consisting of  $n$  nodes numbered from 0 to  $n - 1$  and exactly  $n - 1$  edges. Each node has a value associated with it, and the root of the tree is node 0.

To represent this tree, you are given an integer array  $\text{nums}$  and a 2D array  $\text{edges}$ . Each  $\text{nums}[i]$  represents the  $i$ th node's value, and each  $\text{edges}[j] = [u_j, v_j]$  represents an edge between nodes  $u_j$  and  $v_j$  in the tree.

Two values  $x$  and  $y$  are coprime if  $\text{gcd}(x, y) == 1$  where  $\text{gcd}(x, y)$  is the greatest common divisor of  $x$  and  $y$ .

An ancestor of a node  $i$  is any other node on the shortest path from node  $i$  to the root. A node is not considered an ancestor of itself.

Return an array  $\text{ans}$  of size  $n$ , where  $\text{ans}[i]$  is the closest ancestor to node  $i$  such that  $\text{nums}[i]$  and  $\text{nums}[\text{ans}[i]]$  are coprime, or -1 if there is no such ancestor.

#### 1. 解题思路与分析

- 切入点和解题思路
  - 如果用蛮力检查一个节点的所有的祖先节点, 那么, 一个节点的祖先节点最多能有  $n-1$  个, 显然会超时的。
  - 一个重要的切入点是:  $\text{nums}[i] \leq 50$ 。我们不妨换一种思路: 从节点的值  $xx$  出发, 枚举满足  $1 \leq y \leq 50$  且  $\text{gcd}(x,y) = 1$  的  $yy$ , 并对每个  $yy$  找出离着节点  $ii$  最近的点, 最后再在这些点中求出离着当前点最近的点即可。这样只需检查 50 次即可。
  - 那么, 如何对于任一数字  $yy$ , 找出离当前节点  $ii$  最近的祖先节点呢? 首先可以想到的是, 离着节点  $ii$  最近的满足条件的祖先节点, 也是这些点中最深的。我们不妨对每个数字  $1 \sim 50$  维护一个栈, 并采用  $\text{dfs}$  的思路。每当我们要遍历下一个节点时, 就把当前节点的编号 (node) 和节点的深度 (level) push 到当前节点的值 ( $xx$ ) 对应的栈中。这样, 栈顶就是数字  $xx$  的、最深的节点, 也是我们之后需要的关于数字  $xx$  的最近的节点。此外, 要记得  $\text{dfs}$  完成后要将之前 push 进去的元素 pop 出来。
- 解题思路
  - 1、邻接表建立, 表示每个节点关联的节点
  - 2、准备 50 个栈, 以每个节点的数据值为基准, 栈内存储的数据为当前数据值对应的层数及节点  $i$  标识
  - 3、遍历到某个节点时, 以当前节点为基准, 满足  $\text{gcd}$  条件并且层数最深的为最优解, 也就是最近公共祖先节点
  - 4、满足  $\text{gcd}$  条件可能存在多个节点的数据值, 遍历可能的数据值里面, 离节点  $i$  最近的, 通过 level 来识别; 这里需要识别数值和 level 两重条件
  - 5、为啥取栈顶的元素呢, 因为我们压栈的时候, level 最大的总是在栈顶的, 而这里只需要相同数值里面 level 最大的即可, 因为每轮遍历实际是从根节点到当前节点的, 所以计算当前节点时, stack 里存储的应该是所有的祖先节点, 只需要在所有祖先节点里面取最近的即可

```

public int[] getCoprimes(int[] a, int[][] edges) {
    cop = new boolean [51][51];
    for (int i = 1; i < 51; i++)

```

```

        for (int j = 1; j < 51; j++) {
            if (!cop[i][j] && gcd(i, j) == 1) {
                cop[i][j] = true;
                cop[j][i] = true;
            }
        }
        int n = a.length;
        li = new ArrayList[n];
        for (int i = 0; i < n; i++) li[i] = new ArrayList<>();
        for (int [] e : edges) {
            li[e[0]].add(e[1]);
            li[e[1]].add(e[0]);
        }
        ans = new int [n];
        for (int i = 0; i < 51; i++)
            st[i] = new ArrayDeque<>();
        dfs(0, -1, 0, a);
        return ans;
    }
    List<Integer>[] li;
    ArrayDeque<int []>[] st = new ArrayDeque[51];
    boolean [][] cop;
    int [] ans;
    void dfs(int node, int pre, int level, int [] a) {
        int re = -1, lev = -1;
        for (int i = 1; i < 51; i++)
            if (st[i].size() > 0 && st[i].peekLast()[0] > lev && cop[i][a[node]]) {
                re = st[i].peekLast()[1];
                lev = st[i].peekLast()[0];
            }
        ans[node] = re;
        for (int next : li[node]) {
            if (next != pre) {
                st[a[node]].offerLast(new int [] {level, node});
                dfs(next, node, level + 1, a);
                st[a[node]].pollLast();
            }
        }
    }
    int gcd(int x, int y) {
        if (y == 0) return x;
        return gcd(y, x % y);
    }
}

```

### 17.1.7 1028. Recover a Tree From Preorder Traversal: 栈 + 迭代，递归 - Hard

We run a preorder depth-first search (DFS) on the root of a binary tree.

At each node in this traversal, we output D dashes (where D is the depth of this node), then we output the value of this node. If the depth of a node is D, the depth of its immediate child is D + 1. The depth of the root node is 0.

If a node has only one child, that child is guaranteed to be the left child.

Given the output traversal of this traversal, recover the tree and return its root.

#### 1. 解题思路与分析: 栈 + 迭代

```

public TreeNode recoverFromPreorder(String t) {
    Deque<TreeNode> st = new LinkedList<TreeNode>();
    char [] s = t.toCharArray();
    int n = t.length();
    int idx = 0;
    while (idx < n) {
        int lvl = 0;
        while (s[idx] == '-') {
            ++lvl;
            ++idx;
        }
        int val = 0;
        while (idx < n && Character.isDigit(s[idx])) {
            val = val * 10 + (s[idx] - '0');
            ++idx;
        }
        TreeNode node = new TreeNode(val);
        if (lvl == st.size()) {
            if (!st.isEmpty())
                st.peekLast().left = node;
        } else {
            while (lvl != st.size())
                st.pollLast();
            st.peekLast().right = node;
        }
        st.offerLast(node);
    }
    while (st.size() > 1) st.pollLast();
    return st.peekLast();
}

```

#### 2. 解题思路与分析: 递归

虽然博主最开始想的递归方法不太容易实现，但其实这道题也是可以用递归来做的，这里我们需要一个全局变量 cur，表示当前遍历字符串 S 的位置，递归函数还要传递个当前的深度 level。在递归函数中，首先还是要提取短杠的个数，但是这里有个很 tricky 的地方，我们在统计短杠个数的时候，不能更新 cur，因为 cur 是个全局变量，当统计出来的短杠个数跟当前的深度不相同，就不能再继续处理了，如果此时更新了 cur，而没有正确的复原的话，就会出错。博主成功入坑，检查了好久才找出原因。当短杠个数跟当前深度相同时，我们继续提取出结点值，然后新建出结点，对下一层分别调用递归函数赋给新建结点的左右子结点，最后返回该新建结点即可

```

private int idx = 0; // 遍历 S 的全局指针
public TreeNode recoverFromPreorder(String S) {
    if (S.isEmpty()) return null;
    return buildBinaryTree(S.toCharArray(), 0);
}
public TreeNode buildBinaryTree(char[] ss, int depth) {
    // 判断当前节点是否是 null
    if (idx + depth >= ss.length || isNullPointer(ss, depth)) return null;
    idx += depth; // idx 指针跳过 depth 个 '-'，指向下一个节点的开始位置
    // 左右子树递归
    TreeNode root = new TreeNode(getValue(ss));
    root.left = buildBinaryTree(ss, depth + 1);
    root.right = buildBinaryTree(ss, depth + 1);
    // 返回当前节点
    return root;
}
// 获取当前节点的 val 值，由于可能有多位，需要遍历一下
public int getValue(char[] ss) {
    int value = 0;
    while (idx < ss.length && ss[idx] != '-') {
        value = value * 10 + (ss[idx] - '0');
        idx++;
    }
    return value;
}
// 判断当前位置的节点是不是 null
public boolean isNullPointer(char[] ss, int depth) {
    for (int i = idx; i < idx + depth; i++)
        if (ss[i] != '-') return true;
    return false;
}

```

- 下面是一个简洁版的代码

```

public TreeNode recoverFromPreorder(String S) {
    if (S.isEmpty()) return null;
    n = S.length();
    return buildBinaryTree(S.toCharArray(), 0);
}
private int idx = 0, n; // 遍历 S 的全局指针
TreeNode buildBinaryTree(char[] s, int level) {
    int cnt = 0, val = 0;
    while (idx + cnt < n && s[idx + cnt] == '-') ++cnt;
    if (cnt != level) return null;
    idx += cnt;
    for (; idx < n && s[idx] != '-'; idx++)
        val = val * 10 + s[idx] - '0';
    TreeNode r = new TreeNode(val);
    r.left = buildBinaryTree(s, level + 1);
    r.right = buildBinaryTree(s, level + 1);
    return r;
}

```

### 17.1.8 1932. Merge BSTs to Create Single BST

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大树，方法很传神
    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 null
    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);
}

```

### 17.1.9 687. 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。

```

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;
}

```

### 17.1.10 652. 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.

```

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;
}

```

- 看一下构造的效果图

```

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

map.size(): 4
3-2-4-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, " " + 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-99.5-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

```

### 17.1.11 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));
}

```

### 17.1.12 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  $\text{nums}[j]$  for each index j you visited in the array. Return the maximum score you can get.

```

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];
}

```

### 17.1.13 1345. Jump Game IV - Hard

Given an array of integers arr, you are initially positioned at the first index of the array.

In one step you can jump from index i to index:

```
i + 1 where: i + 1 < arr.length.
i - 1 where: i - 1 >= 0.
```

j where: arr[i] = arr[j] and i != j.

Return the minimum number of steps to reach the last index of the array.

Notice that you can not jump outside of the array at any time.

#### 1. 解题思路与分析

- 首先题目给出了起点和终点，分别是数组的头部和尾部，另外，每次跳跃我们可以跳向相邻的左右 2 点以及与当前数值相同的所有点。描述到这里，题目的图形结构已经非常清晰，这实际上是一道，在已知起点和终点的情况下，求图中最短路径的问题。如果你经常看我的博客，你会马上想到，求最短路径的首选应该是 bfs，某些情况下 dfs 也是可行的。
- 接下来看解题步骤，既然是图型题，我们需要先将图构建出来，比较重要的部分应该是数组中值相同的部分，我们定义一个 Map，key 是数值，value 是具有该数值的数组下标集合。另外这里有一处可以优化的地方，比如数组中有一连串的相同数字：

```
arr = [11,22,7,7,7,7,7,7,22,13]
```

- 对于数组中连续的数字 7，实际上起作用的只有首尾两个，其他 7 无论如何跳都不会优于两边的两个 7 的。因此，当遇上连续相同数字时，我们只在 map 中保存首尾 2 个即可。图形结构构建好之后，就是标准的 bfs 解题逻辑
- 这就是个 BFS 的题，唯一注意的是：如果 left, current, right 都是同一个数，那么 HashMap<Integer, List<Integer>> 又要重新访问一遍，那么解决办法就是访问过当前 node 的所有 index 之后，立刻清零；这样每个 index 只访问一遍；O(N)
- 自己写的臭长的代码

```

public int minJumps(int[] a) {
    int n = a.length;
    if (n == 1) return 0;
    boolean[] vis = new boolean[n];
    Map<Integer, List<Integer>> m = new HashMap<>();
    for (int i = 0; i < n; i++) {
        if (i-1 >= 0 && a[i-1] == a[i] && i+1 < n && a[i+1] == a[i]) { // 任何一端的相等元素都可以 cover 当前元素，直接跳过
            vis[i] = true;
            continue;
        }
        m.computeIfAbsent(a[i], z -> new ArrayList<>()).add(i);
    }
    Deque<Integer> q = new ArrayDeque<>();
    Set<Integer> sc = new HashSet<>(); // set of current
    Set<Integer> sn = new HashSet<>(); // set of next
    sc.add(0);
    int cnt = 0;
    while (sc.size() > 0) {
        for (int v : sc) q.offerLast(v);
        while (!q.isEmpty()) {
            int cur = q.pollFirst();
            if (cur == n-1) return cnt;
            vis[cur] = true;
            if (cur < n-1 && !vis[cur+1]) sn.add(cur+1);
            if (cur > 0 && !vis[cur-1]) sn.add(cur-1);
            for (int idx : m.get(a[cur])) {
                if (vis[idx] || idx == cur) continue;
                if (idx == n-1) return cnt + 1;
                sn.add(idx);
            }
            m.put(a[cur], new ArrayList<>()); // 每个相同数值只处理一次进队列操作
        }
        sc.clear();
        sc.addAll(sn);
        sn.clear();
        cnt++;
    }
    return -1;
}

```

- 再看一下别人逻辑清晰的代码

```

public int minJumps(int [] a) { // 思路简洁：比上面的方法快了很多
    int n = a.length;
    Map<Integer, List<Integer>> m = new HashMap<>();
    for (int i = 0; i < n; i++) {
        m.computeIfAbsent(a[i], z -> new ArrayList<>()).add(i);
    }
    int cnt = 0;
    boolean [] vis = new boolean [n];
    Deque<Integer> q = new ArrayDeque<>();
    q.offerLast(0);
    vis[0] = true;
    while (!q.isEmpty()) {
        for (int z = q.size()-1; z >= 0; z--) {
            int cur = q.pollFirst();
            if (cur == n-1) return cnt;
            for (int idx : m.get(a[cur])) {
                if (idx != cur && !vis[idx]) {
                    q.offerLast(idx);
                    vis[idx] = true;
                }
            }
            if (cur-1 >= 0 && !vis[cur-1]) {
                q.offerLast(cur-1);
                vis[cur-1] = true;
            }
            if (cur+1 < n && !vis[cur+1]) {
                q.offerLast(cur+1);
                vis[cur+1] = true;
            }
        }
        m.put(a[cur], new ArrayList<>()); // 清零操作：每个相同数值只做入队列操作一次
    }
    cnt++;
}
return -1;
}

```

### 17.1.14 968. 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。
// 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;
}

```

### 17.1.15 112. Path Sum

Given the root of a binary tree and an integer targetSum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals targetSum.

A leaf is a node with no children.

```

int cnt = 0; // 用一个全局变量
public boolean hasPathSum(TreeNode r, int t) {
    if (r == null) return false;
    cnt += r.val;
    if (r.left == null && r.right == null) // {
        if (cnt == t) return true;
        // return false; // 这一句可以不要，是因为还可以再往下遍历一层，会自动返回 false
    }
    boolean left = hasPathSum(r.left, t);

```

```
boolean right = hasPathSum(r.right, t);
cnt -= r.val;
return left || right;
}
// 下面是太习惯用两个方法来写，以至于用一个方法写，不太适应
public boolean hasPathSum(TreeNode r, int t) {
    traversal(r, 0, t);
    return vis;
}
boolean vis = false;
void traversal(TreeNode r, int v, int t) {
    if (r == null) return ;
    if (r.left == null && r.right == null) {
        if (v + r.val == t) vis = true;
        return ;
    }
    traversal(r.left, v + r.val, t);
    traversal(r.right, v + r.val, t);
}
```



# Chapter 18

## Segment Tree 与 Binary Index Tree 线段树与 树状数组

- 【要解决的问题】快速区间查找:  $O(\log N)$  线段树主要用于高效解决连续区间的动态查询问题, 由于二叉结构的特性, 使用线段树可以快速的查找某一个节点在若干条线段中出现的次数, 时间复杂度为  $O(\log N)$ 。
- 【其它操作效率】对于树状数组, 线段树进行更新 (update) 的操作为  $O(\log n)$ , 进行区间查询 (range query) 的操作也为  $O(\log n)$ 。
- 【缺点】空间占用大 而未优化的空间复杂度为  $2N$ , 因此有时需要离散化让空间压缩。
- 【与树状数组的区别】与树状数组不同的是, 线段树不止可以适用于【区间求和的查询】, 也可以进行【区间最大值, 区间最小值 (Range Minimum/Maximum Query problem) 或者区间异或值的查询】。
- 【分类】就目前的理解, 暂分为两类, 一类求和的, 一类求最大最小值、或是位操作值 ( $\mid \&$ ) 等等。权值线段树是属于哪里的?

### 18.1 求和 Sum 的线段树

#### 18.1.1 327. Count of Range Sum - Hard 重点这几个题要好好再理解消化几遍

Given an integer array `nums` and two integers `lower` and `upper`, return the number of range sums that lie in `[lower, upper]` inclusive.

Range sum  $S(i, j)$  is defined as the sum of the elements in `nums` between indices  $i$  and  $j$  inclusive, where  $i \leq j$ .

1. 解题思路与分析: 分治法, 自底向上的解决问题

## 方法一：归并排序

### 思路与算法

设前缀和数组为  $preSum$ , 则问题等价于求所有的下标对  $(i, j)$ , 满足

$$preSum[j] - preSum[i] \in [lower, upper]$$

我们先考虑如下的问题：给定两个升序排列的数组  $n_1, n_2$ , 试找出所有的下标对  $(i, j)$ , 满足

$$n_2[j] - n_1[i] \in [lower, upper]$$

在已知两个数组均为升序的情况下, 这一问题是相对简单的: 我们在  $n_2$  中维护两个指针  $l, r$ 。起初, 它们都指向  $n_2$  的起始位置。

随后, 我们考察  $n_1$  的第一个元素。首先, 不断地将指针  $l$  向右移动, 直到  $n_2[l] \geq n_1[0] + lower$  为止, 此时,  $l$  及其右边的元素均大于或等于  $n_1[0] + lower$ ; 随后, 再不断地将指针  $r$  向右移动, 直到  $n_2[r] > n_1[0] + upper$  为止, 则  $r$  左边的元素均小于或等于  $n_1[0] + upper$ 。故区间  $[l, r)$  中的所有下标  $j$ , 都满足

$$n_2[j] - n_1[0] \in [lower, upper]$$

接下来, 我们考察  $n_1$  的第二个元素。由于  $n_1$  是递增的, 不难发现  $l, r$  只可能向右移动。因此, 我们不断地进行上述过程, 并对于  $n_1$  中的每一个下标, 都记录相应的区间  $[l, r)$  的大小。最终, 我们就统计得到了满足条件的下标对  $(i, j)$  的数量。

在解决这一问题后, 原问题就迎刃而解了: 我们采用归并排序的方式, 能够得到左右两个数组排序后的形式, 以及对应的下标对数量。对于原数组而言, 若要找出全部的下标对数量, 只需要再额外找出左端点在左侧数组, 同时右端点在右侧数组的下标对数量, 而这正是我们此前讨论的问题。

- 下面是最初级的归并排序的解法与写法

```
// 【最基本的数据结构的解法】: 归并排序。整个过程是一个自底向上, 不断求值与归并的过程
public int countRangeSum(int[] a, int lo, int hi) {
    int n = a.length;
    long[] s = new long[n+1]; // 用来求和 prefixSum
    for (int i = 0; i < n; i++) s[i+1] = s[i] + a[i]; // 不一定是: 升序排列
    return countRangeSumRecursive(s, lo, hi, 0, n);
}
int countRangeSumRecursive(long[] sum, int lo, int hi, int l, int r) { // l: 左下标, r: 右下标
    if (l == r) return 0;
    int m = (l + r) / 2;
    // 【首先, 递归, 分别解决左右半部分的问题】: 分别解决了左右部分之后, 左右部分分别是有序排列的片段
    int n1 = countRangeSumRecursive(sum, lo, hi, l, m);
    int n2 = countRangeSumRecursive(sum, lo, hi, m+1, r);
    int ans = n1 + n2;
    // 【再来解决归并相关】
    // 首先统计下标对的数量
    int i = l, left = m+1, right = m+1;
    while (i <= m) {
        while (left <= r && sum[left] - sum[i] < lo) left++; // 左边界右移, 直到达标【lo, ...】
        right = left; // 可要可不要, 要了可以少遍历上面的过程...
        while (right <= r && sum[right] - sum[i] <= hi) right++; // 右边界右移, 直到不达标越界... hi-1】hi...
        ans += right - left;
        i++;
    }
    // 随后合并两个排序数组
    long[] sorted = new long[r - l + 1];
    int x = l, y = m+1, z = 0; // x,y,z: 分别为左右两个片段的遍历下标, 以及合并数组的遍历下标
    while (x <= m || y <= r)
        if (x > m) sorted[z++] = sum[y++];
        else if (y > r) sorted[z++] = sum[x++];
        else if (sum[x] < sum[y]) sorted[z++] = sum[x++];
        else sorted[z++] = sum[y++];
    // 再把这个排序好的数组, 更新同步到累积和数组里去
    for (int j = 0; j < sorted.length; j++)
        sum[l+j] = sorted[j];
    return ans;
}
```

- 复杂度分析为:

### 复杂度分析

- 时间复杂度:  $O(N \log N)$ , 其中  $N$  为数组的长度。设执行时间为  $T(N)$ , 则两次递归调用的时间均为  $T(N/2)$ , 最后需要  $O(N)$  的时间求出下标对数量以及合并数组, 故有

$$T(N) = 2 \cdot T(N/2) + O(N)$$

根据主定理, 有  $T(N) = O(N \log N)$ 。

- 空间复杂度:  $O(N)$ 。设空间占用为  $M(N)$ , 递归调用所需空间为  $M(N/2)$ , 而合并数组所需空间为  $O(N)$ , 故

$$M(N) = \max \{M(N/2), O(N)\} = M(N/2) + O(N)$$

根据主定理, 有  $M(N) = O(N)$ 。

- 下面是一个代码更为简洁的写法, 排序的步骤本地用语言自带的排序法

```
public int countRangeSum(int[] a, int lower, int upper) { // 这个 merge sort 的思维很奇特: 二分, O(NlogN)
    long [] sum = new long[a.length+1];
    for (int i = 0; i < a.length; i++)
        sum[i+1] = sum[i] + a[i];
    return mergeAnalyse(sum, 0, a.length+1, lower, upper);
}
int mergeAnalyse(long [] a, int l, int r, int lo, int hi) { // l, r: 寻找【l, r】范围内和为【lower, upper】的片段的个数
    if (r - l <= 1) return 0;
    int m = l + (r - l) / 2;
    // int mid = l + (r - l) / 2;
    // int m = mid, n = mid, ans = 0;
    int ans = mergeAnalyse(a, l, m, lo, hi) + mergeAnalyse(a, m, r, lo, hi);
    int x = m, y = m;
    for (int i = l; i < m; i++) { // 遍历【l, r】的半段长度: pivot 右移, 滑动窗口, 寻找合法窗口 // 通过遍历寻找当前范围内符合要求的个数,
        while (x < r && a[x] - a[i] < lo) x++; // 左端点右移, 直到找到合法 (sum >= lo) 的解: m 合法
        y = x; // 可要可不要...
        while (y < r && a[y] - a[i] <= hi) y++; // 右端点右移, 直到右端点右移至不再合法 (sum > hi), n 不合法
        ans += y - x; // 对于【l, r】范围内的当前 i 来说, 满足要求的总个数为 n - m
    }
    Arrays.sort(a, l, r); // 将【l, r】片段排序, 本地排序
    return ans;
}
```

## 2. 解题思路与分析: 求和 sum 线段树 + 离散化数据的连续化处理

### 方法二: 线段树

#### 思路与算法

依然考虑前缀和数组  $preSum$ 。

对于每个下标  $j$ , 以  $j$  为右端点的下标对的数量, 就等于数组  $preSum[0..j - 1]$  中的所有整数, 出现在区间  $[preSum[j] - upper, preSum[j] - lower]$  的次数。故很容易想到基于线段树的解法。

我们从左到右扫描前缀和数组。每遇到一个数  $preSum[j]$ , 我们就在线段树中查询区间  $[preSum[j] - upper, preSum[j] - lower]$  内的整数数量, 随后, 将  $preSum[j]$  插入到线段树当中。

注意到整数的范围可能很大, 故需要利用哈希表将所有可能出现的整数, 映射到连续的整数区间内。

```
// 最基本的【求和 sum 线段树】: 将前缀和数值中的取值范围, 分布到 Bit 的数组上来, 可能需要必要的取值偏移【不是偏移, 用个策略!!!】, 以保证 bit 每个元素取值 >= 1
// 这里还是来参考别人更为精典的解决办法
public int countRangeSum(int[] a, int lo, int hi) {
    int n = a.length;
    long [] s = new long [n+1];
    for (int i = 0; i < n; i++) s[i+1] = s[i] + a[i];
    // Set<Long> allNumbers = new HashSet<>(); // 考量: 去重作用【BUGGLY CODING:】没有考虑到这里的排序作用...
    Set<Long> allNumbers = new TreeSet<>(); // 考量: 去重作用【BUGGLY CODING:】没有考虑到这里的排序作用...【狠狠狠重要! 否则不成解!!!】
    for (long v : s) {
        allNumbers.add(v);
        allNumbers.add(v - lo);
        allNumbers.add(v - hi);
    }
    // 利用哈希表进行离散化: 这里离散化的本质是: 将离散的取值, 重新序列化为【0, n-1】下标的值序列片段!!!
    // 注意到整数的范围可能很大, 故需要利用哈希表将所有可能出现的整数, 映射到连续的整数区间内。
}
```

```

// 【前面的离散长整形数据，如果不排列，映射到连续片段的下标计数，没有意义。。。】
Map<Long, Integer> values = new HashMap<>();
int idx = 0;
for (long v : allNumbers)
    values.put(v, idx++); // 【只有 allNumbers 排序过】：映射过来的 idx 的取值才能正确反映离散值在线段树中所处的正确位置...
Node root = build(0, values.size()-1);
int ans = 0;
for (long v : s) {
    int l = values.get(v - hi), r = values.get(v - lo);
    ans += count(root, l, r); // 【理解困难的地方】第一次的数个数，是什么时候，这个时候调用，会更新哪些？
    insert(root, values.get(v)); // 这里就添加了【0, n-1】下标范围内的某个下标，把离散的值连续化到一个有效片段，最小区间求和 sum 线段树
}
return ans;
}

// 求和 sum 线段树：三个最基本的方法：构建线段树，更新（插入一个值）线段树，查询区间内的个数
public Node build(int lo, int hi) { // 【lo, hi】：问题是，创建树的时候，没有，不曾数过、更新过每个区间的元素个数和 ????
    Node r = new Node(lo, hi);
    if (lo == hi) return r;
    int m = (hi + lo) / 2;
    r.left = build(lo, m);
    r.right = build(m+1, hi);
    // r.s = r.left.s + r.right.s; // 父节点的个数 = 左右子树节点个数的和【为什么它这里没有更新？】
    return r;
}

public int count(Node r, int left, int right) { // 求和线段树：是否与 BST 一样，右边节点计数大于左边与根节点呢？
    // 因为下面这一行的处理：区间外完全不用考虑，返回 0
    if (left > r.l || right < r.l) return 0; // 查询区间，当前区间节点，完全不用考虑
    if (left <= r.l && r.r <= right) return r.s; // 【我是什么时候，才来更新这个计数 s 的？】
    // 所以，这里就可以直接调用，递归左右子树，求计数和
    return count(r.left, left, right) + count(r.right, left, right);
}

public void insert(Node r, int v) { // 这里可以理解为：动态更新，过程中随机增加一个元素
    r.s++; // 这个时候，才知道，计数，计算区间内元素个数，原来是如此精妙地完成的...
    if (r.l == r.r) return; // 叶子节点
    int m = (r.l + r.r) / 2;
    if (v <= m) insert(r.left, v);
    else insert(r.right, v);
}

class Node {
    int l, r, s; // 区间值的范围【l, r】
    Node left, right;
    public Node(int left, int right) {
        l = left;
        r = right;
        s = 0;
        this.left = null;
        this.right = null;
    }
}
}

```

- 线段树：方法的复杂度分析

### 复杂度分析

- 时间复杂度： $O(N \log N)$ 。使用哈希离散化之后，线段树维护的区间大小为  $O(N)$ ，故其深度、单次查询或插入的时间复杂度均为  $O(\log N)$ 。而离散化本身的复杂度也为  $O(N \log N)$ 。
- 空间复杂度： $O(N)$ 。线段树的深度为  $O(N)$ ，而第  $i$  层拥有的节点数量为  $2^{i-1}$ ，故线段树总的节点数量为  $2^{O(\log N)} = O(N)$ 。

### 3. 解题思路与分析：动态增加节点的线段树

#### 方法三：动态增加节点的线段树

##### 思路与算法

与方法二类似，但我们可以不实用哈希表进行映射，而是只在线段树的插入操作过程中动态地增加树中的节点。而当我们进行查询操作时，如果到达一个空节点，那么说明对应的区间中暂时还没有值，就可以直接返回 0。

```

class Node {
    long l, r; // 区间值的范围【l, r】
    int s;
    Node left, right;
    public Node(long left, long right) {
        l = left;
        r = right;
        s = 0;
        this.left = null;
        this.right = null;
    }
}

```

```

    }
}

public int countRangeSum(int[] a, int lo, int hi) {
    int n = a.length;
    long [] s = new long [n+1];
    for (int i = 0; i < n; i++) s[i+1] = s[i] + a[i];
    // 可以不使用哈希表进行映射，而是只在线段树的插入操作过程中动态地增加树中的节点。
    // 而当我们进行查询操作时，如果到达一个空节点，那么说明对应的区间中暂时还没有值，就可以直接返回 0
    long lowrBound = Long.MAX_VALUE, highBound = Long.MIN_VALUE;
    for (long v : s) {
        lowrBound = Math.min(Math.min(lowrBound, v), Math.min(v - lo, v - hi));
        highBound = Math.max(Math.max(highBound, v), Math.max(v - lo, v - hi));
    }
    Node root = new Node(lowrBound, highBound);
    int ans = 0;
    for (long v : s) {
        ans += count(root, v - hi, v - lo); // 【理解困难的地方】第一次的数个数，是什么时候，这个时候调用，会更新哪些？
        insert(root, v); // 这里就添加了【0,n-1】下标范围内的某个下标，把离散的值连续化到一个有效片段，最小区间求和 sum 线段树
    }
    return ans;
}

// 求和 sum 线段树：这里精简成了，两个方法。。。因为不曾分步构建过树，所以必要的时候，必须先判断是否为空，添加节点
public long count(Node r, long left, long right) { // 求和线段树：是否与 BST 一样，右边节点计数大于左边与根节点呢？
    if (r == null) return 0;
    // 因为下面这一行的处理：区间外完全不用考虑，返回 0
    if (left > r.r || right < r.l) return 0; // 查询区间，当前区间节点，完全不用考虑
    if (left <= r.l && r.r <= right) return r.s; // 【我是什么时候，才来更新这个计数 s 的？】
    // 所以，这里就可以直接调用，递归左右子数，求计数和
    return count(r.left, left, right) + count(r.right, left, right);
}

public void insert(Node r, long v) { // 这里可以理解为：动态更新，过程中随机增加一个元素
    r.s++; // 这个时候，才知道，计数，计算区间内元素个数，原来是如此精妙地完成的...
    if (r.l == r.r) return; // 叶子节点
    // int m = (r.l + r.r) / 2;
    long m = (r.l + r.r) >> 1;
    if (v <= m) {
        if (r.left == null)
            r.left = new Node(r.l, m);
        insert(r.left, v);
    } else {
        if (r.right == null)
            r.right = new Node(m+1, r.r);
        insert(r.right, v);
    }
}
}

```

- 复杂度分析：

### 复杂度分析

- 时间复杂度： $O(N \log C)$ ，其中  $C$  是线段树根节点对应的区间长度。由于我们使用 64 位整数类型进行存储，因此  $\log C$  不会超过 64。使用动态增加节点的线段树，单次查询或插入的时间复杂度均为  $O(\log C)$ 。
- 空间复杂度： $O(N \log C)$ 。需要进行  $N$  次线段树的插入操作，每次会添加不超过  $\log C$  个新节点。

## 4. 解题思路与分析：树状数组

### 方法四：树状数组

#### 思路与算法

树状数组与线段树基于类似的思想，不过树状数组支持的基本查询为求出  $[0, val]$  之间的整数数量。为了查询区间  $[preSum[j] - upper, preSum[j] - lower]$  内的整数数量，需要执行两次查询，即分别查询  $[0, preSum[j] - upper - 1]$  区间的整数数量  $L$  和  $[0, preSum[j] - lower]$  区间的整数数量  $R$ ，答案即为两者作差  $R - L$ 。

```

// 【方法四】树状数组。好像我前面没能区分，线段树，与树状数组的区别？
public int countRangeSum(int[] a, int lo, int hi) {
    int n = a.length;
    long [] s = new long [n+1];
    for (int i = 0; i < n; i++) s[i+1] = s[i] + a[i];
    Set<Long> allNumbers = new TreeSet<>();
    for (long v : s) {
        allNumbers.add(v);
        allNumbers.add(v - lo);
        allNumbers.add(v - hi);
    }
    // 利用哈希表进行离散化
    Map<Long, Integer> values = new HashMap<Long, Integer>();

```

```

int idx = 0;
for (long v : allNumbers)
    values.put(v, idx++);
int ans = 0;
BIT bit = new BIT(values.size());
for (int i = 0; i < s.length; i++) {
    int left = values.get(s[i] - hi), right = values.get(s[i] - lo);
    ans += bit.query(right + 1) - bit.query(left);
    bit.update(values.get(s[i]) + 1, 1);
}
return ans;
}
class BIT {
    int [] tree;
    int n;
    public BIT(int n) {
        this.n = n;
        this.tree = new int[n+1];
    }
    public static int lowbit(int x) {
        return x & (-x);
    }
    public void update(int idx, int d) {
        while (idx <= n) {
            tree[idx] += d;
            idx += lowbit(idx);
        }
    }
    public int query(int x) {
        int ans = 0;
        while (x != 0) {
            ans += tree[x];
            x -= lowbit(x);
        }
        return ans;
    }
}
}

```

### 复杂度分析

- 时间复杂度:  $O(N \log N)$ 。离散化本身的复杂度为  $O(N \log N)$ , 而树状数组单次更新或查询的复杂度为  $O(\log N)$ 。
- 空间复杂度:  $O(N)$ 。

## 5. 解题思路与分析: 平衡二叉搜索树

### 方法五：平衡二叉搜索树

#### 思路与算法

考虑一棵平衡二叉搜索树。若其节点数量为  $N$ , 则深度为  $O(\log N)$ 。二叉搜索树能够在  $O(\log N)$  的时间内, 对任意给定的值  $val$ , 查询树中所有小于或等于该值的数量。

因此, 我们可以从左到右扫描前缀和数组。对于  $preSum[j]$  而言, 首先进行两次查询, 得到区间  $[preSum[j] - upper, preSum[j] - lower]$  内的整数数量; 随后再将  $preSum[j]$  插入到平衡树中。

平衡二叉搜索树有多种不同的实现, 最经典的为 AVL 树与红黑树。此外, 在算法竞赛中, 还包括 Treap、SBT 等数据结构。

下面给出基于 Treap 的实现。

```

// 【方法五：平衡二叉搜索树】
public int countRangeSum(int[] a, int lo, int hi) {
    long [] s = new long [a.length+1];
    for (int i = 0; i < a.length; i++) s[i+1] = s[i] + a[i];
    BT tr = new BT();
    int ans = 0;
    for (long v : s) {
        long numLeft = tr.lowerBound(v - hi);
        int rankLeft = (numLeft == Long.MAX_VALUE ? (int)(tr.getSize()+1) : tr.rank(numLeft)[0]);
        long numRight = tr.upperBound(v - lo);
        int rankRight = (numRight == Long.MAX_VALUE ? (int)tr.getSize() : tr.rank(numRight)[0]-1);
        ans += rankRight - rankLeft + 1;
        tr.insert(v);
    }
    return ans;
}

```

```

class BT { // Treap = Tree+Heap:
    private class Node {
        long v, s;
        int cnt, size;
        Node l, r;
        Node(long val, long seed) {
            v = val;
            s = seed; // 为什么要这个种子？伪随机数吗？【Treap 的修正值】：修正值满足最小堆性质
            cnt = 1;
            size = 1;
            l = null; r = null;
        }
        //      this      r <= root
        // /   \   /   \
        // l   r   this   r.r(root.r)
        //           /   \
        //           l   r.l(root.l)
    }
    Node leftRotate() { // 左旋：当前根 this 变成左子节点；先前右变成根
        int prevSize = size;
        int currSize = (l != null ? l.size : 0) + (r.l != null ? r.l.size : 0) + cnt; // 左右子树的 size + 当前根节点的 cnt
        Node root = r; // 这里先把 root 当作 r 的另一个索引指针
        r = root.l;
        root.l = this;
        root.size = prevSize; // 【没看明白】是怎么变过来的？
        size = currSize;
        return root;
    }
    //      this      l <= root
    // /   \   /   \
    // l   r   l.l   this
    // /   \   /   \
    // l.l  l.r   l.r   r
    Node rightRotate() {
        int prevSize = size;
        int currSize = (r != null ? r.size : 0) + (l.r != null ? l.r.size : 0) + cnt;
        Node root = l;
        l = root.r;
        root.r = this;
        root.size = prevSize; // 【没看明白】是怎么变过来的？
        size = currSize;
        return root;
    }
}
private Node root;
private int size;
private Random rand;
public BT() {
    this.root = null;
    this.size = 0;
    this.rand = new Random();
}
public long getSize() {
    return size;
}
public void insert(long v) {
    ++size;
    root = insert(root, v);
}
public long lowerBound(long v) { // 这是找，最小的一个不小于 v 【>= v】 的值吗？
    Node r = root;
    long ans = Long.MAX_VALUE;
    while (r != null) {
        if (v == r.v) return v;
        if (v < r.v) {
            ans = r.v;
            r = r.l;
        } else r = r.r;
    }
    return ans;
}
public long upperBound(long v) { // 找一个最大的 【<= v】 的值
    Node r = root;
    long ans = Long.MAX_VALUE;
    while (r != null) {
        if (v < r.v) {
            ans = r.v;
            r = r.l;
        } else r = r.r;
    }
    return ans;
}
public int [] rank(long v) {
    Node r = root;
    int ans = 0;
    while (r != null) {
        if (v < r.v) r = r.l;
        else { // v >= r.v
            ans += (r.l != null ? r.l.size : 0) + r.cnt;
            if (v == r.v)

```

```

        return new int [] {ans - r.cnt + 1, ans};
    }
}
return new int [] {Integer.MIN_VALUE, Integer.MAX_VALUE};
}
private Node insert(Node r, long v) {
    if (r == null) return new Node(v, rand.nextInt());
    ++r.size;
    if (v < r.v) { // 左子树
        r.l = insert(r.l, v);
        if (r.l.s > r.s) // 这里有步检查是否平衡的步骤 ?
            r = r.rightRotate();
    } else if (v > r.v) { // 右子树
        r.r = insert(r.r, v);
        if (r.r.s > r.s)
            r = r.leftRotate();
    } else ++r.cnt; // 当前根节点
    return r;
}
}

```

- 复杂度分析

- 时间复杂度:  $O(N \log N)$ 。

- 空间复杂度:  $O(N)$ 。

- 这里简单介绍一下 Treap 这个数据结构，因为最易编程，被广泛使用，应该掌握。

### (1) Treap = Tree + Heap

**Treap** 是一种平衡树。Treap 发音为 [tri:p]。这个单词的构造选取了 Tree(树)的前两个字符和 Heap(堆)的后三个字符, Treap = Tree + Heap。顾名思义, Treap 把 BST 和 Heap 结合了起来。它和 BST 一样满足许多优美的性质, 而引入堆目的就是为了维护平衡。

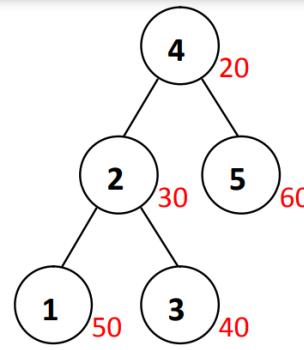


图 5

Treap 在 BST 的基础上, 添加了一个修正值。在满足 BST 性质的基础上, Treap 节点的修正值还满足最小堆性质<sup>2</sup>。最小堆性质可以被描述为每个子树根节点都小于等于其子节点。于是, Treap 可以定义为有以下性质的二叉树:

1. 若它的左子树不空, 则左子树上所有结点的值均小于它的根结点的值, 而且它的根节点的修正值小于等于左子树根节点的修正值;
2. 若它的右子树不空, 则右子树上所有结点的值均大于它的根结点的值, 而且它的根节点的修正值小于等于右子树根节点的修正值;
3. 它的左、右子树也分别为 Treap。

图 5 为一个 Treap。

修正值是节点在插入到 Treap 中时随机生成的一个值, 它与节点的值无关。代码 6 给出了 Treap 的一般定义。

---

```

struct Treap_Node
{
    Treap_Node *left,*right; //节点的左右子树的指针
    int value,fix; //节点的值和修正值
};
  
```

---

代码 6

- 维护平衡的原因: 修正值
  - 为什么平衡: 我们发现, BST 会遇到不平衡的原因是因为有序的数据会使查找的路径退化成链, 而随机的数据使 BST 退化的概率是非常小的。在 Treap 中, 修正值的引入恰恰是使树的结构不仅仅取决于节点的值, 还取决于修正值的值。然而修正值的值是随机生成的, 出现有序的随机序列是小概率事件, 所以 Treap 的结构是趋向于随机平衡的。

#### 18.1.2 2407. Longest Increasing Subsequence II: 【线段树】: 【贴出来方便自己查询, 解题印象深刻】活宝妹就是一定要嫁给亲爱的表哥!!!

You are given an integer array `nums` and an integer `k`.

Find the longest subsequence of `nums` that meets the following requirements:

The subsequence is strictly increasing and The difference between adjacent elements in the subsequence is at most `k`. Return the length of the longest subsequence that meets the requirements.

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

- 添加这个题目, 主要是昨天晚上写的时候, 感觉对于开闭区间, 下标等, 似乎还没有理解透彻。这个题算是比较简单, 自己基本上会写的题, 再总结一下。

在求解「上升子序列 (IS)」问题时，一般有两种优化方法：

1. 维护固定长度的 IS 的末尾元素的最小值 + 二分优化；
2. 基于值域的线段树、平衡树等数据结构优化。

这两种做法都可以用  $O(n \log n)$  的时间解决 300. 最长递增子序列。

对于本题，由于有一个差值不超过  $k$  的约束，用线段树更好处理。

具体来说，定义  $f[i][j]$  表示  $nums$  的前  $i$  个元素中，以元素  $j$ （注意不是  $nums[j]$ ）结尾的满足题目两个条件的子序列的最长长度。

当  $j \neq nums[i]$  时， $f[i][j] = f[i - 1][j]$ 。

当  $j = nums[i]$  时，我们可以从  $f[i - 1][j']$  转移过来，这里  $j - k \leq j' < j$ ，取最大值，得

$$f[i][j] = 1 + \max_{j'=j-k}^{j-1} f[i - 1][j']$$

上式有一个「区间求最大值」的过程，这非常适合用线段树计算，且由于  $f[i]$  只会从  $f[i - 1]$  转移过来，我们可以把  $f$  的第一个维度优化掉。这样我们可以用线段树表示整个  $f$  数组，在上面查询和更新。

最后答案为  $\max(f[n - 1])$ ，对应到线段树上就是根节点的值。

## 1. 线段树的标准简洁写法：

```
public int lengthOfLIS(int[] a, int k) { // 动规：+线段树来找前 f[i][v-k] 范围内的最大值
    int n = a.length, m = Arrays.stream(a).max().getAsInt();
    t = new int [4 * m]; // 线段树？下标是从 1 开始的吗？这里感觉取不到最大值【m】
    for (int v : a)
        if (v == 1) update(1, 1, m, 1, 1); // 更新单点：【v, res】成 t[1] = 1
        else {
            int res = 1 + query(1, 1, m, Math.max(1, v-k), v-1); // 查询区间：【v-k, v-1】
            update(1, 1, m, v, res); // 更新单点：【v, res】成 t[v] = res
        }
    return t[1];
}
int [] t; // 线段树：最大值线段树，下标从 1 开始的标准写法
void update(int u, int l, int r, int i, int v) { // 更新下标为 i 元素的值为 v，从 u 节点开始遍历
    if (l == r) {
        t[u] = v;
        return ;
    }
    int m = l + (r - l) / 2;
    if (i <= m) update(u << 1, l, m, i, v);
    else update(u << 1 | 1, m+1, r, i, v); // 【左右节点的下标】: U << 1 | 1
    t[u] = Math.max(t[u << 1], t[u << 1 | 1]); // 根节点最大值：取左右节点的最大值
}
// 查询【l,R】范围内的最大值，线段树的跨区间为【l,r】。l 和 R 在整个递归过程中均不变，将其大写，视作常量
int query(int u, int l, int r, int L, int R) { // 返回区间 [l,R] 内的最大值
    if (L <= l && r <= R) return t[u]; // 整个线段树，处于查询区间内，返回根节点最大值
    int m = l + (r - l) / 2, leftMax = 0, rightMax = 0;
    if (L <= m) leftMax = query(u << 1, l, m, L, R);
    if (m+1 <= R) rightMax = query(u << 1 | 1, m+1, r, L, R);
    return Math.max(leftMax, rightMax);
}
```

## 2. 线段树的【奇葩版本的】写法：

```
public int lengthOfLIS(int[] a, int k) { // 动规：+线段树来找前 f[i][v-k] 范围内的最大值【这个题仍然成了学习题】
    int n = a.length, m = Arrays.stream(a).max().getAsInt() + 1, ans = 1;
    t = new int [4 * m]; // 不是说，线段树？下标是从 1 开始的吗？最大值 m 元素在哪里
    int [][] f = new int [n][m]; // 第二维表达的是以当前数 a[i] 为结尾的最长合法子序列长度，所以取最值
    for (int i = 0; i < n; i++) { // 注意【0】下标更新线段树...
        int v = a[i];
        f[i][v] = 1;
        // 这里要找：前所有 i 个数【0,i-1】中，以【v-k,v-1】结尾的最大值，最大长度，
        // 这里我是在想要遍历，总复杂度为【O(N^2)】，线段树可以做到【O(NLogN)】线段树中的第一维就给消除掉，只累加更新【0,maxVal+1】范围内的最大值
        // for (int j = Math.max(0, v - k); j < v; j++) // 因为线段树区间求最大值：这里就不用遍历，一次【O(logN)】查询就可以了
        //     f[i][v] = Math.max(f[i][v], f[i-1][j] + 1); // 【分不清：哪个 i?】
        f[i][v] = Math.max(f[i][v], getMax(0, 0, m-1, v-k, v-1, t) + 1); // 查询线段树【v-k,v-1】区间最大值：下标 1 开始，左闭右闭区间
        // f[i][v] = Math.max(f[i][v], getMax(0, 0, n-1, v-k, v-1, t) + 1); // 查询线段树【v-k,v-1】区间最大值：下标 1 开始，左闭右闭区间
        update(0, 0, m-1, v, f[i][v], t); // 更新线段树单点元素：v 下标值为 f[i][v]
    }
}
```

```

    // update(0, 0, n-1, i, f[i][v], t); // 更新线段树单点元素: v 下标值为 f[i][v]
    // ans = Math.max(ans, f[i][v]);
}
return t[0];
}
int [] t; // 【奇葩线段树】: 下标从 0 开始的
void update(int u, int l, int r, int idx, int v, int [] t) { // 我这里参考别人的奇葩写法, 写得自己稀里糊涂的。。。重写一遍
    if (l == r) {
        t[u] = v;
        return;
    }
    int m = l + (r - l) / 2;
    if (idx <= m) update(u << 1 | 1, l, m, idx, v, t);
    else update((u << 1) + 2, m+1, r, idx, v, t);
    t[u] = Math.max(t[u << 1 | 1], t[(u << 1) + 2]); // 最大值线段树: 根节点最大值, 取左右子节点最大值
}
int getMax(int u, int l, int r, int L, int R, int [] t) { // 【l,r】: 现存线段树的有效区间跨度; 【L,R】: 查询区间跨度
    if (R < l || r < L) return 0;
    if (L <= l && r <= R) return t[u];
    int m = l + (r - l) / 2;
    int ll = getMax(u << 1 | 1, l, m, L, R, t);
    int rr = getMax((u << 1) + 2, m+1, r, L, R, t);
    return Math.max(ll, rr);
}
}

```

### 18.1.3 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:

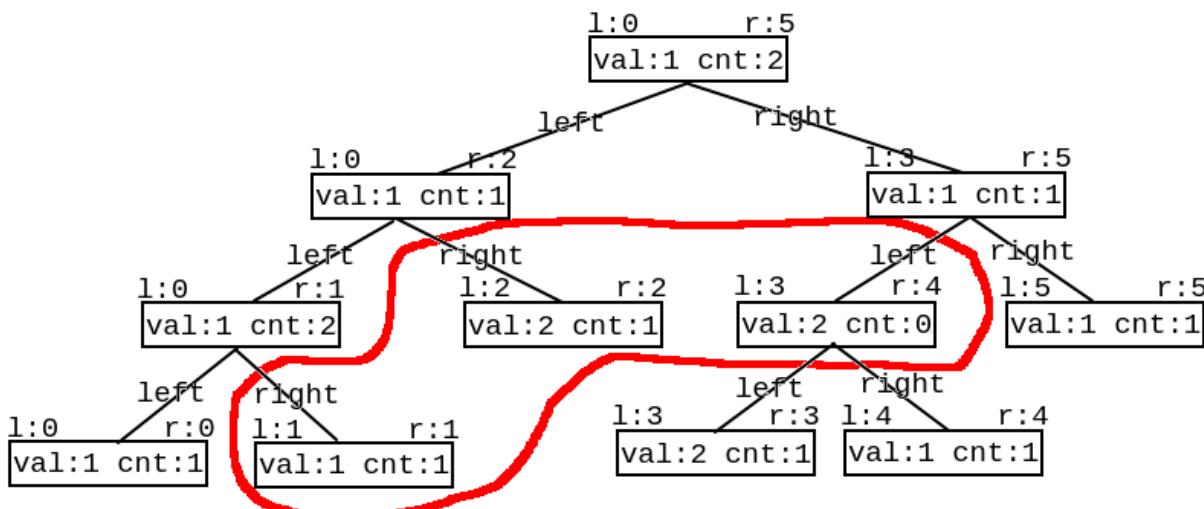
MajorityChecker(int[] arr) Initializes the instance of the class with the given array arr. int query(int left, int right, int threshold) returns the element in the subarray arr[left...right] that occurs at least threshold times, or -1 if no such element exists.

- <https://www.cnblogs.com/slowbirdoflsh/p/11381565.html> 思路比较清晰

数组: [1, 1, 2, 2, 1, 1]

0 1 2 3 4 5

使用线段树查询 query(1, 4, 3)



```

Map<Integer, List<Integer>> idx; // idx 存储数组出现元素种类 以及该元素下标索引
Node root; // 线段树的根节点
int key = 0, cnt = 0; // key 所查找的区域众数; count 所查找的区域众数出现次数,
public MajorityChecker(int[] a) {
    idx = new HashMap<>(); // idx 存储数组出现元素种类 以及该元素下标索引
    for (int i = 0; i < a.length; i++)
        idx.computeIfAbsent(a[i], z -> new ArrayList<>()).add(i);
    root = buildTree(a, 0, a.length-1);
}
public int query(int left, int right, int threshold) {
    key = 0; cnt = 0; // 初始化 所查询众数 key 及辅助判断的计数 cnt
}

```

```

searchTree(root, left, right); // 查询线段树
// 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
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;
}
int upper_bound(List<Integer> list, int v) { // 排序数组中 第一个大于 tar 的下标
int l = 0, r = list.size();
while (l < r) {
    int mid = l + (r - l) / 2;
    if (list.get(mid) <= v) l = mid + 1;
    else r = mid;
}
return l;
}
int lower_bound(List<Integer> list, int v) { // 排序数组中 第一个大于等于 tar 的下标
int l = 0, r = list.size();
while (l < r) {
    int mid = l + (r - l) / 2;
    if (list.get(mid) < v) l = mid + 1;
    else r = mid;
}
return l;
}
void searchTree(Node root, int l, int r) {
if (root == null || l > r) return ;
if (root.l > r || root.r < l) return ;
if (root.l >= l && root.r <= r) { // 当查询边界被节点边界覆盖, 该节点就是查询区域
    if (key == root.v) cnt += root.cnt;
    else if (cnt <= root.cnt) {
        key = root.v;
        cnt = root.cnt - cnt;
    } else cnt = cnt - root.cnt;
    return ;
}
int mid = (root.l + root.r) / 2; // 这两个查询条件再好好想想 !!!!!!!!!!!!!!!
if (l <= mid) // root.l <= l <= mid 左节点也可以是查询区域
    searchTree(root.left, l, r);
if (r >= mid+1) // mid+1 <= r <= root.r 右节点也可以是查询区域
    searchTree(root.right, l, r);
}
Node buildTree(int[] a, int l, int r) {
if (l > r) return null;
Node root = new Node(l, r); // 初始一个线段树的根节点
if (l == r) { // 叶子节点
    root.v = a[l];
    root.cnt = 1;
    return root;
}
int mid = (l + r) / 2;
root.left = buildTree(a, l, mid);
root.right = buildTree(a, mid+1, r);
makeRoot(root); // 整合父节点
return root;
}
void makeRoot(Node r) { // 整合父节点
if (r == null) return ;
if (r.left != null) { // 如果该节点有左子节点 该节点的值"先"等于左子节点
    r.v = r.left.v;
    r.cnt = r.left.cnt;
}
if (r.right != null) { // 如果该节点还有右子节点 融合父节点和子节点
    if (r.v == r.right.v)
        r.cnt = r.cnt + r.right.cnt;
    else {
        if (r.cnt >= r.right.cnt)
            r.cnt = r.cnt - r.right.cnt;
        else {
            r.v = r.right.v;
            r.cnt = r.right.cnt - r.cnt;
        }
    }
}
}
class Node {
int l, r, v, cnt;
Node left, right;
public Node(int l, int r) {
    this.l = l; this.r = r;
    v = 0; cnt = 0;
    left = null; right = null;
}
}

```

### 18.1.4 1825. Finding MK Average - Hard

You are given two integers,  $m$  and  $k$ , and a stream of integers. You are tasked to implement a data structure that calculates the MKAverage for the stream.

The MKAverage can be calculated using these steps:

If the number of the elements in the stream is less than  $m$  you should consider the MKAverage to be -1. Otherwise, copy the last  $m$  elements of the stream to a separate container. Remove the smallest  $k$  elements and the largest  $k$  elements from the container. Calculate the average value for the rest of the elements rounded down to the nearest integer. Implement the MKAverage class:

MKAverage(int  $m$ , int  $k$ ) Initializes the MKAverage object with an empty stream and the two integers  $m$  and  $k$ . void addElement(int num) Inserts a new element num into the stream. int calculateMKAverage() Calculates and returns the MKAverage for the current stream rounded down to the nearest integer.

```
// 根据题意需要找到前 k 大的数，又需要求区间和，就自然想到线段树。写起来较不容易出错。
// 维护 2 个线段树数组，一个记录数的个数，一个记录区间值。
// 注意一般线段树中 [s, e] 指固定的区间，这里类似线段数求第 k 小的数，所以 [s, e] 指第 s 小的值到第 e 小的值的区间。
Deque<Integer> q = new ArrayDeque<>(); // 始终维护 m 个数
int[] cnt; // 每个元素出现的次数
long[] sum; // 累积和
int m, k, n = 100000, N = n * 4 + 1; // 线段树所占用的空间为数组的四倍大小
public MKAverage(int m, int k) {
    cnt = new int[N];
    sum = new long[N];
    this.m = m;
    this.k = k;
}
public void addElement(int num) {
    if (q.size() == m) {
        int v = q.pollFirst();
        insert(1, 0, n, v, -1); // 当删除掉一个元素的时候，需要更新线段树中的和
    }
    insert(1, 0, n, num, 1);
    q.offerLast(num);
}
public int calculateMKAverage() {
    if (q.size() < m) return -1;
    int bgn = k + 1, end = m - k; // idx: 1 - based
    return (int)(query(1, 0, n, bgn, end) / (m - 2 * k));
}
void insert(int idx, int l, int r, int v, long d) { // d:
    cnt[idx] += d;
    sum[idx] += d * v;
    if (l == r) return;
    int m = l + (r - l) / 2;
    if (v <= m)
        insert(idx << 1, l, m, v, d); // 向左子树查询
    else insert(idx << 1 | 1, m+1, r, v, d); // 向右子树查询
}
long query(int idx, int l, int r, int bgn, int end) { // 线段中第 bgn 个到第 end 个
    if (l == r) { // 起始和结束最多出现 2 次此情况 ?
        int c = end - bgn + 1;
        return (long)c * l; //
    } else if (cnt[idx] == end - bgn + 1)
        return sum[idx];
    else {
        int m = l + (r - l) / 2;
        int cl = cnt[idx << 1]; // left child cnt
        // int cr = cnt[idx << 1 | 1]; // right child cnt
        if (cl >= end) // 搜索左子树
            return query(idx << 1, l, m, bgn, end);
        else if (cl >= bgn) // 搜索左右子树
            return query(idx << 1, l, m, bgn, cl) + query(idx << 1 | 1, m+1, r, 1, end - cl);
        else // cl < bgn, 搜索右子树
            return query(idx << 1 | 1, m+1, r, bgn - cl, end - cl);
    }
}
```

#### 1. 解题思路与分析: 三个 TreeMap, 自定义 TreeMap

```
CusTreeMap[] ms;
Deque<Integer> q;
int m, k, n;
public MKAverage(int m, int k) {
    this.m = m;
    this.k = k;
    q = new ArrayDeque<>();
    if (m - 2 * k > 0) {
        n = 3;
        ms = new CusTreeMap[n];
        ms[1] = new CusTreeMap(m - 2 * k);
    } else {
        n = 2;
        ms = new CusTreeMap[n];
    }
}
```

```

        ms[0] = new CusTreeMap(k);
        ms[n-1] = new CusTreeMap(k);
    }
    // 删除 num, 结果总是使 mapList 的小、中、大三个 treemap 依次填充。(先保证最小的 TreeMap 填充、再保证中间的 TreeMap 填充、最后是最大的填充)
    private void removeElement(int num) {
        boolean removed = false;
        for (int i = 0; i < n; i++) {
            if (!removed)
                removed = ms[i].remove(num);
            else { // 将后现一两个图中的最小元素向后一个图中挪动一个数值
                Integer minK = ms[i].pollFirst();
                if (minK == null) break;
                ms[i-1].add(minK);
            }
        }
    }
    public void addElement(int num) {
        if (q.size() == m) {
            int v = q.pollFirst();
            removeElement(v);
        }
        q.offerLast(num);
        Integer vToAdd = num;
        for (int i = 0; i < n && vToAdd != null; i++)
            vToAdd = ms[i].add(vToAdd); // 记得这里返回的是: 如果图中已有 k 个元素, 扔出来的最大键
    }
    public int calculateMKAverage() {
        if (q.size() < m || n < 3) return -1;
        return ms[1].avg();
    }
    class CusTreeMap {
        TreeMap<Integer, Integer> m;
        final int capacity;
        int size, sum;
        public CusTreeMap(int capacity) {
            m = new TreeMap<>();
            this.capacity = capacity;
        }
        public boolean remove(int key) {
            if (m.containsKey(key)) {
                m.put(key, m.get(key)-1);
                if (m.get(key) == 0) m.remove(key);
                sum -= key;
                size--;
                return true;
            }
            return false;
        }
        public Integer pollFirst() { // return key
            if (m.size() > 0) {
                int k = m.firstKey();
                // m.remove(k); // BUG: 你也不能用原始的 TreeMap.remove(), 因为它会移走所有的重复 (如果这个元素存在重复的话)
                remove(k); // !!!
                return k; // 这里没有自动更新 和
                // return m.firstKey(); // BUG: 这里并没有真正移走这个元素, 只是返回了第个元素的键
            }
            return null;
        }
        public Integer add(int key) { // 返回的是删除掉元素的键
            m.put(key, m.getOrDefault(key, 0) + 1); // 这里新填入的元素是否是最后一个元素, 关系不大
            size++;
            sum += key;
            if (size > capacity) {
                int last = m.lastKey();
                m.put(last, m.get(last)-1);
                if (m.get(last) == 0) m.remove(last);
                sum -= last;
                size--;
                return last;
            }
            return null;
        }
        public int avg() {
            return sum / size;
        }
    }
}

```

## 2. 解题思路与分析: 树状数组

- 数状数组的解法: 另外第一次看到别人二分 + 树状数组也能求前 k 大的值。

```

// We can have a queue to maintain m elements
// Use two Fenwick tree, 1 for count and 1 for prefix sum
// Do 2 times binary search for the first k elements and the last k elements by using the count from our first fenwick tree
// We can get the sum by subtracting the sum of first k elements and sum of last k element by using our second fenwick tree
Queue<Integer> q = new LinkedList<>();
Fenwick fone, ftwo;
int [] cnt = new int [100010];

```

```

long sum = 0;
int m,k;
public MKAverage(int m, int k) {
    this.m = m;
    this.k = k;
    long A [] = new long [100010];
    long B [] = new long [100010];
    fone = new FenWick(A);
    ftwo = new FenWick(B);
}
public void addElement(int num) {
    q.add(num);
    sum += num;
    fone.update(num, 1);
    ftwo.update(num, num);
    cnt[num]++;
}
public int calculateMKAverage() {
    if (q.size() < m) return -1;
    while (q.size() > m) {
        int cur = q.poll();
        cnt[cur]--;
        sum -= cur;
        fone.update(cur, -1);
        ftwo.update(cur, -cur);
    }
    // binary search for the first k (there may be duplicated)
    int l = 0, r = cnt.length-1;
    int i = -1, j = -1; // pos1, pos2
    while (l <= r) { // 二分查找总计数
        int m = (r + l) / 2;
        long count = fone.sumRange(0, m);
        if (count >= k) {
            i = m;
            r = m - 1;
        } else l = m+1;
    }
    // binary search for the last k (there may be duplicated)
    l = 0;
    r = cnt.length-1;
    while (l <= r) {
        int m = l + (r-l)/2;
        long count = fone.sumRange(m, cnt.length-1);
        if (count >= k) {
            j = m;
            l = m + 1;
        } else r = m-1;
    }
    long sum1 = ftwo.sumRange(0, i);
    long sum2 = ftwo.sumRange(j, cnt.length-1);
    long cnt1 = fone.sumRange(0, i);
    long cnt2 = fone.sumRange(j, cnt.length-1);
    if (cnt1 > k)
        sum1 -= i*(cnt1-k);
    if (cnt2 > k)
        sum2 -= j*(cnt2-k);
    long remain = sum - sum1 - sum2; // 总和，减去两边最小最大各 K 个数的和
    return (int)(remain / (m-2*k));
}
class FenWick {
    long tree []; //1-index based
    long A [];
    long arr[];
    public FenWick(long [] A) {
        this.A = A;
        arr = new long [A.length];
        tree = new long [A.length + 1];
    }
    public void update(int i, int v) {
        arr[i] += v;
        i++;
        while (i < tree.length) {
            tree[i] += v;
            i += (i & -i); // 这是的原理细节再回去复习一下
        }
    }
    public long sumRange(int i, int j) {
        return pre(j+1)-pre(i);
    }
    public long pre(int i) {
        long sum = 0;
        while (i > 0) {
            sum += tree[i];
            i -= (i & -i);
        }
        return sum;
    }
}

```

- 其它比较有兴趣以的 BST 二叉树的解法，改天补起来

### 18.1.5 315. Count of Smaller Numbers After Self - Hard

You are given an integer array `nums` and you have to return a new counts array. The counts array has the property where `counts[i]` is the number of smaller elements to the right of `nums[i]`.

- 解题思路与分析: 二分查找的插入排序

```
public List<Integer> countSmaller(int[] a) { // O(NlogN) 插入排序
    int n = a.length;
    List<Integer> ans = new ArrayList<>();
    List<Integer> list = new ArrayList<>(); // 新建一个 list，用于排序
    int [] tmp = new int [n]; // 为了提高效率，新建一个数组型的返回结果
    for (int i = n-1; i >= 0; i--) {
        int v = a[i]; // 将当前数字插入到新建 list 中，使用二分查找找到插入位置
        int l = 0, r = list.size()-1; // l: left; r: right 从排好序的 list 中二分查找正确的插入位置
        while (l <= r) {
            int m = l + (r - l) / 2;
            if (v <= list.get(m)) r = m-1;
            else l = m + 1;
        }
        list.add(l, v); // 将当前数字插入到相应位置，保证 list 升序排列
        tmp[i] = l; // 当前位置前所有数字均小于当前数字，将个数加入返回结果
    }
    for (Integer v : tmp) ans.add(v);
    return ans;
}
```

- 解题思路与分析: 数状数组

- 官方题解:<https://leetcode-cn.com/problems/count-of-smaller-numbers-after-self/solution/ji-suan-you->

```
private int[] c;
private int[] a; // 离散化、去重复 后的数组
public List<Integer> countSmaller(int[] nums) {
    List<Integer> ans = new ArrayList<Integer>();
    discretization(nums);
    init(nums.length + 5);
    for (int i = nums.length - 1; i >= 0; --i) {
        int id = getId(nums[i]);
        ans.add(query(id - 1));
        update(id);
    }
    Collections.reverse(ans);
    return ans;
}
private void init(int length) {
    c = new int[length];
    Arrays.fill(c, 0);
}
private int lowBit(int x) {
    return x & (-x);
}
private void update(int pos) {
    while (pos < c.length) {
        c[pos] += 1;
        pos += lowBit(pos);
    }
}
private int query(int pos) {
    int ret = 0;
    while (pos > 0) {
        ret += c[pos];
        pos -= lowBit(pos);
    }
    return ret;
}
private void discretization(int[] nums) { // 离散化、去重复 ?
    Set<Integer> set = new HashSet<Integer>(Arrays.stream(nums).boxed().collect(Collectors.toList()));
    int size = set.size();
    a = new int[size];
    int index = 0;
    for (int num : set) a[index++] = num;
    Arrays.sort(a);
}
private int getId(int x) {
    return Arrays.binarySearch(a, x) + 1; //
}
```

- 解题思路与分析: 归并排序 todo 补上

### 18.1.6 699. Falling Squares - Hard

There are several squares being dropped onto the X-axis of a 2D plane.

You are given a 2D integer array positions where  $\text{positions}[i] = [\text{left}_i, \text{sideLength}_i]$  represents the  $i$ th square with a side length of  $\text{sideLength}_i$  that is dropped with its left edge aligned with X-coordinate  $\text{left}_i$ .

Each square is dropped one at a time from a height above any landed squares. It then falls downward (negative Y direction) until it either lands on the top side of another square or on the X-axis. A square brushing the left/right side of another square does not count as landing on it. Once it lands, it freezes in place and cannot be moved.

After each square is dropped, you must record the height of the current tallest stack of squares.

Return an integer array ans where  $\text{ans}[i]$  represents the height described above after dropping the  $i$ th square.

1. 解题思路与分析:  $O(N^2)$  本能土办法 方块的大小不是固定的, 有可能很大, 但是不管方块再大, 只要有一点点部分搭在其他方块上面, 整个方块都会在上面, 并不会掉下来, 让我们求每落下一个方块后的最大高度。我们知道返回的是每落下一个方块后当前场景中的最大高度, 那么返回的数组的长度就应该和落下方块的个数相同。所以我们可以建立一个 heights 数组, 其中  $\text{heights}[i]$  表示第  $i$  块方块落下后所在的高度, 那么第  $i$  块方块落下后场景的最大高度就是  $[0, i]$  区间内的最大值。那么我们在求出 heights 数组后, 只要不停返回  $[0, i]$  区间内的最大值即可。继续来看, 这道题的难点就是方块重叠的情况, 我们先来想, 如果各个方块不重叠, 那么  $\text{heights}[i]$  的高度就是每个方块自身的高度。一旦重叠了, 就得在已有的基础上再加上自身的高度。那么我们可以采用 brute force 的思想, 对于每一个下落的方块, 我们都去看和后面将要落下的方块有没有重叠, 有的话, 和后面将要落下的方块的位置相比较, 取二者中较大值为后面要落下的方块位置高度  $\text{heights}[j]$ 。判读两个方块是否重叠的方法是如果方块 2 的左边界小于方块 1 的右边界, 并且方块 2 点右边界大于方块 1 点左边界。就拿题目中的例子 1 来举例吧, 第一个下落的方块的范围是  $[1, 3]$ , 长度为 2, 则  $\text{heights}^1=2$ , 然后我们看其和第二个方块  $[2, 5]$  是否重叠, 发现是重叠的, 则  $\text{heights}^2$  更新为 2, 再看第三个方块  $[6, 7]$ , 不重叠, 不更新。然后第二个方块落下, 此时累加高度, 则  $\text{heights}^2=5$ , 再看第三个方块, 不重叠, 不更新。然后第三个方块落下,  $\text{heights}^3=1$ 。此时我们 heights 数组更新好了, 然后我们开始从头遍历, 维护一个当前最大值  $\text{curMax}$ , 每次将  $[0, i]$  中最大值加入结果 res 即可,

```
public List<Integer> fallingSquares(int[][] a) {
    List<Integer> ans = new ArrayList<>();
    int n = a.length, max = 0;
    int[] hi = new int[n]; // 表示第 i 块方块落下后所在的高度
    for (int i = 0; i < n; i++) {
        int h = a[i][1], l = a[i][0], r = a[i][0] + h;
        hi[i] += h;
        for (int j = i+1; j < n; j++) {
            int ll = a[j][0], rr = ll + a[j][1];
            // [[6,1],[9,2],[2,4]] 因为不能保证是从左往右延 x 轴顺序掉落, 所以加上 l < rr 也很重要 确保不管左右边有交叠
            if (ll < r && rr > l) // 保证 j 在 i 的右边, 并且有重叠区域
                hi[j] = Math.max(hi[j], hi[i]);
        }
        max = Math.max(max, hi[i]);
        ans.add(max);
    }
    return ans;
}
```

2. 解题思路与分析: 线段树 + 离散化

想象  $x$  轴是地面, 如果某个方块掉落的过程中遇到了之前的某个方块 (擦边而过不算), 则该方块会叠到上面。现在给定一个长  $n$  数组  $A$ ,  $A[i]$  存了第  $i$  个掉落的方块的信息, 其中  $A[i][0]$  表示它的左下角的  $x$  坐标,  $A[i][1]$  表示它的边长。要求返回一个长  $n$  数组  $B$ , 使得  $B[i]$  表示在  $A[i]$  掉落之后, 当前所有方块的最高点的  $y$  坐标。

思路是线段树 + 离散化。可以将  $x$  坐标离散化, 这样可以节省存储空间 (离散化的过程其实就是将一个数组  $d$  排序后去重, 然后将每个数映射到它的下标。这样在线段树建树的时候, 就只需维护  $[0, 1, d-1, [0, 1, d-1]]$  这个区间的信息就行了, 这会极大减少线段树的空间消耗, 也从而会减少要做的操作的时间消耗)。具体来说, 给定一个将要下落的方块, 比如该方块的左端点的  $x$  坐标和右端点的  $x$  坐标分别是  $a$  和  $b$ , 边长是  $c$ , 那么我们需要实现两个操作, 第一是查询  $(a, b)$  里的最大值  $M$  (注意这里查询的是开区间  $(a, b)$  的最大值, 因为下落的方块擦着另一个方块的边的话, 是不会叠上去的), 另一个是将  $[a, b]$  里所有值都变成  $M + c$ 。本质上是要求一个数据结构可以查询区间最大值, 以及将区间修改为某一值, 这可以用线段树 + 懒标记来做到。在离散化之后, 为了使得区间  $(a, b)$  非空 (注意这里  $a$  和  $b$  都是离散化之后的值, 此时  $(a, b) = [a+1, b-1]$ ), 我们可以在离散化的时候将方块的中点也加入一起做离散化, 但是这会导致中点变成非整数, 这里将原坐标乘以 222 就行了。

给定一个数学里的二维平面直角坐标系 $xOy$ , 想象有若干正方形方块掉落, 题目保证正方形方块的左下端点的坐标是正整数, 也保证方块边长是正整数。想象 $x$ 轴是地面, 如果某个方块掉落的过程中遇到了之前的某个方块(擦边而过不算), 则该方块会叠到上面。现在给定一个长 $n$ 数组 $A$ ,  $A[i]$ 存了第 $i$ 个掉落的方块的信息, 其中 $A[i][0]$ 表示它的左下角的 $x$ 坐标,  $A[i][1]$ 表示它的边长。要求返回一个长 $n$ 数组 $B$ , 使得 $B[i]$ 表示在 $A[i]$ 掉落之后, 当前所有方块的最高点的 $y$ 坐标。

思路是线段树 + 离散化。可以将 $x$ 坐标离散化, 这样可以节省存储空间(离散化的过程其实就是将一个数组 $d$ 排序后去重, 然后将每个数映射到它的下标)。这样在线段树建树的时候, 就只需维护 $[0, l_d - 1]$ 这个区间的信息就行了, 这会极大减少线段树的空间消耗, 也从而会减少要做的操作的时间消耗)。具体来说, 给定一个将要下落的方块, 比如该方块的左端点的 $x$ 坐标和右端点的 $x$ 坐标分别是 $a$ 和 $b$ , 边长是 $c$ , 那么我们需要实现两个操作, 第一是查询 $(a, b)$ 里的最大值 $M$  (注意这里查询的是开区间 $(a, b)$ 的最大值, 因为下落的方块擦着另一个方块的边的话, 是不会叠上去的), 另一个是将 $[a, b]$ 里所有值都变成 $M + c$ 。本质上是要求一个数据结构可以查询区间最大值, 以及将区间修改为某一值, 这可以用线段树 + 懒标记来做到。在离散化之后, 为了使得区间 $(a, b)$ 非空 (注意这里 $a$ 和 $b$ 都是离散化之后的值, 此时 $(a, b) = [a + 1, b - 1]$ ), 我们可以在离散化的时候将方块的中点也加入一起做离散化, 但是这会导致中点变成非整数, 这里将原坐标乘以2就行了。代码如

```

public List<Integer> fallingSquares(int[][] a) { // 需要对数据进行离散化处理, 离散化的目的是为了线段树处理起来方便; 离散的是 x 轴的横坐标
    List<Integer> x = new ArrayList<>();
    for (int [] v : a) {
        int i = v[0], j = i + v[1];
        x.add(i * 2);
        x.add(j * 2);
        x.add(i + j);
    }
    x = getUniques(x);
    MaxSeg maxSeg = new MaxSeg(x.size());
    List<Integer> ans = new ArrayList<>();
    for (int [] v : a) {
        int i = v[0], j = i + v[1];
        i = getIdxInList(i * 2, x);
        j = getIdxInList(j * 2, x);
        int h = maxSeg.query(1, i+1, j-1);
        maxSeg.update(1, i, j, h + v[1]);
        ans.add(maxSeg.query());
    }
    return ans;
}
int getIdxInList(int v, List<Integer> list) { // 找到 x 在离散化之后的值是多少, 其实就是求 xs 里 x 的下标, 可以二分来找到
    int l = 0, r = list.size() - 1;
    while (l < r) {
        int m = l + (r - l) / 2;
        if (list.get(m) >= v) r = m;
        else l = m + 1;
    }
    return l;
}
List<Integer> getUniques(List<Integer> l) {
    l.sort(Integer::compareTo);
    int j = 0; // 返回结果链表的下标 idx
    for (int i = 0; i < l.size(); i++) {
        if (i == 0 || l.get(j-1) != l.get(i))
            l.set(j++, l.get(i));
    }
    return l.subList(0, j);
}
class MaxSeg { // 实现一下带懒标记的线段树: 这棵树好强大
    class Node { // v 是 [l, r] 区间的最大值, lazy 是懒标记
        int l, r, v, lazy;
        public Node(int l, int r) {
            this.l = l;
            this.r = r;
        }
    }
    Node [] tree;
    public MaxSeg(int n) {
        tree = new Node[n << 2]; // n * 2 * 2
        buildTree(1, 0, n-1); // 下标从 1 开始 自顶向下
    }
    void buildTree(int i, int l, int r) {
        tree[i] = new Node(l, r);
        if (l == r) return;
        int m = l + r >> 1; // (l + r) / 2
        buildTree(i << 1, l, m);
        buildTree(i << 1 | 1, m+1, r);
    }
    void pushUp(int i) { // 自底向上: 左自、右叶子节点向顶更新最大值, 取左右节点的最大值
}

```

```

        tree[i].v = Math.max(tree[i << 1].v, tree[i << 1 | 1].v);
    }
    void pushDown(int i) { // 懒标记向底、叶子方向推进一层
        int c = tree[i].lazy;
        if (c != 0) { // 打有懒标记
            tree[i].lazy = 0;
            tree[i << 1].v = tree[i << 1 | 1].v = c;
            tree[i << 1].lazy = tree[i << 1 | 1].lazy = c;
        }
    }
    void update(int i, int l, int r, int c) { // 自顶向下传递懒标记，再自底向上更新父节点的值：取左右子节点的最大值
        if (l <= tree[i].l && tree[i].r <= r) { // 任务不需要下发，可以用懒标记懒住
            tree[i].v = tree[i].lazy = c; // 这里 tree[i].v = tree[i].lazy = c : c 是想要更新到的新值 v，用它来更新懒标记和 v 值
            return ;
        }
        pushDown(i); // 任务不得不下发，则先下发给两个孩子
        int m = tree[i].l + tree[i].r >> 1;
        if (l <= m) update(i << 1, l, r, c); // 回归调用，下传更新至左右子节点
        if (m + 1 <= r) update(i << 1 | 1, l, r, c);
        pushUp(i); // 孩子完成了任务，再修改自己的值
    }
    int query(int i, int l, int r) {
        if (l <= tree[i].l && r >= tree[i].r) return tree[i].v;
        pushDown(i);
        int ans = 0, m = tree[i].l + tree[i].r >> 1;
        if (l <= m) ans = Math.max(ans, query(i << 1, l, r));
        if (m + 1 <= r) ans = Math.max(ans, query(i << 1 | 1, l, r));
        return ans;
    }
    int query() {
        return tree[1].v;
    }
}

```

### 3. 解题思路与分析: 超简洁版的线段树, 效率奇高

- <http://www.noobyard.com/article/p-sxwzvpgp-nz.html>

- 去找一下原文件中的优化步骤

```

private class Node { // 描述方块以及高度
    int l, r, h, maxR;
    Node left, right;
    public Node(int l, int r, int h, int maxR) {
        this.l = l;
        this.r = r;
        this.h = h;
        this.maxR = maxR;
        this.left = null;
        this.right = null;
    }
}
public List<Integer> fallingSquares(int[][] positions) {
    List<Integer> res = new ArrayList<>(); // 建立返回值
    Node root = null; // 根节点， 默认为零
    int maxH = 0; // 目前最高的高度
    for (int[] position : positions) {
        int l = position[0]; // 左横坐标
        int r = position[0] + position[1]; // 右横坐标
        int e = position[1]; // 边长
        int curH = query(root, l, r); // 目前区间的最高的高度
        root = insert(root, l, r, curH + e);
        maxH = Math.max(maxH, curH + e);
        res.add(maxH);
    }
    return res;
}
private Node insert(Node root, int l, int r, int h) {
    if (root == null) return new Node(l, r, h, r);
    if (l <= root.l)
        root.left = insert(root.left, l, r, h);
    else
        root.right = insert(root.right, l, r, h);
    root.maxR = Math.max(r, root.maxR); // 最终目标是仅仅须要根节点更新 maxR
    return root; // 返回根节点
}
private int query(Node root, int l, int r) {
    // 新节点的左边界大于等于目前的 maxR 的话，直接获得 0，不须要遍历了
    if (root == null || l >= root.maxR) return 0;
    int curH = 0; // 高度
    if (!(r <= root.l || root.r <= l)) // 是否跟这个节点相交
        curH = root.h;
    // 剪枝
    curH = Math.max(curH, query(root.left, l, r));
    if (r > root.l)
        curH = Math.max(curH, query(root.right, l, r));
    return curH;
}

```

### 18.1.7 1483. Kth Ancestor of a Tree Node - Hard 倍增法 binary lifting

You are given a tree with  $n$  nodes numbered from 0 to  $n - 1$  in the form of a parent array  $\text{parent}$  where  $\text{parent}[i]$  is the parent of  $i$ th node. The root of the tree is node 0. Find the  $k$ th ancestor of a given node.

The  $k$ th ancestor of a tree node is the  $k$ th node in the path from that node to the root node.

Implement the `TreeAncestor` class:

`TreeAncestor(int n, int[] parent)` Initializes the object with the number of nodes in the tree and the parent array. `int getKthAncestor(int node, int k)` return the  $k$ th ancestor of the given node  $node$ . If there is no such ancestor, return -1.

#### 1. 解题思路与分析: 倍增 binary lifting

给定一个 $n$ 个节点的树，每个节点编号 $0 \sim n - 1$ ，另外给出每个节点的父亲节点是谁，以数组 $p$ 给出。要求设计一个数据结构可以应答这样的询问，每次询问给出一个节点编号 $u$ 和一个正整数 $k$ ，问 $u$ 的第 $k$ 个祖先是谁（即 $u$ 向上走 $k$ 步是谁）。如果该祖先不存在则返回-1。

可以用倍增法。设 $f[u][k]$ 是节点 $u$ 向上走 $2^k$ 步能走到的节点，那么有

$$f[u][k] = f[f[u][k-1]][k-1]$$

初始条件 $f[u][0] = p[u]$ ，从而可以递推出 $f$ 数组。因为一共有 $n$ 个节点，树最高就是 $n$ ，也就是说向上最多能走 $n - 1$ 步，我们找到最小的 $2^k \geq n - 1$ ，把 $f$ 的第二维开 $k = \lfloor \log_2 n - 1 \rfloor + 1$ 这么长，以保证询问都能被正确应答。接下来考虑如何应答询问，可以用类似快速幂的思想，比方说 $k$ 的二进制表示是 $1101_2$ ，那么 $1101_2 = 2^0 + 2^2 + 2^3$ ，那么可以先求 $u_1 = f[u][0]$ ，这样就跳了 $2^0$ 步，再接着求 $u_2 = f[u_1][2]$ ，这样又跳了 $2^2$ 步，再接着求 $f[u_2][3]$ ，这样就将 $k$ 步全跳完了。中途如果发现要跳的幂次大于了 $f$ 的第二维能取的最大值，或者跳到了-1，则直接返回-1。代码如下：

- 预处理时间复杂度  $O(n \log n)$ ，每次询问时间  $O(\log n)$ ，空间  $O(n \log n)$ 。

```
private int [][] p;
private int log;
public TreeAncestor(int n, int[] parent) {
    log = (int) (Math.log(n - 1) / Math.log(2)) + 1;
    p = new int[n][log];
    for (int i = 0; i < parent.length; i++) // 初始化 p 数组
        p[i][0] = parent[i];
    for (int i = 1; i < log; i++) // 按公式递推 p 数组
        for (int j = 0; j < n; j++)
            if (p[j][i-1] != -1)
                p[j][i] = p[p[j][i-1]][i-1];
            else p[j][i] = -1;
}
public int getKthAncestor(int node, int k) {
    int pow = 0;
    while (k > 0) {
        if (pow >= log || node == -1) return -1;
        if ((k & 1) == 1)
            node = p[node][pow];
        k >= 1;
        pow++;
    }
    return node;
}
```

#### 2. 解题思路与分析

```
Map<Integer, List<Integer>> adj;
int [][] par;
public TreeAncestor(int n, int[] parent) {
    par = new int [n][30]; // 30 , 16: 不能证它是一棵很平衡的二叉树
    adj = new HashMap<>();
    for (int i = 0; i < n; i++) {
        Arrays.fill(par[i], -1);
        adj.put(i, new ArrayList<>());
    }
    for (int i = 0; i < parent.length; i++)
        if (parent[i] != -1) {
            adj.get(parent[i]).add(i); // 自顶向下: 父 --> 子节点
            par[i][0] = parent[i]; // 每个子节点的第一个父节点 ( $2^0 = 1$ ), 即为父节点 // 自底向上: 子节点:  $2^0$  父节点、 $2^1$  节点、 $2^2$  节点
        }
    dfs(0);
}
public int getKthAncestor(int node, int k) {
    for (int i = 0; k > 0; i++, k >= 1) // k /= 2
```

```

    if ((k & 1) == 1) {
        node = par[node][i];
        if (node < 0) return -1;
    }
    return node;
}
private void dfs(int idx) { // 自顶向下：从父节点遍历子节点
    for (int i = 1; par[idx][i-1] >= 0; i++) // 穷追溯源：一直找到整棵树的根节点： 0
        par[idx][i] = par[par[idx][i-1]][i-1]; // 这里多想想
    for (int next : adj.get(idx))
        dfs(next);
}

```

### 18.1.8 236 二叉树的最近公共祖先

### 18.1.9 1505. Minimum Possible Integer After at Most K Adjacent Swaps On Digits - Hard BIT 树状数组

You are given a string num representing the digits of a very large integer and an integer k. You are allowed to swap any two adjacent digits of the integer at most k times.

Return the minimum integer you can obtain also as a string.

#### 1. 解题思路与分析

```

public String minInteger(String t, int k) {
    int n = t.length();
    t = " " + t;
    char [] s = t.toCharArray();
    ArrayDeque<Integer> [] q = new ArrayDeque [10];
    for (int i = 1; i <= n; i++) {
        int j = s[i] - '0';
        if (q[j] == null) q[j] = new ArrayDeque<>();
        q[j].offerLast(i);
    }
    BIT bit = new BIT(n);
    StringBuilder sb = new StringBuilder();
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j < 10; j++) { // 从小数值往大数值遍历
            if (q[j] == null || q[j].isEmpty()) continue;
            int top = q[j].peekFirst(), pos = top + bit.sum(top); // pos 是最优解的位置，最优解的位置是原来的位置加上偏移量
            if (pos - i <= k) {
                k -= pos - i;
                sb.append(j);
                q[j].pollFirst();
                bit.add(1, 1); // 更新 [1, t] 这段的值每个加 1，即向右偏移 1 位。为什么要从 1 开始更新：假装每次都移动到最前端，方便计算 ?
                bit.add(top, -1);
                break;
            }
        }
    }
    return sb.toString();
}
class BIT { // 开一个树状数组类，维护每个位置的字符的向右的偏移量 ? 向左偏移量
    private int n;
    private int [] a;
    public BIT(int n) {
        this.n = n;
        this.a = new int [n+1];
    }
    public void add(int idx, int v) { // 只有发生偏移，才移动某段区间的值
        while (idx <= n) {
            a[idx] += v;
            idx += lowbit(idx);
        }
    }
    public int sum(int idx) { // 得到以 i 为下标 1-based 的所有子、叶子节点的和，也就是 [1, idx] 的和，1-based
        int ans = 0;
        while (idx > 0) {
            ans += a[idx];
            idx -= lowbit(idx);
        }
        return ans;
    }
    int lowbit(int x) {
        return x & -x;
    }
}

```

### 18.2 求最大最小值、位操作值的线段树