

Algorithm Puzzles 解题笔记

目录

Algorithm Puzzles 解题笔记	1
0. 概要	8
1. Q01: 回文十进制数	9
2. Q02: 四则运算组合游戏	9
2.1 问题描述	9
2.2 解题分析	10
2.3 算式值的计算	10
3. Q03: 翻牌	11
4. Q04: 切分木棒	11
4.1 问题描述	11
4.2 数学解法	11
5. Q05: 硬币兑换	12
5.1 问题描述	12
5.2 递推表达式	12
6. Q06: 改版考拉兹猜想	13
7. Q07: 日期的二进制转换	13
7.1 解题分析	13
7.2 优化	13
8. Q08: 优秀的扫地机器人	14
8.1 问题描述	14
8.2 解题分析	14
9. Q09: 落单的男女	15
9.1 问题描述	15
9.2 解题分析	16
10. Q10: 轮盘的最大值	16
10.1 问题描述	16
10.2 解题分析	17
11. Q11: 斐波那契数列	18
11.1 问题描述	18
11.2 解题分析	18
12. Q12: 平方根数字	19
12.1 问题描述	19
12.2 解题分析	19
13. Q13: 满足字母算式的解法	20
13.1 问题描述	20
13.2 解题分析	20
14. Q14: 国名接龙	21

14.1	问题描述	21
14.2	解题分析 1—深度优先搜索	22
14.3	解题分析 2—广度优先搜索	22
14.3.1	BFS 基本要素	22
14.3.2	最长路径问题 vs 最短路径问题	22
14.3.3	算法流程 1	23
15.	Q15: 走楼梯	23
15.1	问题描述	23
15.2	递推表达式	24
16.	Q16: 3 根绳子折成四边形	24
16.1	问题描述	24
16.2	初始解法—解法 1	25
16.2.1	三者互素的判断	25
16.3	改进解法—解法 2	25
16.4	改进解法—解法 3	25
16.5	改进解法—解法 4	26
17.	Q17: 30 人 31 足游戏	26
17.1	问题描述	26
17.2	解法 1—作为计数问题的闭式解	26
17.3	解法 2—作为计数问题解的递推关系式	27
18.	Q18: 水果酥饼日	28
18.1	问题描述	28
18.2	解题分析—深度优先搜索	28
18.3	代码实现有两个小细节	30
19.	Q19: 朋友的朋友也是朋友吗	30
19.1	问题描述	30
19.2	解题分析	31
20.	Q20: 受难立面魔方阵	32
20.1	问题描述	32
20.2	解法 1—笨办法	33
20.3	解法 2—逆向思维	33
20.4	解法 3—动态规划	34
21.	Q21: 异或运算杨辉三角形	35
21.1	问题描述	35
21.2	解法 1—直观方法	35
21.3	解法 2—利用比特运算	35
21.4	实现代码	36
22.	Q22: 不缠绕的纸杯电话	36
22.1	问题描述	36
22.2	解题分析	37
23.	Q23: 二十一点通吃	38
23.1	问题描述	38
23.2	解法 1—暴力搜索	39
23.3	解法 2—深度优先路径搜索	39
23.4	解法 3—动态规划	40
23.5	后记	40
24.	Q24: 完美的三振出局	41

24.1	问题描述	41
24.2	解题分析	41
24.2.1	DFS 算法流程	41
24.2.2	遍历下一个状态	42
24.3	后记	42
25.	Q25: 鞋带的时髦系法	43
25.1	问题描述	43
25.2	解题分析	43
25.2.1	状态表示方法	43
25.2.2	DFS 算法流程	44
25.2.3	遍历下一个状态	44
25.2.4	交叉判断	44
25.2.5	运行结果	45
25.3	后记	45
26.	Q26: 高效的立体停车场	45
26.1	问题描述	45
26.2	解题分析	46
26.2.1	BFS 算法流程	46
26.2.2	状态表示方法	47
26.2.3	遍历下一个状态	47
27.	Q27: 禁止右转—深度优先搜索	47
27.1	问题描述	47
27.2	分析	48
27.3	Solution#1—递归式实现	49
27.4	Solution#2—非递归式实现	50
28.	Q28: 社团活动的最优分配方案	50
28.1	问题描述	50
28.2	解题分析	51
28.3	递归实现	52
29.	Q29: 合成电阻的黄金分割比	53
29.1	问题描述	53
29.2	解题分析	53
29.2.1	分割成两组	53
29.2.2	N=5 时的分割例	55
29.2.3	可能的阻值计算例	55
29.2.4	算法实现流程	56
29.3	代码及测试	56
29.4	后记	57
29.4.1	分割的洞见	57
29.4.2	另一种思路	57
30.	Q30: 插线板连接方式	57
30.1	问题描述	57
30.2	递归表达式	57
30.3	去重(repetition removal)	58
30.3.1	双口插线板的去重	59
30.3.2	三口插线板的去重	59
31.	Q31: 计算最短路径	59
31.1	问题描述	59

31.2	解题分析	59
31.2.1	BFS or DFS?	59
31.2.2	节点的表示	59
31.2.3	如何避免往返程通过同一条边	60
31.3	运行结果	60
32.	Q32: 榻榻米的铺法	60
32.1	问题描述	60
32.2	解题分析	61
32.2.1	如何判断能否铺	61
32.2.2	状态表示和遍历	62
32.2.3	围栏	63
33.	Q33: 飞车与角行的棋步	64
33.1	问题描述	64
33.2	解题分析	64
34.	Q34: 命中注定的相遇	65
34.1	问题描述	65
34.2	解题分析	65
34.3	后记	66
34.4	Bug-Fix	67
34.4.1	分析	67
34.4.2	后记 2	68
35.	Q35: 0 和 7 的回文数	68
35.1	问题描述	68
35.2	解法 1—暴力搜索	69
35.3	解法 2—逆向思考	69
36.	Q36: 翻转骰子	71
36.1	问题描述	71
36.2	解题分析	72
37.	Q37: 翻转 7 段码	73
37.1	问题描述	73
37.2	解题分析 1—暴力搜索	73
37.3	解题分析 2	74
37.4	后记	74
38.	Q38: 填充白色	75
38.1	问题描述	75
38.2	解题分析	76
38.2.1	Naïve Approach	76
38.2.2	最大路径问题—广度优先搜索	76
38.2.3	节点状态表示及翻转操作	76
38.3	后记	77
39.	Q39: 反复排序	78
39.1	问题描述	78
39.2	解题分析	79
39.2.1	Naïve Approach—正向全量搜索	79
39.2.2	缩小搜索范围	79
39.2.3	以递归的方式实现	80
39.2.4	反向搜索	80
40.	Q40: 优雅的 IP	80

40.1	问题描述	80
40.2	解题分析	81
40.2.1	笨办法	81
40.2.2	逆向思考	81
41.	Q41: 只用 1 个数字表示 1234	83
41.1	问题描述	83
41.2	解题分析	83
41.3	后记	84
42.	Q42: 30 人 31 足游戏	85
42.1	问题描述	85
42.2	分析	85
42.3	BFS 算法描述	86
42.4	初始实现	86
42.5	Bug-fix and Optimization	87
42.5.1	When to update visited?	87
42.5.2	What to implement visited?	87
42.5.3	Queue or deque?	87
42.6	优化实现：代码及测试	88
43.	Q43: 让玻璃杯水量减半	88
43.1	问题描述	88
43.2	解题分析	88
43.2.1	节点状态表示	88
43.2.2	邻节点搜索	89
43.2.3	路径历史记忆以及判断邻节点是否在路径历史中	89
44.	Q44: 质数矩阵	90
44.1	问题描述	90
44.2	解题分析	90
44.3	后记	91
45.	Q45: 排序交换次数的最少化	92
45.1	问题描述	92
45.2	解题分析	92
46.	Q46: 唯一的 Ox 序列	93
46.1	问题描述	93
46.2	解题分析	94
46.3	后记	94
47.	Q47: 格雷码循环	95
47.1	问题描述	95
47.2	解题分析	95
48.	Q48: 翻转得到交错排列	97
48.1	问题描述	97
48.2	解题分析	97
48.2.1	圆圈排列的表示	97
48.2.2	翻转运算	98
48.2.3	算法流程	98
49.	Q49: 欲速则不达	100
49.1	问题描述	100
49.2	解题分析	100

50. Q50: 完美洗牌	102
50.1 问题描述	102
50.2 解题分析	103
50.2.1 思路 1	103
50.2.2 思路 2	103
51. Q51: 同时结束的沙漏	105
51.1 问题描述	105
51.2 原题的表述	106
51.3 解题分析	106
51.3.1 转换为线性排列	106
52. Q52: 糖果恶作剧	108
52.1 问题描述	108
52.2 解题分析	108
53. Q53: 同数包夹	110
53.1 问题描述	110
53.2 分析	110
53.2.1 节点（状态）表示	110
53.2.2 邻节点搜索	110
53.3 递归实现：从小到大插入	110
53.4 递归实现：从大到小插入	111
53.5 递归实现：二进制存储	111
53.6 非递归实现	111
54. Q54: 偷懒的算盘	112
54.1 问题描述	112
54.2 解题分析	112
54.3 代码及测试	113
54.4 递推关系	113
54.5 代码及测试 2	114
55. Q55: 平分蛋糕	115
55.1 问题描述	115
55.2 解题分析	115
55.2.1 初始算法流程	116
55.2.2 优化	117
55.3 代码及测试	118
56. Q56: 鬼脚图中的横线	119
56.1 问题描述	119
56.2 思路 1—贪心策略	120
56.3 思路 2	122
56.3.1 从鬼脚图出发进行变形	122
56.3.2 反向变换	124
56.3.3 进一步探索	125
56.3.4 小结	126
56.3.5 实现	126
56.4 代码实现	127
56.5 后记	127
57. Q57: 最快的联络网	130
57.1 问题描述	130
57.2 解题分析	130

57.2.1	学生的状态	130
57.2.2	状态转移	131
57.2.3	广度优先搜索	133
58.	Q58: 丢手绢游戏中的总移动距离	135
58.1	问题描述	135
58.2	解题分析	136
58.2.1	搜索树示意图	136
58.3	后记	137
59.	Q59: 合并单元格的方式	138
59.1	问题描述	138
59.2	解题分析	138
60.	Q60: 分割为同样大小	139
60.1	问题描述	139
60.2	解题分析	140
60.2.1	思路 1	140
60.2.2	思路 2	140
60.2.3	思路 3	140
60.2.4	连通性检测	142
60.3	代码及测试	142
61.	Q61: 不交叉一笔画	143
61.1	问题描述	143
61.2	解题分析	144
62.	Q62: 日历中的最大矩形	145
62.1	问题描述	145
62.2	解题分析	145
62.2.1	每个月的日历的矩阵表示	145
62.2.2	假日和调休公休日的处理	146
62.2.3	求最大矩形	147
62.2.4	暴力搜索	147
62.2.5	滑动窗搜索	147
62.3	代码及测试	148
63.	Q63: 迷宫会合	149
63.1	问题描述	149
63.2	解题分析	149
63.2.1	有效的迷宫	150
63.2.2	移动	150
63.2.3	能否碰上	150
63.2.4	搜索	151
63.3	代码及测试	151
64.	Q64: 麻烦的投接球	152
64.1	问题描述	152
64.2	解题分析	153
64.2.1	状态表示	153
64.2.2	算法流程	153
65.	Q65: 图形的一笔画	154
65.1	问题描述	154
65.2	解题分析	155
65.2.1	一笔画的条件	155

65.2.2	各模块的定点度数	155
65.2.3	如何计算整个拼图的度数	155
65.2.4	算法流程	156
66.	Q66: 设计填字游戏	157
66.1	问题描述	157
66.2	解题分析	157
66.2.1	基本算法流程	157
66.2.2	连通性检查	159
67.	Q67: 不挨着坐是一种礼节吗	159
67.1	问题描述	159
67.2	解题分析	160
67.2.1	基本思路	160
67.2.2	动态规划	160
68.	Q68: 异性相邻的座次安排	162
68.1	异性相邻的座次安排	162
68.2	解题分析	163
68.3	代码及测试	164
69.	Q69: 蓝白歌会	165
69.1	问题描述	165
69.2	基本思路	165
69.3	算法流程	166
69.4	实现要点	167

[Notations and Abbreviations]

0. 概要



本文是关于该书的完整的私房 python 解题笔记。

所有解题笔记最先发表在“[程序员的算法趣题：详细分析和 Python 全解](#)”，有一些在本文档中缺失的信息在以上博客中可以找到。

1. Q01: 回文十进制数

略。参见以上博客。

2. Q02: 四则运算组合游戏

2.1 问题描述

在任意 4 个数字 (0,1,2,...,9) 构成的数字序列每两个数字之间插入四则运算符 {+, -, *, /} 或者不插入，计算由此得到该算式的结果。

例 1: '1+2-3*4' = -9

例 2: '1+23*4' = 93, 此例中，2 与 3 之间没有插入运算符，因此构成一个两位数

如果要求计算结果等于原数字序列逆序排列构成的四位数，求满足以上条件的数。

但是要求至少要插入一个运算符。因为一个都不插入的话，其实得到的就是原 4 个数字表示的四位数，那任何一个四位回文数都满足这个条件，这个结果没有什么意义。

为了避免一些 trivial solution，把 10 的倍数都排除在外，这样确保算式评估结果以及逆序数仍然为有效的四位数。

2.2 解题分析

本题涉及两个要点：

[算式的构成]

算式的构成就是构造不同的将运算符插入数字之间的组合，可以把空字符（注意，不是说 white-space，而是说长度为 0 的字符串“”）也看成是一个运算符，插入空字符也就相当于是不插入。

这样总共就有 5 个运算符：{‘+’，‘-’，‘*’，‘/’，‘’}，而四个数字构成的序列中共有 3 个位置可以插入，再扣除不能插入三个空字符‘’，因此总共有 $5^3 - 1 = 124$ 种组合。

在本体中考虑用字符串的方式表示所得到的算式，比如说：‘1+2-3*4’。

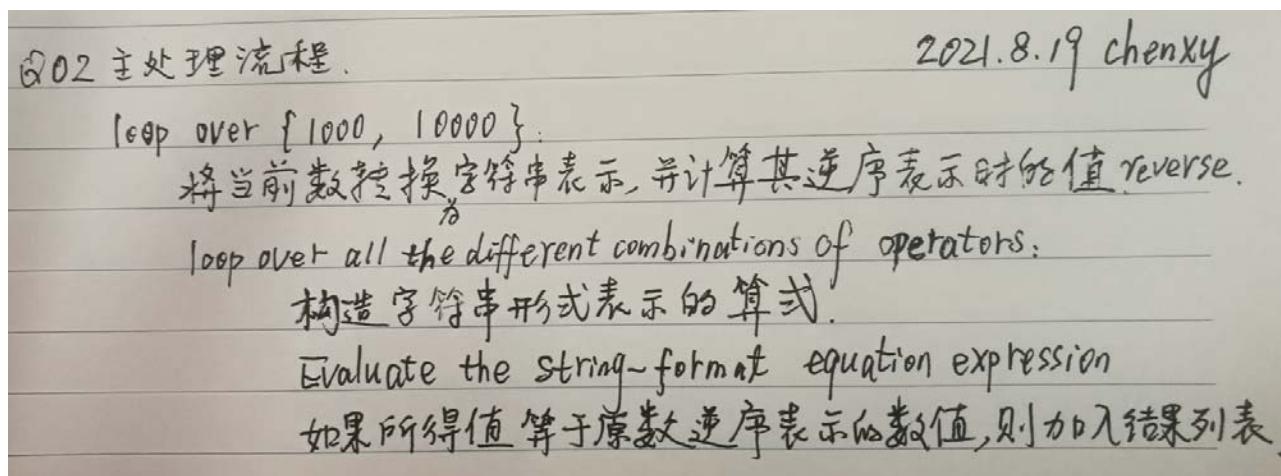
构造好算式后，接下来就是计算该算式的值。

[算式值的计算]

以上第一步中所构造的算式是所谓的中序表达式，而且没有括号，属于相对简单的情况。计算机中对算式的估计通常是将算式先由中序表达式转换为逆波兰表达式，然后再对逆波兰表达式进行评估处理。

感觉这还是一个比较有意思的东西，具体算法思路将另文单独解说。

基于以上讨论，本题的处理流程如下所示：



2.3 算式值的计算

To be added.

3. Q03: 翻牌

略。参见以上博客。

4. Q04: 切分木棒

4.1 问题描述

假设要把长度为 n 厘米的木棒切分为 1 厘米长的小段，但是每段木棒只能由 1 人切分。比如说，当木棒切分为 3 段，可以由 3 人同时分别切分各段。求最多 m 个人时，最少要经过几轮（原文是几次，容易误解）才能完成切割。

比如说 $n=8, m=3$ 时，分 4 轮切分即可。第 1 轮，由 1 个人将原始木棒切分为两段；第 2 轮（此时有 2 段木棒），由 2 个人同时它们切分为共 4 段；第 3 轮（此时有 4 段木棒），由 3 人分别对其中 3 段进行切分操作；第 4 轮（此时还剩 1 段木棒需要切分），由 2 人将最后剩下的 1 段切分为 2 段。

4.2 数学解法

这道题目其实数学的方式来解决可能更为简单。“数学解法”是相对于“编程解法”的说法，即仅通过数学或者逻辑的推到的方式求解的方法。以下我们通过数学推导的方式得到本题的解析解或者说闭式解(closed-form solution)

首先，将 n 厘米的木棒切分为 1 厘米长的小段总共需要 $(n-1)$ 次切分

其次，在有 m 个人时，每一轮最多可以（由 m 个人同时工作）进行 m 次切分操作。当木棒段数还小于 m 时，当前轮次所能进行的切分操作数受限于木棒段数；当木棒段数大于等于 m 时，当前轮次所能进行的切分操作数为 m

第三，在第 1 轮过后，木棒数变为 2 段；在第 2 轮过后，木棒数变为 4 段；在第 k 轮过后，木棒数变为 2^k (2 的 k 次方)段。因此到第 $\lceil \log_2 m \rceil$ 轮时，因为 $2^{\lceil \log_2 m \rceil - 1} \leq m$ 所能执行的切分操作数仍然是受限于木棒段数的。从第 $\lceil \log_2 m \rceil + 1$ 轮开始，则可以每轮最多执行 m 次切分操作

基于以上观察，可以知道：

- (1) 在前 $\lceil \log_2 m \rceil$ 轮中总共会将木棒切分为 $2^{\lceil \log_2 m \rceil}$ 段，总共进行了 $2^{\lceil \log_2 m \rceil} - 1$ 次切分操作
- (2) 接下来还需要的切分操作数为 $(n - 1) - (2^{\lceil \log_2 m \rceil} - 1) = n - 2^{\lceil \log_2 m \rceil}$ ，由于每轮可以执行 m 次切分操作，因此还需要 $\lceil (n - 2^{\lceil \log_2 m \rceil}) / m \rceil$ 轮

因此，总共需要 $\lceil \log_2 m \rceil + \lceil (n - 2^{\lceil \log_2 m \rceil}) / m \rceil$ 轮。

当 $n = 8, m = 3$ 时，代入上式可得答案为 4

当 $n = 20, m = 3$ 时，代入上式可得答案为 8

当 $n=100, m=5$ 时，代入上式可得答案为 22

5. Q05: 硬币兑换

5.1 问题描述

公交车上的零钱兑换机可以将纸币兑换成 10 日元、50 日元、100 日元和 500 日元的硬币，且每种硬币的数量都足够多。因为兑换出太多的硬币不太方便，只允许机器兑换成最多 15 枚硬币。比如说 1000 日元的纸币就不能对换成 100 枚 10 日元的硬币。

求兑换 1000 日元纸币会出现多少种不同组合？注意不计硬币兑出的先后顺序（即可以认为硬币是一起出来的）。

5.2 递推表达式

这道题目可以表达为如下数学方程的形式：

$$\begin{aligned} 500 * k_1 + 500 * k_2 + 500 * k_3 + 500 * k_4 &= 1000 \\ k_1 + k_2 + k_3 + k_4 &\leq 15 \end{aligned}$$

这是一道线性规划（Linear Programming）问题。

More generally, 令所需要兑换的纸币记为 $money$, 可用的硬币以数组形式（降序排列）表示为 $coins$, 最多允许的硬币数为 $maxNum$ 。记 $f(money, coins, maxNum)$ 表示符合题设要求的组合数。

首先考虑最大面值的硬币 $coins[0]$ （假定 $coins$ 中按降序排列）的使用。很显然， $coins[0]$ 最少可以用 0 枚（即不同），最多可以用 $\lfloor money/coins[0] \rfloor$ 枚，据此可以把问题分解为若干个更“小”的子问题来求解。由此可以得到以下递推关系式：

$$\begin{aligned} f(money, coins, maxNum) \\ = \sum_{k=0}^{\lfloor money/coins[0] \rfloor} f(money - k * coins[0], coins[1:], maxNum - k) \end{aligned}$$

Baseline cases 或者说边界条件如下：

$$\begin{aligned} f(0, *, *) &= 1 \\ f(money, coins, maxNum) &= 0, \text{if } money < \text{the smallest coin} \\ f(money, coins, maxNum) &= 0, \text{if } \lceil money/\text{the largest coin} \rceil > maxNum \end{aligned}$$

6. Q06: 改版考拉兹猜想

略。参见以上博客。

7. Q07: 日期的二进制转换

不同国家的日期的表示方式（习惯）不相同。

把年月日表示为 YYYYMMDD 这样的 8 位整数，然后把这个整数转换成二进制数并逆序排列，再把得到的二进制数转换成十进制数，求与原日期一致的日期。求得的日期要在上一次东京奥运会（1964-10-10）到下一次东京奥运会（2020-07-24）之间。

7.1 解题分析

其实，就是找转换为二进制数后构成了回文数（关于回文数参见 Q01）的日期。

本题的焦点之一是关于日期的处理，纯粹人工处理的话需要考虑月份大小、闰年以及二月份这种特殊月份等情况，非常容易出错。好在各种编程语言都有内置的库来做这件事情。Python 中由 `datetime` 模块来处理，所以问题变成了 `datetime` 模块的使用的问题。特别是日期递增的处理方式。

焦点之二是字符串的处理。

在以下代码中，有以下几个细节值得注意（不一定是最优的，只是本渣采用了这样比较笨拙的办法而已^-^）：

- (1) 由 `date` 变成 `string` 后中间会有分隔符“-”（没有找到不包含“-”的字符串变换方式），需要去除。
这里用 `str.replace()` 处理
- (2) 十进制数变成二进制数时输出的字符串，头上以“0b”开始，这个在进行是否回文数的判断中需要先去掉
- (3) 回文数的判断，这个采用和 Q01 相同的处理方式，用 `str[::-1]` 的方式取字符串逆序再与原字符串比较

7.2 优化

原书中提示了可以利用构成回文数的日期的特性可以大幅度削减搜索范围。当然其代价就是解决方案的可读性以及可扩展性。

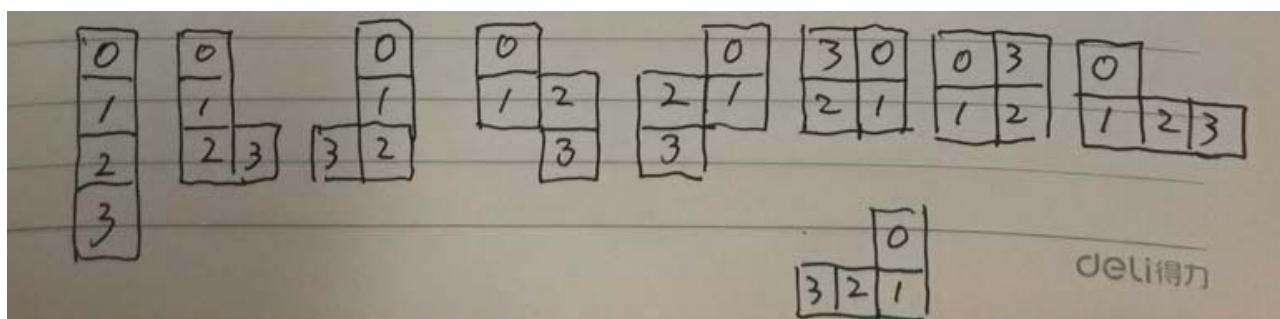
To be discussed.

8. Q08: 优秀的扫地机器人

8.1 问题描述

现在市面上有些扫地机器人有时候会反复地清扫某一个地方。

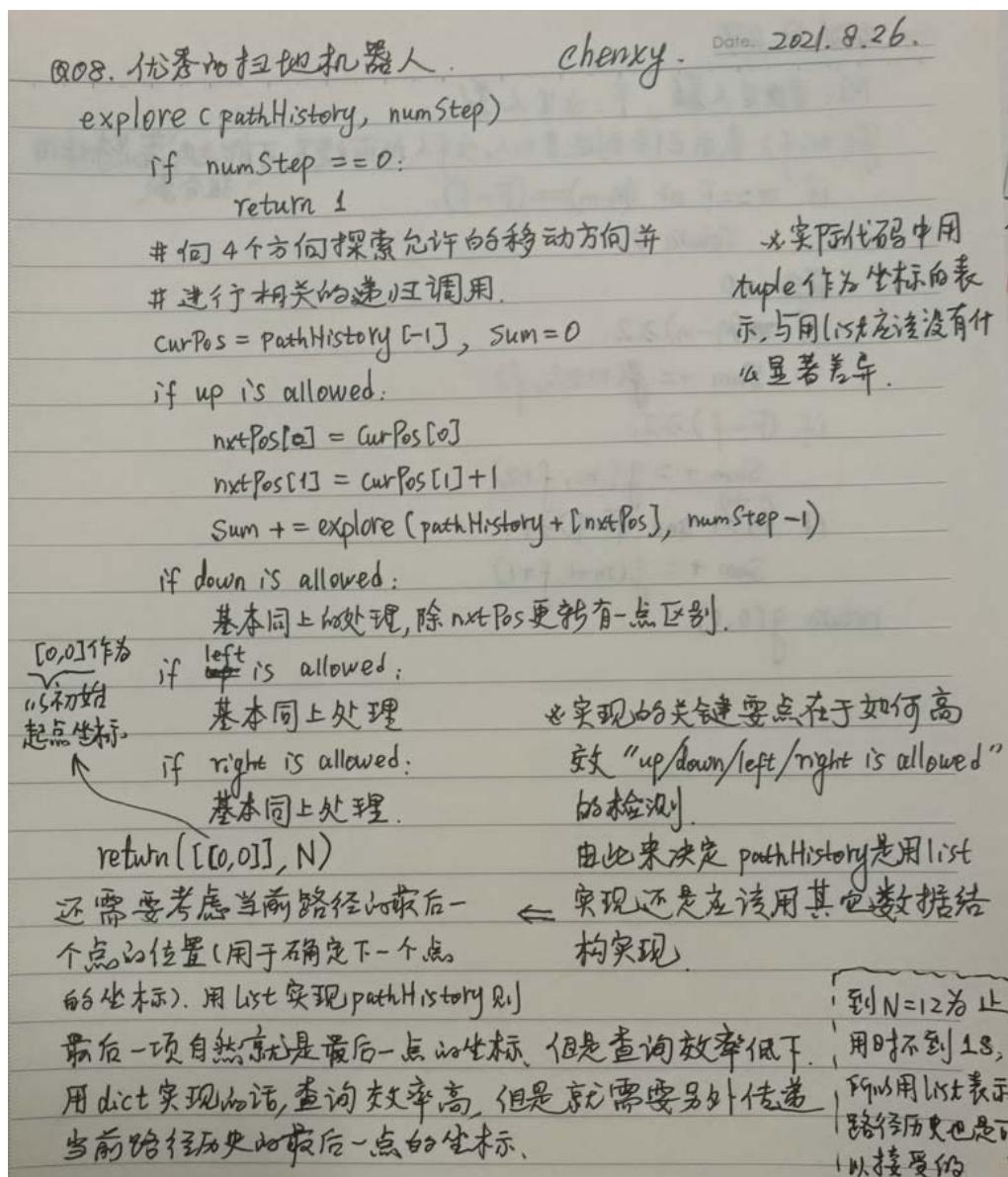
假设有不会反复地清扫某一个地方的扫地机器人，它只能前后左右移动。举个例子，如果第 1 次向后移动，那么连续移动三次后，它的轨迹会出现 9 种情况，如下所示(0 表示起点的位置， $k=\{1,2,3\}$ 表示经过 k 步移动后机器人所在的位置)。



求这个机器移动 12 次时，有多少种移动路径？

8.2 解题分析

深度优先路径遍历搜索问题。



9. Q09: 落单的男女

9.1 问题描述

人们聚集在某个活动会场上，根据到场顺序排成一排等待入场，活动的主办人员，想把人们从队列的某个位置分成两组，想要让分开的两组里至少有一组（原题是每一组）的男女人数都均等。但如果到场顺序不对，可能出现无论怎么分，两组都不能出现男女均等的情况。

举个例子，有 3 位男性、3 位女性以“男男女男女女”的顺序到达，如图所示，无论从队列的哪个位置分开，两组的男女人数都不均等。但如果到场的顺序为“男男女女男女”，那么只需要在第 4 个人处分组就可以令分开的两组男女人数均等了。

求男性 20 人、女性 10 人的情况下，有多少种到场顺序会导致无论怎么分组都没有任何一组的组内男女人数能够均等？（原书中表述的有问题，因为如果要求两组中男女人数都相等，男女总人数就必然相

等。而本题题设就是男女人数不等，这样无论按什么顺序到来都满足条件，过于 trivial 的问题)

9.2 解题分析

按照动态规划的思路进行子问题分解。

令男女人数分别为 $M(\text{Male})$ 和 $F(\text{Female})$ ，令当前已经到达的男生和女生人数分别为 m 和 f 。记这种情况下接下来满足题设要求的人员到来顺序数为 $g(m,n)$ 。

接下来要么来男生，要么来女生：

如果来男生的话，则问题变成了子问题 $g(m+1,f)$

如果来女生的话，则问题变成了子问题 $g(m,f+1)$

因此可以得到递推关系式：

$$g(m,f) = g(m+1,f) + g(m,f+1)$$

接下来要确定边界条件。

Case1：如果在某个时刻，到达的男生人数 m 和女生人数 f 相等，就相当于找到一个可以“均等分割”的情况（不符合题设且无需继续搜索），因此应该返回 0。但是需要注意的是，必须在 $m>0$ 的条件下以上论断才有效，否则的话，一开始就是 $m=n=0$ 的情况，这个没有意义。

Case2：如果在某个时刻，尚未到达的男生人数 $M-m$ 和尚未到达的女生人数 $F-f$ 相等，也就相当于找到一个可以“均等分割”的情况（不符合题设且无需继续搜索），因此应该返回 0。同样需要注意要在 $(M-m)>0$ 的条件下以上论断才有效。

Case3：前两种情况都是属于搜索失败的情况，接下来确定搜索成功的情况。第一感是男生已经到齐而还没有碰到可以平均分割，或者女生已经到齐而还没有碰到可以平均分割，都是属于找不到“均等分割”的情况。但是这里有漏洞。比方说， $M=20, F=10$ ，女生已经到达 10 人，男生只到了 9 人，此时虽然尚未满足“均等分割”，但是接下来再来一个男生就凑出“均等分割”了。所有还得在以上条件的基础上加上一个约束条件，即任一方（记为 A 方）已经到齐而另一方（记为 B 方）到达的人数也不少于 A 方时，此时没有到达“均等分割”条件，后续只有 B 方人员到来，就不可能再出现“均等分割”的情况了。

以下用递归+memoization 的方式实现（无它，就是这个代码写起来简单，哪怕效率低一些也值得^-^）。

10. Q10：轮盘的最大值

10.1 问题描述

轮盘游戏被称为“赌场女王”。流传较广的轮盘数字排布和设计有“欧式规则”和“美式规则”两种。

欧式规则:

0,32,15,19,4,21,2,25,17,34,6,27,13,36,11,30,8,23,10,5,24,16,33,1,20,14,31,9,22,18,29,7,28,12,35,3,26

美式规则:

0,28,9,26,30,11,7,20,32,17,5,22,34,15,3,24,36,13,1,00,27,10,25,29,12,8,19,31,18,6,21,33,16,4,23,35,14,2

下图是欧式规则的轮盘。



下面我们找到这些规则下“连续 n 个数字之和”最大的位置。举个例子，当 $n=3$ 时，按照欧式规则得到的最大的组合是 36,11,30 这个组合，和为 77；而美式规则下则是 24,36,13 这个组合，得到的和为 73。（图中所示轮盘为欧式规则的轮盘，注意轮盘是圆形的）

问题：当 $2 \leq n \leq 36$ 时，求连续 n 个数之和最大的情况，并找出满足条件“欧式规则下的最大和小于美式规则下的最大和”的 n 的个数。

10.2 解题分析

求连续 n 个数字的平均（本题是求和，但是求和与求平均只差一个平均系数，求平均也是先求和再除以参与求和的数据的个数）在信号处理领域中称为求移动平均（moving average, or running average），其计算方式有一个小小的 trick。令数据序列用 $x[k]$ 表示，则从第 m 数开始的连续 n 个数的累加和的计算可以表示如下：

$$\begin{aligned} sum[m] &= \sum_{m}^{m+n-1} x[m] \\ &= x[m] + x[m+1] + \cdots + x[m+n-1] \\ &= -x[m-1] + x[m-1] + x[m] + \cdots + x[m+n-2] + x[m+n-1] \\ &= sum[m-1] - x[m-1] + x[m+n-1] \end{aligned}$$

这样就得到了一个递推关系，从上一个连续和 $sum[m-1]$ 开始，只要把减去最前面一个数，加上后面

一个数就可以得到新的 $\text{sum}[m]$ 。这样可以极大地降低运算复杂度，在信号处理领域是常用技巧。

本题还有另外一个需要注意的要点就是轮盘是圆形的，用线性数组表示轮盘上的数字排列数组的话，一部分的连续和涉及到头上一部分数字和尾巴上一部分数字的求和。最基本的做法就是用地址对数字长度进行求模运算。也可以用对数组进行线性扩展的方法以避免地址求模的处理。由于这个问题比较简单，本题解就用“简单粗暴”的前者了。

11. Q11: 斐波那契数列

11.1 问题描述

已知斐波那契数列：1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...（从第3个数字开始，每个数字等于它前两个数字之和）。如下例所示，斐波那契数列中的有些数可以被它各个数位上的数字之和整除（前几个斐波那契数本身是个位数，这个结果是显而易见的）。

$$2 \rightarrow 2 \div 2$$

$$3 \rightarrow 3 \div 3$$

$$5 \rightarrow 5 \div 5$$

$$8 \rightarrow 8 \div 8$$

$$21 \rightarrow 21 \div 3 \dots 2+1=3, \text{ 因而除以 } 3$$

$$144 \rightarrow 144 \div 9 \dots 1+4+4=9, \text{ 因而除以 } 9$$

请继续例中的计算，求出后续5个最小的能被整除的数。

11.2 解题分析

这道题比较简单。简单地进行迭代循环即可：

(1) (以前两个1为初始条件开始)计算下一个斐波那契数 F_k

(2)

(3) 计算斐波那契数的各个数位的数字之和 $F_k_digitSum$

(4) 判断 F_k 是否能被 $F_k_digitSum$ 整除

计算斐波那契数列的常用技巧是，只记忆两个斐波那契数（当前 cur 和上一个 $prev$ ），然后据此计算下一个 nxt ，然后再更新为 $cur \rightarrow prev$, $nxt \rightarrow cur$ 。在 python 可以很方便用一条语句完成（参见以下代码）。

```
while cnt < N:
```

```
    prev, cur = cur, prev+cur
```

CSDN @笨牛慢耕

在以下代码中用两种方法实现了计算各个数位的数字之和 `Fk_digitalSum` 的方法。

方法一非常 `python-style` 的实现方法。将整数变换为字符串，再变换为 `list`，然后再将列表元素（字符）变换回整数并求和。

```
# Alternative 1
```

```
rslt = sum([int(s) for s in list(str(num))])
```

CSDN @笨牛慢耕

方法二是常规且通用的循环迭代方法。每个循环中取当前值的个位数并求和，然后将当前值整除 10 得到新的当前值，直到当前值为 0 为止。

12. Q12: 平方根数字

12.1 问题描述

求在计算平方根的时候，最早让 0~9 的数字全部出现的最小整数。注意这里只求平方根为正数的情况，并且请分别求包含整数部分的情况和只看小数部分的情况。

例) 2 的平方根 : 1.414213562373095048... (0~9 全部出现需要 19 位)

12.2 解题分析

这道题目要求比较含糊。

其实是找平方根（含整数部分或不含整数部分的）前十位数字正好让 0~9 的数字全部出现（且不重复）的最小整数—即构成 0~9 的一个排列（permutation）。

关键在于题目要求既要求“最早”又要求“最小”。假定按自然数从小到大遍历，只有找到了一个平方根前十位数字正好让 0~9 的数字全部出现的数，你才能确信找到了满足题目条件的最小整数。在没有找到满足“平方根前十位数字正好让 0~9 的数字全部出现”的数之前，你无法确定后面是不是存在满足“平方根前十位数字正好让 0~9 的数字全部出现”的数，所以你只能继续找下去。所幸，满足“平方根前十位数字正好让 0~9 的数字全部出现”的数是存在的。

基本步骤如下：

从小到大遍历自然数：

对每一个自然数：

求其平方根

转换成字符串

取前十个（除小数点以外的字符），或取小数点后的前十个字符

判断这十个字符是否恰好包含所有的 0~9

如果满足条件则退出，否则继续搜索下一个

代码中用以下判断前十个字符是否满足题设条件：

```
len(set(list(first10))) == 10
```

这是因为 set 会自动删除重复元素，因此如果 set 的长度是 10 而其可能的元素又只能是 0~9 的话，那就必然满足条件。

13. Q13: 满足字母算式的解法

13.1 问题描述

所谓字母算式，就是用字母表示的算式，规则是相同字母对应相同数字，不同字母对应不同数字，并且第一位字母的对应数字不能是 0。

譬如给定算式 We * love = CodeIQ，则可以对应填上下面这些数字以使之成立。

W = 7, e = 4, l = 3, o = 8, v = 0, C = 2, d = 1, I = 9, Q = 6

这样一来，我们就能得到 $74 * 3804 = 281496$ 这样的等式。使这个字母算式成立的解法只有这一种。

求使下面这个字母算式成立的解法有多少种？

READ + WRITE + TALK = SKILL

13.2 解题分析

基本步骤如下：

Step1: 从输入字符串中提取出表示数字（即除运算符和等号外）(不重复, or unique) 字符集合

Step2: 遍历以上所有字符与数字(0~9)的映射方式，假定有 k 个字符，则有 $P(10,k)$ 中排列，每一种排列对应一种映射方式

Step2-1 将每一种映射方式代入到输入字符串得到字符串算式

Step2-2 判断所得到的字符串算式是否是合法的

Step2-3 分别评估字符串算式的 LHS (Left-hand-side: 左手边表达式) 和 RHS (Right-hand-side: 右手边表达式) 并判断是否相等

在 Step2 中用 python itertools 模块中的 permutation() 生成所有的组合。

在 Step2-3 中用 pyhton 内置的 eval() 函数进行字符串形式的算式值的评估

在 Step2-2 中判断字符串表达式是否合法依据的规则是多位数字的操作数的第一个字母不能为 0。首先基于 ‘=’ 的位置分割得到 LHS 和 RHS。

RHS 中因为没有运算符所以比较简单，只要有多位数字（长度大于 1）且首位为 0 就表示是非法表达式。

LHS 的情况比较复杂，分以下三种情况考虑：

- (1) 第一个操作数的判断：如果 LHS[0] 为 0 且 LHS[1] 仍为数字则表明肯定非法
- (2) 其后，搜索每个运算符，如果运算符后的第一个字符为 0 且第二个字符仍为数字，则为非法
- (3) LHS 的最后一个字符不能为操作符—这个判断，基于以下假设其实并不需要

当然以上判断方法是基于原输入字符串表达式肯定可以表达成一个合法的算式，比如不会有两个连续的运算符出现，等等。

14. Q14: 国名接龙

14.1 问题描述

FIFA 世界杯对足球爱好者而言是四年一次的盛事。下面我们拿 2014 年世界杯参赛国的国名做个词语接龙游戏。不过，这里用的不是中文，而是英文字母（忽略大小写）。2014 年 FIFA 世界杯的 32 个参赛国参见代码中的国名列表。

举个例子，如果像下面这样，那么连续 3 个国名之后接龙就结束了（因为以上国名列表中没有英文名称以 D 开头的国家）。

“Japan” → “Netherlands” → “Switzerland”

问题：假设每个国名只能使用一次，求能连得最长的顺序，以及相应的国名个数。

14.2 解题分析 1—深度优先搜索

本题可以用深度优先搜索算法来解决。

针对某个国家开始的情况，以深度搜索的方式搜索每一条可行的接龙路径。按照每条接龙路径一直搜索到底（直到当前接龙路径的最后一个国家再也找不到下一个可以接上的国家了）。此时将当前接龙的长度与保存的最大长度的接龙（在实现中可以作为全局变量）进行比较并根据比较结果相应更新。

沿每条路径深度搜索时，用 `visited` 和 `unvisited` 分别管理已经搜索过的国家和尚未搜索过的国家。进一步的 `exploration` 仅从 `unvisited` 中选取下一个探索对象，因此省掉了“是否已被访问过”的检查判断，另一方面，`visited` 是按照访问顺序存入被访问对象，所以其中存储的就是当前搜索的接龙顺序。`visited` 和 `unvisited` 都需要以栈的方式进行管理，因此如果用递归调用的方式实现的话，将它们作为递归函数的接口参数传递即可；如果用循环方式实现的话，则需要注意显式的入栈和出栈管理。

注意：本题是要求接龙中同一国名只能使用一次，这意味着路径不能形成 `loop`。正因为这个，才可以以上述的 `visited`、`unvisited` 的方式进行分割以实现节点（每个国名就是一个节点）不重复访问的管理。在有些问题中，允许节点在路径上重复出现，但是不允许 `edge` 重复，则需要另外的防止重复访问的管理机制。

此外，最长接龙搜索的结果依赖于从哪个国家开始，因此需要在针对以某个国家为起点的深度优先搜索的基础上再追加一层外层循环，遍历国家名字列表中的每一个国家作为起始国家分别进行接龙搜索。

14.3 解题分析 2—广度优先搜索

本题的目标是要求最长接龙，这可以看作是一个图搜索中的最长路径问题，可以用广度优先搜索(BFS: Breadth First Search) 策略来解决。

接龙游戏的一个潜在规则是不能形成环路，即不能重复访问同一个节点。

14.3.1 BFS 基本要素

队列：以队列的方式存储带访问节点，等价于根据与出发点的距离（由近及远）进行搜索

已访问节点的记忆

Layer 数（也即距离出发点的距离）的记录

14.3.2 最长路径问题 vs 最短路径问题

最短路径问题相对来说要简单一些，只要找到一个符合搜索条件的 `target` 即可以停止搜索。

而最长路径问题则需要穷尽整个图（不穷尽则无法判断是否找到了最大值）。因此在基于广度优先搜索的最长路径搜索问题中，必须一直搜索直到队列变空为止。

14.3.3 算法流程 1

以伪代码（不严格，python-style）形式表示的算法流程如下所示：

```
longestJieLong(names):
    maxLength = 0
    for start in names:
        Initialize: q, visited, etc
        q.push((start, 1))
        visited[start] = 1
        while (q is not empty):
            cur, curLayer = p.pop()
            for next in names:
                if next is not in visited and next can follow cur:
                    visited[next] = curLayer+1
                    p.push((next, curLayer+1))
            maxLength = max(maxLength, curLayer)
    return maxLength
```

但是以上算法流程只给出了最长蟠龙长度，并没有给出具体的蟠龙方案。

15. Q15: 走楼梯

15.1 问题描述

本题来自《程序员的算法趣题》中的第 15 题。

A 和 B 分别站在楼梯的底部和顶部，A 和 B 同步移动（即每次都是同时移动），A 向下走，B 向上

走，每次可以走的级数为{1,2,3,4}，但是两人每次的级数不一定相同。两人同时开始走，求共有多少种“两人最终在同一级碰头”的情况。

在一次移动的中途插肩而过不算在同一级碰头。比如说两个相距 1 级，A 向上走 1 级，同时 B 向下走 1 级，两个人中途肯定在某个瞬间相遇，但是完成这一次移动后，A 将停留在 B 的上一级楼梯上。

有 4 级楼梯时示例如下：

15.2 递推表达式

假设两人在某一时刻相距的级数为 N，相向而行走到碰头的可能情况数用 f(N) 表示。

考虑 A 向上走 Ak 级楼梯，而 B 向上走 Bk 级楼梯，则（假设两人还没有走过头）此时两人还相距 (N-Ak-Bk) 级，这是原始的“相距为 N”的求解问题的一个子问题，其解记为 f(N-Ak-Bk)。对合法的(Ak,Bk) 组合所代表的子问题进行遍历求和即可得到 f(N)，因此可以得到递推表达式如下所示：

$$f(N) = \sum_{Ak=1}^{\min(4,N-1)} \sum_{Bk=1}^{\min(4,N-1)} f(N - Ak - Bk)$$

初始条件：

$$f(n) = \begin{cases} 0 & n < 0 \\ 1 & n = 0 \\ 0 & n = 1 \end{cases}$$

“n<0”表示两人错过了在同一级碰头而且已经走过头了，没有碰头，因此返回 0

“n=0”表示两人成功地碰头，因此返回 1

“n=1”表示两人相距 1 级，因为最小移动级数为 1 级，两人已经不可能成功碰头，因此返回 0（这个初始条件不是必须的。可以由移动规则加上“n<0”推导出）。

可以用递归的方式（再加上 memoization technique）来时求解递推表达式。

当然也可以用动态规划的策略来实现。

16. Q16: 3 根绳子折成四边形

16.1 问题描述

本题来自《程序员的算法趣题》中的第 16 题。

假设分别将 3 根长度相同的绳子摆成 3 个四边形。其中 2 根摆成长方形，另外 1 根摆成正方形。这时，当长度选择适当的话，会出现两个长方形的面积之和等于正方形的面积的情况（假设绳子长度和各四边形的边长都是整数）。

此外，将同比整数倍的结果看作是同一种解法。

16.2 初始解法—解法 1

令绳子长度为 $L=4*a$ ，即正方形的边长为 a 。令另外两个长方形的较短的边分别为 x_1 和 x_2 ，并且不失一般性可以假定它们满足关系： $x_1 \leq x_2 < a$ 。

这个问题可以通过遍历搜索来解决。其中，对于任意的 a ，基于以上假设必然有 $x_1^2 \leq x_2^2 < a^2$ ，进一步可以推导出 $x_1 < a/\sqrt{2}$ 。这样可以将 x_1 的搜索范围缩小。代码如下：

插入代码

16.2.1 三者互素的判断

其中，将同比整数倍的结果看作是同一种解法，这意味着， $\{a,x_1,x_2\}$ 必须满足三者互素，换句话说三者的最大公约数为 1，才被计算为一个独立的答案。容易证明，在本题中，如果 $\{a,x_1,x_2\}$ 满足题设要求的平方和关系的话， $\{a,x_1,x_2\}$ 三者互素等价于任何两者互素。因此，在代码中仅判断 a 和 x_1 是否互素（用 python 中 math 模块中的 gcd() 函数）。

16.3 改进解法—解法 2

在解法 1 中，每次条件判断都是直接计算 “ $x_1*(2*a-x_1) + x_2*(2*a-x_2) == a*a$ ”，这会导致非常多的重复计算。事实上针对每个 a （即针对最外层的每个循环），只需要计算一次 $a*a$ ；同理针对每个 x_1 （即针对第 2 层的每个循环），只需要计算一次 $x_1*(2*a-x_1)$ 。这样可以将代码优化如下：

插入代码

运行结果表明这一简单的改进导致了运行时间下降了 60%！

16.4 改进解法—解法 3

进一步观察可以发现，在以上处理流程中，是找到了满足平方和关系的解再来判断是否满足三者互素。如果交换顺序先判断是否满足互素关系，仅针对满足互素关系的集合判断是否满足平方和关系，对运行时间会不会有改善效果呢？这个可能取决于互素关系的判断和平方和关系的判断哪个更耗时。不过这里尝试以下，改进后的代码如下：

插入代码

运行结果表明这一简单的改进导致了运行时间进一步下降了 30%！至少说明在本实现中，平方和关系的判断比互素关系的判断更为耗时。

16.5 改进解法—解法 4

进一步思考可以发现，当三个四边形满足题设条件时，假设其中一个长方形（不失一般性记为长方形 1）的边长分别为 $a-x$ 和 $a+x$ ，则长方形 2 的面积必然为 $a^2 - (a-x)(a+x) = x^2$ 。反过来，令长方形 2 的边长分别为 $a-y$ 和 $a+y$ ，可以得到长方形 1 的面积必然为 $a^2 - (a-y)(a+y) = y^2$ ，也即三者的面积（的平方根）构成一组勾股数的关系！反之，容易证明，任何满足 $a^2 = x^2 + y^2$ 的一组数 $\{a, x, y\}$ 都对应着满足题设条件的三个四边形。由此可知，求满足题设条件下的解等价于寻找勾股数！

17. Q17：30 人 31 足游戏

17.1 问题描述

本题来自《程序员的算法趣题》中的第 17 题。

男生和女生一起参加“30 人 31 足”比赛。由于女生体力有劣势，所以两个女生排在一起的话整个队伍就会在体力上处于明显不利地位。所以原则上尽量不让女生相邻排列，称没有女生相邻排列的排列方式为“合理”排列。

请问，当总共 N 个男生和女生一起排成一排时，一共有多少种合理的排列方式？

假设这里男生之间、女生之间不考虑区分（即不考虑男生之间的相对位置的不同，也不考虑女生之间的相对位置的不同）。举个例子，4 个人（4 人 5 足）的情况下共有 8 种排列方式，如下所示（B 表示 boy，G 表示 girl）：

BBBB, GBBB, BGBB, BBGB, BBBG, GBGB, BGBG, GBBG

17.2 解法 1—作为计数问题的闭式解

这个问题可以当作一个计数问题来考虑，可以直接通过数学推导得到闭式(closed-form)解。

令男生人数为 N_b ，女生人数为 N_g ，首先它们满足关系：

$$N_b + N_g = N \quad (1)$$

由于不允许两个女生相邻排列，因此必然满足（考虑 N_b 个男生排成一排，包括两端在内一共有 N_b+1 个位置可以插入女生，即 N_b 个男生构成的队列中最最多可以插入 N_b+1 个女生以构成“合理”排列）：

$$N_b + 1 \geq N_g \quad (2)$$

结合(1)和(2)可得:

$$N_g \leq \left\lfloor \frac{N+1}{2} \right\rfloor, N_b \geq \left\lceil \frac{N-1}{2} \right\rceil \quad (3)$$

接下来,由于男生人数为Nb,包括两端在内一共有Nb+1个空档可以插入女生。同时又由于不允许女生相邻排列,所以不能有两个女生插入同一个空档。在这个条件下,所能构成的合理的排列数的问题就转变成了:Ng个女生放入Nb+1个位置有多少种安排方法?由于女生之间不区分,所以这是一个组合问题,总共有 $\binom{N - N_g + 1}{N_g} = C_{N-N_g+1}^{N_g} = C_{N_b+1}^{N-N_b}$,然后对所有Nf的可能取值进行求和即可得到(将N个人的队伍的合理排列方式数记为f(N)):

$$f(N) = \sum_{N_g=0}^{\left\lfloor \frac{N+1}{2} \right\rfloor} C_{N-N_g+1}^{N_g} \quad (4)$$

17.3解法2—作为计数问题解的递推关系式

将N个人的队伍的合理排列方式数记为f(N)。

考虑从左往右安排队伍,则最右边的人为最后一个安排的人。最后的人(即第N个人)有男生和女生两种可能性,分别讨论如下:

- (1) 如果最后一个人为男生,则对前N-1个人的排列方式没有约束条件。即这种情况下的排列方式等于前(N-1)个人的“合理”排列方式,也即是f(N-1)
- (2) 如果最后一个人为女生,则倒数第二个人必定为男生,进而同理可得对前N-2个人的排列方式没有约束条件。即这种情况下的排列方式等于前(N-2)个人的“合理”排列方式,也即是f(N-2)

基于以上讨论可以得到递推关系式,如下所示:

$$f(N) = f(N-1) + f(N-2) \quad (5)$$

这个递推关系式与斐波那契数列的递推关系式完全相同,但是本问题的解序列f(N)的解序列并不构成标准的斐波那契数列,原因在于初始化条件不同。为了利用(5)式解出f(N)的具体值,我们需要给出序列的最初几个值作为初始化条件。

N=1: 很显然f(1)=2,这里我们假定允许单独一个女生参赛(“1人2足”)

N=2: f(2)=3, “BB”, “BG”, “GB”

N=3: f(3)=5, “BBB”, “MBB”, “BMB”, “BBM”, “MBM”

...

因此,本问题的递推关系式表达的解为:

$$\begin{aligned} f(N) &= f(N-1) + f(N-2), N \geq 3 \\ f(1) &= 2 \\ f(2) &= 3 \end{aligned}$$

有兴趣的读者可以自行验证式(4)表示的解和式(6)表示的解式完全等价的。

18. Q18:水果酥饼日

18.1 问题描述

日本每月的 22 日是水果酥饼日。因为看日历的时候，22 日的上方刚好是 15 日，也就是“‘22’这个数字上面点缀着草莓”(如果将日语的 15 拆为 1 和 5 发音[イチゴ]，则与日语“草莓[イチゴ]”一词发音相同，而水果酥饼中最为著名的草莓酥饼)

切分酥饼的时候，要求切分后每一块上面的草莓个数都不相同。假设切分出来的 N 块酥饼上要各有“1~ N 个(共 $N(N + 1)/2$ 个草莓)”。但这里要追加一个条件，那就是“一定要使相邻的两块酥饼上的数字之和是平方数”。

举个例子，假设 $N = 4$ 时采用如下图的切法。这时，虽然 $1 + 3 = 4$ 得到的是平方数，但“1 和 4”“2 和 3”“2 和 4”的部分都不满足条件。



18.2 解题分析—深度优先搜索

本题等价于这样一个问题：求 $[1\dots N]$ 的一个圆排列(circular permutation)，使得任意相邻两个数之和为完全平方数。

作为一个直观的办法，可以考虑遍历 $[1\dots N]$ 的圆排列，然后检验每一个圆排列是否满足条件。然后对 N 进行从小到大遍历，直到找到一个 N 使得对于这个 N 存在一个满足条件的圆排列。但是对于 N 来说，圆排列数等于 $(N-1)!$ (注意，由于圆排列的对称性，如果只考虑各个数的相对位置的话， N 个线性排列对应于同一个圆排列)。对于略大的 N 这个需要遍历的排列数将无法忍受。

本题可以用深度优先搜索算法来解决。

从 1 开始，寻找与它的和能构成完全平方数的数(看作是 1 的子节点)，然后针对 1 的子节点进一步寻找与它的和能构成完全平方数的子节点...

除了问题的表象以外，本题作为一个深度优先搜索问题与 Q14 非常相似，可以用几乎相同的方式解决。

一个圆排列对应一个深度优先搜索路径。由于在圆排列中每个数只能用一次，所以用 `used` 和 `unused` 分别表示已经使用的数和尚未使用的数。进一步的 `exploration` 仅从 `unused` 中选取下一个探索对象，因此省掉了“是否已被访问过”的检查判断，另一方面，`used` 是按照访问顺序存入被访问对象，所以其中存储的就是当前搜索的接龙顺序。`used` 和 `unused` 都需要以栈的方式进行管理，因此如果用递归调用的方式实现的话，将它们作为递归函数的接口参数传递即可；如果用循环方式实现的话，则需要注意显式的入栈和出栈管理。

如果 `used` 的长度等于 N ，并且首尾的和也是完全平方数的话就表示找到了一个符号条件的圆排列。此时的 `used` 中存储的就是对应的圆排列。

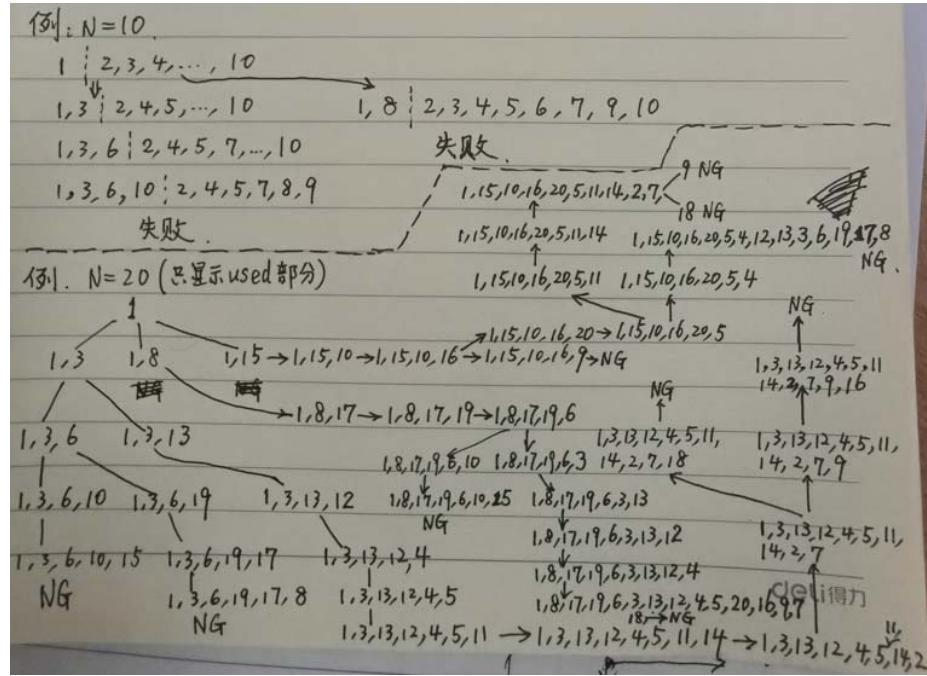
与 Q14 不同的是，从任何一点开始搜索圆排列都是等价的，所以不需要针对起点遍历，固定从 1 开始即可。

算法流程（python-style 伪代码）如下图所示：

Date: 2021.9.6.
chenxy.

```
def cutFruitCake(N):
    def explore(used, unused):
        if len(used) == N and used[0] + used[-1] 是完全平方数:
            return True
        cur = used[-1]
        cutOK = False
        for k, nxt in enumerate(unused):
            if cur + nxt 是完全平方数:
                rsit = explore(used + [nxt], unused[0:k] + unused[k+1:])
                cutOK = cutOK || rsit
        return cutOK
    used = [1] # Always start from 1.
    unused = [2, ..., N]
    return explore(used, unused)
```

以下是针对 $N=10$ 和 $N=20$ 的笔算示意图（如果可能的话，在纸上进行一些计算有助于建立对问题的直观感觉，我认为是非常有用。只不过 $N=20$ 时居然能走这么远一开始没想到否则的话可能就不敢挑战了）。



18.3 代码实现有两个小细节

- (1) 预算完全平方数列表用于提高是否平方数的检验效率。如果用 dict() 来存储完全平方数用于查询应该会更快

(2) 深度搜索是线性向前的（即每次只检查当前数跟它后面的数之和是否满足条件），但是到最后不能忘记首尾是否满足条件的判断：`used[0]+used[-1]`

19. Q19:朋友的朋友也是朋友吗

19.1 问题描述

“六度空间理论”非常有名。大概的意思是1个人只需要通过6个中间人就可以和世界上任何1个人产生间接联系。本题将试着找出数字的好友（这里并不考虑亲密指数）。

假设拥有同样约数（不包括 1）的数字互为“好友”，也就是说，如果两个数字的最大公约数不是 1，那么称这两个数互为好友。

从 1~N 中任意选取一个“合数”，求从它开始，要经历几层好友，才能和其他所有的数产生联系（所谓的“合数”是指“有除 1 以及自身以外的约数的自然数”）。

举个例子， $N = 10$ 时，1~10 的合数有 4、6、8、9、10 这 5 个。

如果选取的是 10，那么 10 的好友数字就是公约数为 2 的 4、6、8 这 3 个。

而 9 是 6 的好友数字（公约数为 3），所以 10 只需要经过 2 层就可以和 9 产生联系。

如果选取的是 6，则只需经过 1 层就可以联系到 4、8、9、10 这些数字。

因此 $N=10$ 时，无论最初选取的合数是什么，最多经过 2 层就可以与其它所有数产生联系。

问题：求从 1~N 中选取 7 个合数时，最多经过 6 层就可以与其它所有数产生联系的最小的 N。

(不知道是原文如此还是翻译不当) 本题的背景描述和最后的问题描述 (我认为) 都是有明显问题的：

- (1) 以上第 3 段话中，“... 才能和其他所有的数产生联系...”，从后文来看，这里应该是“... 才能和其他所有的合数产生联系...”。因为一个合数不可能与非自己因子的素数产生联系。
- (2) 最后的问题。直接从字面理解的话， $N=15$ ，比如说 $\{2,4,6,8,10,12,14,15\}$ 就满足条件。因为题目是要求最多经过 6 层，只要不超过 6 层（事实上只有 7 个数要产生联系也不可能超过 6 层）即可，并没有说一定必须到达 6 层。

问题描述应该修改如下：从 1~N 中选取 7 个合数，每个数字都可以与其它的 6 个数字建立联系。求使得 7 个数字中关系最远的两个数字必须经过 6 层才能产生联系最小的 N。

19.2 解题分析

记两个数要产生联系所需经过的层数为两数之间的“距离”，记为 $dist(x,y)$ 。题目的要求可以改写为：要求满足“从 1~N 中任选的 7 个合数中至少存在两个数，它们之间的距离为 6”的条件的最小的 N。

由于 N_1 和 N_7 之间的距离为 6，则 N_1 与其它 5 个数之间的距离必然分别为 1~5，不失一般性，我们假定 N_1 与 N_2 的距离为 1， N_1 与 N_3 的距离为 3，余者依此类推。

可以证明如下：

由于 $dist(N_1, N_7)=6$ ，则必然存在 1 个数与 N_7 距离为 1，且与 N_1 距离为 5，令该数记为 N_6 ；

由于 $dist(N_1, N_6)=5$ ，则必然存在 1 个数与 N_6 距离为 1，且与 N_1 距离为 4，令该数记为 N_5 ；...

余者依此类推。

满足条件的 N 的最小值必然是 $N_1 \sim N_7$ 中的最大值，即 $N=\max(N_1, N_2, \dots, N_7)$ 。所以寻找满足条件的最小的 N，就是寻找满足条件的最小的 7 个合数。

由于 $(N_1, N_2), (N_2, N_3), \dots, (N_6, N_7)$ 的距离分别为 1，要使得这些数最小，它们之间分别应该具有(大于 1 的)为质数的唯一的公约数，分别记为： $gcd(N_1, N_2)=a, gcd(N_2, N_3)=b, \dots, gcd(N_6, N_7)=f$ (gcd : Greatest Common Divisor, 表示最大公约数)。因此， N_2 的最小取值为 $a \cdot b$ ， N_3 的最小取值为 $k_3 \cdot b \cdot c$ ，...， N_6 的最小取值为 $e \cdot f$ ，而 N_1 和 N_7 由于分别可能只有一个好友，它们的取值可以写为 $k_1 \cdot a$ 和 $k_7 \cdot f$ 。

显而易见，将 $\{a, b, c, d, e, f\}$ 取最小的 6 个素数，然后分别令 $k_1=a, k_7=f$ 可以满足以上条件 (距离条

件，且 $\max(N1, \dots, N7)$ 最小)。

(好吧，承认失败。本来我是想给出一个严格地证明，但是最后看起来如此显而易见的事情真要给出逻辑严谨的证明似乎并不是一件容易的事情，最后还是得靠一个“显而易见”敷衍了事。**只叹：书到用时方恨少，数学学得太潦草**) 先搁在这里，后面有机会再回头看能不能给出给严谨的说明。

基于以上讨论可得，本题求解流程如下所示：

(1) 取最小的 6 个素数 $\{2, 3, 5, 7, 11, 13\}$

(2) 遍历这 6 个素数的排列，对每个排列，求 $\{a^*a, a^*b, b^*c, c^*d, d^*e, e^*f, f^*f\}$ 中的最大值

比较各排列所得最大值，取其中最小值，即为本题的解。

20. Q20:受难立面魔方阵

20.1 问题描述

西班牙有个著名景点叫圣家堂，其中“受难立面”上主要画着耶稣从“最后的晚餐”到“升天”的场景，其中还有一个如下图所示的魔方阵，因“纵、横、对角线的数字之和都是 33”而闻名（据说耶稣辞世时是 33 岁）。

如果不局限于由纵、横、对角线的数字相加，那么和数为 33 的组合有 310 种之多（网上有很多“4 个数字相加……”这样的问题，如果限定只能由 4 个数字相加，则是 88 种）。

1	14	14	4
11	7	6	9
8	10	10	5
13	2	3	15

问题：使用这个魔方阵，进行满足下列条件的加法运算，求“和相同的组合”的种数最多的和。

条件：

- (1) 不限于由纵、横、对角线上的数字相加
- (2) 加数的个数不限于 4

其实就是给定 16 个数中，求其中任意组合的相加和，其中得到相同的和的组合种类数最多的和。

20.2 解法 1—笨办法

这是基于直感想到的第一个办法（通常我会先从最 naïve 的方法开始着手，虽然笨拙，但是也有它的价值）。一上来就能直中靶心想到最优解毕竟不是凡人所能奢望的事情，循序渐进或许更符合普通人的步调。

针对每一个可能的和 target（和的可能数为以上所有数字总和加 1，把 0 也当作一种可能考虑，但是只考虑给定的数组中的数都为非负数的情况），扫描所有的组合，确认和为 target 的组合种类数。

然后比较所有可能的 target 对应的组合种类数。求解流程如下所示：

MaxCnt = 0

遍历 target = 0...sum(nums), for each of them:

targrtCnt = 0

遍历 nums 中所有数的所有各种长度(len(nums)+1)的组合，

检验其和是否等于 target

如果是的话，则 targrtCnt 加一

更新 maxCnt: maxCnt = max (maxCnt, targrtCnt)

代码参见下面 shouNanMoFang1()。

20.3 解法 2—逆向思维

解法 1 太慢了，在 nums 数组中为 16 个数时就需要秒级的时间。16 个数的所有可能长度的组合数是多少呢？恰好是二项式公式，如下所示：

$$C_{16}^0 + C_{16}^1 + \dots + C_{16}^{16} = \sum_{k=0}^{16} C_{16}^k = 2^{16}$$

Nums 中的数为 16 个时不过 65536 种组合，如果魔方阵为 5*5=25 是，组合数将变为 2^{25} ，比 $4*4=16$ 的情况增大了 512 倍，显然按解法 1 是无法承受的。

解法 1 的问题在于，以 target 为着眼点，对每一个 target 都进行了全组合扫描，这导致巨大的浪费。如果反过来的话，其实只需要进行一遍全组合扫描，针对每一个组合根据其和给对应 target 的计数器进行计数即可！这样运行时间就缩短为 $(1/\text{sum}(nums))$ 了。

在以下代码中用哈希表（python dict）来记录对应各可能 target 的组合数计数器。

代码参见下面 shouNanMoFang2()。

虽然解法 2 相比解法 1 有约两个数量级的效率提升，但是在 5*5 魔方阵（即 25 个数）的情况下仍然需要数十秒的时间。对于更大的魔方阵（比如说 6*6）就无能为力了。

20.4 解法 3—动态规划

这里所说的动态规划的方法是参考原书的提示。但是原书提示就是寥寥一句话“每次设置一个数。。。”。怎么肥四啊？有点类似于数学证明中的“显而易见、易证”之类的（人与人之间的脑回路的差别有这么大嘛）。查阅了几个 CSDN 博客上的解也基本上就是直接上了一段代码。。。肝疼。。。(**copy 一段代码很简单，或者把 ruby 代码翻译成别的语言的代码也很简单**) 不过我还是希望能用自己的话把这个事情说清楚。本题的动态规划解法的要点如下：

按顺序考虑魔方阵（代码中的 `nums`）的每个数的处理。

令 `targetCnt` 为一个 2 维数组（或者说一张表格—是的，动态规划的最基本的方法就是填表！），记录在使用 `nums` 不同子集时所有可能和值 `targetSum` 的组合种类数。`targetCnt[k, m]` 代表只考虑 `nums` 中前 $k+1$ 个数（即 $\{nums[0], \dots, nums[k]\}$ ）的条件下的和为 m 的组合种类数。

假设已经处理完 `nums[0], \dots, nums[k-1]` 了。当前 `targetCnt[k-1,:]` 的内容相当于是只考虑 $\{nums[0], \dots, nums[k-1]\}$ 的所有可能组合所得到的结果。接下来追加考虑 `nums[k]`，很显然，对于 $targetSum=m$ 而言，只考虑 $\{nums[0], \dots, nums[k-1]\}$ 且和等于 $(m - nums[k])$ 的所有组合再加上 `nums[k]` 就构成了和为 m 的组合了，由此可以得到递推关系式如下(为了让公式简洁一些，将第 1 个坐标用作下标，第 2 个坐标用作上标)：

$$targetCnt_k^m = \begin{cases} targetCnt_{k-1}^{m-n} + targetCnt_{k-1}^{m-nums[k]}, & \text{if } m - nums[k] > 0 \\ targetCnt_{k-1}^m, & \text{otherwise} \end{cases}$$

这样，本题的解题过程就变成了经典的动态规划填表的过程了。进一步，与一般的动态规划问题比如背包问题中不同的是，本问题中 $targetCnt_k^m$ 只依赖于 $targetCnt_{k-1}^{m'}$ ，而且 $m' < m$ ，因此我们事实上并不需要维护一张 2 维的表格，而只需要维护一个 1 维的数组就可以了。只不过需要注意，在更新 `targetCnt` 时要按上标从大到小更新。

最后，作为动态规划解法，当然必须要定义初始条件或者边界条件。本题的边界条件可以定义为，在空组合（或者说从 `nums` 中选择 0 个数）时， $targetSum=0$ 时的组合数为 1， $targetSum>0$ 时的组合数为 0，即：

$$targetCnt_0^m = \begin{cases} 0, & m > 0 \\ 1, & m = 0 \end{cases}$$

代码参见以下 `shouNanMoFang3()`。

21. Q21 :异或运算杨辉三角形

21.1 问题描述

本题来自《程序员的算法趣题》中的第 21 题。

著名的杨辉三角形（帕斯卡三角形）的计算法则是“某个数值是其左上角的数和右上角的数之和”。这里用异或运算代替单纯的和运算，便得到所谓的“异或运算版杨辉三角形”。如下图所示：

【插入图片】

问题：在异或运算版杨辉三角形中，自上而下计算时，第 2014 个 0 会出现在哪一层。更一般地，第 $n(>0)$ 个 0 会出现在哪一层？

如上图所示，对比标准杨辉三角形和异或运算版杨辉三角形，可以发现，后者中的 0/1 与前者中的偶数/奇数是恰好对应的。因此这个问题相当于是问：在标准杨辉三角形中第 n 个偶数会出现在第几层？

21.2 解法 1—直观方法

直观的方法就是按照迭代的（iterative）方式从第 1 层开始从上而下计算每一层，计算过程中对 0 的个数以及层数(count from one)进行计数，当 0 的个数满足题设要求时即结束退出。

在每一层的计算中，要注意杨辉三角形的一个特征是每一个层的首尾两个数字都是 1（不管是原始版还是异或运算版），这两个数字不需要也无法通过以上计算法则计算得出。

【插入伪代码示意图】

当然，其中的 `cmp_flg` 并不是必须的。因为可以把结束判决条件放到 `for-loop` 外面，判决条件由 `(zeroCnt==2014)` 改为 `(zeroCnt>=2014)`。

具体实现代码如下：

【插入代码】

21.3 解法 2—利用比特运算

以上的直观方法不管是对于标准杨辉三角形中求偶数出现情况的问题还是异或运算杨辉三角中求 0 出现情况的都可行。但是，后者由于只要求异或运算，这就为利用 bit-wise 二进制操作运算特征来提高效率提供了可能性。

每一层的中间的数是由（在上一层中）它的左上的数和右上的数的异或运算而得。如果将每一层表达为一个整数（每个比特表示该层中的每一个数），则以上运算等价于是将上一层的数左移 1 比特后再与原数进行按比特异或运算而得，如下图所示：

【插入图片】

当用于表示每一层的整数都初始化为 0，且假定整数左移时右端补入 0（各种编程语言中一般的缺省动作都是这样的），从以上运算示例可以看出，连两端的两个 1 也自然地被这样的计算所覆盖，因而不需要额外的初始化处理。

这里唯一需要注意的是，当计算的层数太多，所需要的比特数超过整型数表达范围的话，以上运算法则就行不通，需要进行数的拼接处理才能对应。在 python3.x 中支持超长整数的表达，所以不必担心这个问题。

21.4 实现代码

在代码实现中，通过参数 N 指定所要搜索的 0 的序号，这样可以搜索任意某个序号的 0 所出现的行数。另外，从运行结果来看，两种实现方法的运行时间没有本质的差异。

22. Q22:不缠绕的纸杯电话

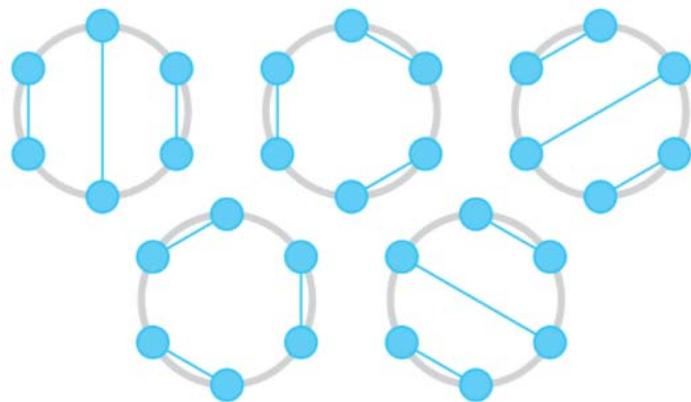
22.1 问题描述

用绳子连接纸杯制作“纸杯电话”——这应该勾起了很多人对理科实验的回忆。如果把绳子拉直，对着一边的纸杯讲话，声音就可以从另一边的纸杯传出。

假设有几个小朋友以相同间隔围成圆周，要结对用纸杯电话相互通话。如果绳子交叉，很有可能会缠绕起来，所以结对的原则是不能让绳子交叉。

举个例子，如果有 6 个小朋友，则只要如下图一样结对，

就可以顺利用纸杯电话通话。也就是说，6 个人的时候，有 5 种结对方式。



求：有 16 个小朋友的时候，一共有多少种结对方式？

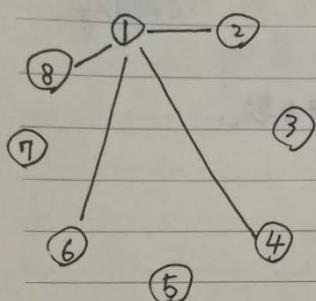
22.2 解题分析

以 $f(N)$ 表示当有 N 个小朋友时的结对组合方式数可以推导出以下递推关系式：

$$f(N) = \sum_{m=0}^{\frac{(N-2)}{2}} f(N - 2 - 2m) * f(2m) \quad N \geq 2, \text{ and } N \text{ is even}$$
$$f(0) = 1$$

递推过程以及笔算的结果如下所示：

Date. 2021/9/7. Q22 不缠绕的纸杯电话. chenxy.



考虑 $N=8$ 的情况，记此时结对方式

表示为 $f(N) = f(8)$.

① 只能与②、④、⑥、⑧ 等 4 人结对。

①-②，其余 6 人结对情况数可以表达为 $f(6)$

①-④，如上图所示，分成了两个区，一边是 2 个人，另一边是 4 个人，所以总的结对情况可以表示为 $f(2) \cdot f(4)$.

以下可依此类推。

为了方便描述，假设 $f(0) = 1$ (①-② 情况，可以认为以①-② 为分界线，当然很明显 $f(2) = 1$ 。其中一边 0 个人)。

基于以上考虑可知： $f(8) = f(6) \cdot f(0) + f(4) \cdot f(2) + f(2) \cdot f(4) + f(0) \cdot f(6)$.

同理可以得出当 N 为偶数时，有以下递推关系式：

$$\begin{aligned} f(N) &= f(N-2) \cdot f(0) + f(N-4) f(2) + \dots + f(2) \cdot f(N-4) + f(N-2) \cdot f(0) \\ &= \sum_{m=0}^{\frac{(N-2)/2}{2}} f(N-2-2m) \cdot f(2m) \quad \forall N \geq 2, \text{ 且 } N \text{ 为偶数}. \end{aligned}$$

由此可以推导得出：

$f(2) = f(0) \cdot f(0) = 1$. 例如，假设 $f(0) = 1$. 当然这个假设只是为方便计算，不是绝对必要的。关键是这个假设与本问题是相容的、合理的。

$$f(6) = f(4) \cdot f(0) + f(2) \cdot f(2) + f(0) \cdot f(4) = 5$$

$$f(8) = f(6) \cdot f(0) + f(4) \cdot f(2) + f(2) \cdot f(4) + f(0) f(6) = 14$$

$$f(10) = f(8) \cdot f(0) + f(6) \cdot f(2) + f(4) f(4) + f(2) \cdot f(6) + f(0) \cdot f(8) = 38 \quad 42$$

$$f(12) = f(10) \cdot f(0) + f(8) f(2) + \dots + f(2) f(8) + f(0) \cdot f(10) = 144 \quad 132$$

$$f(14) = f(12) \cdot f(0) + f(10) \cdot f(2) + \dots + f(2) \cdot f(10) + f(0) \cdot f(12) = 445 \quad 429$$

$$f(16) = f(14) f(0) + f(12) \cdot f(2) + \dots + f(2) \cdot f(12) + f(0) \cdot f(14)$$

$$= 2 (445 + 144 + 2 \times 38 + 5 \times 11)$$

$$= 2 (589 + 138) = 2 \times$$

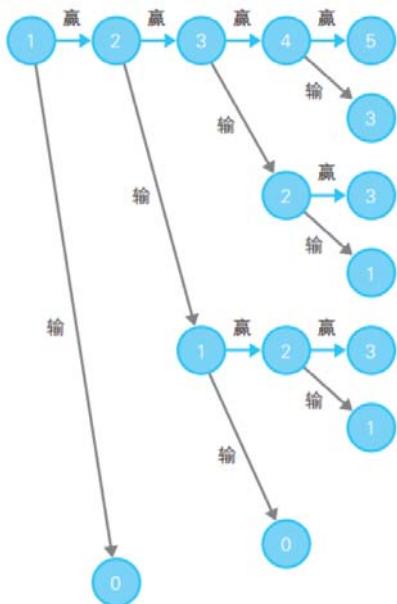
$$= 2 (429 + 132 + 84 + 70) = 1430$$

23. Q23: 二十一点通吃

23.1 问题描述

赌场经典的二十一点游戏中，每回合下注 1 枚硬币，赢了可以得到 2 枚硬币 (+1 枚)，输了硬币会被收走 (-1 枚)。

假设最开始只拥有 1 枚硬币，并且每回合下注 1 枚，那么 4 回合后还能剩余硬币（即没有输光）的硬币枚数变化情况如图所示，共有 6 种（圆形中间的数字代表硬币枚数）。



求最开始拥有 10 枚硬币时，持续 24 回合后硬币还能剩余的硬币枚数变化情况共有多少种？

23.2 解法 1—暴力搜索

每次投注有赢和输两种情况，则 24 次投注有总共 2^{24} 种组合情况。注意，由于输光即游戏终止（即不准透支），因此输赢的顺序是有关系的。有些组合虽然从 24 把投注的总成绩来看是有硬币剩余的，但是在前面部分已经出现了输超过赢导致游戏提前终止了。

算法流程如下：

遍历所有可能的输赢组合，**for each of one:**

检验是否所有从头开始到中间任意步投注后剩余硬币都大于 0：

如果是，则计数加 1

代码参见：point21game1()

由于没有把中间可以提前退出的情况考虑进去，本算法存在比较大的运算效率浪费。

23.3 解法 2—深度优先路径搜索

本问题可以按照深度优先路径搜索的思路来解决。这样可以有效地解决“解法 1”的问题，即中间碰到已经输光的情况时，就不必沿着当前路径继续向下探索了。本系列有很多类似的题目，比如说，Q14, Q18，基本上可以用相同的框架来解决，这里不再做过多说明。

代码参见：point21game2()

意外的是，运行时间相比解法 1 并没有质的提升。不过从最终结果来看，在 $\{10, 24\}$ 的条件下，总的可能路径数接近 2^{24} ，或者说本题的合法路径解是比较稠密的，因此 DFS 策略能带来的效率提升也就有限了。

23.4 解法 3—动态规划

进一步，本题还可以用动态规划的策略来解决。

考虑 $\{\text{steps}=k, \text{coin}=c\}$ 条件下的总可能路径数记为 $f(k, c)$ ，当前下注的结果有两种可能，赢了则硬币数变为 $c+1$ （相应地步数 k 减一），输了则硬币数变为 $c-1$ （相应地步数 k 减一），因此可以得到以下递推关系式：

$$f(k, c) = f(k - 1, c + 1) + f(k - 1, c - 1)$$

考虑本游戏的规则，当硬币变为 0 就表示失败了；另一方面，在硬币变为 0 之前步数用完了，就表示赢了。因此可以得到以上递推关系式的初始或边界条件：

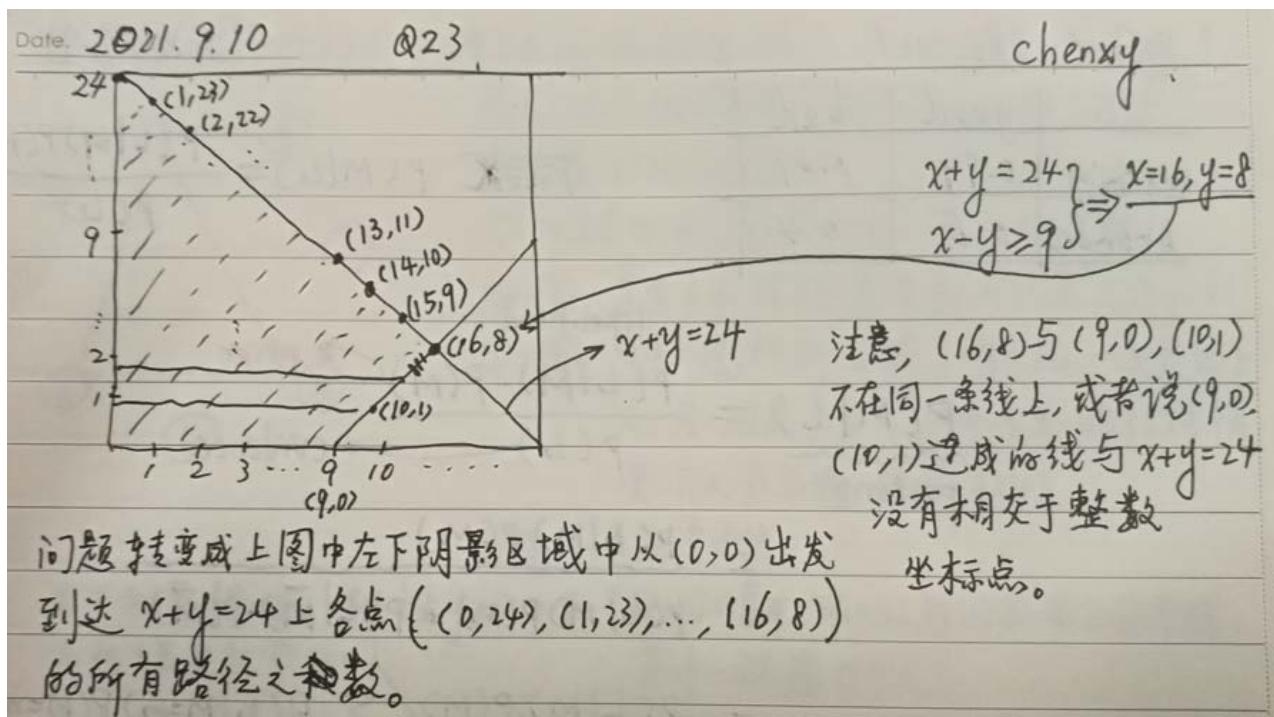
$$\begin{aligned} f(*, c) &= 0 \\ f(0, c > 0) &= 1 \end{aligned}$$

代码参见：point21game3()

注意，代码中要先判断硬币是不是为 0，因为即便走完了最后一步硬币数才变为 0 也算输了。

23.5 后记

本题还可以转换为以下问题。考虑从 $(0,0)$ 出发，只能往右（对应于输）或向上（对应于赢），考虑总步数 24 的前提下，限于在图中阴影区域中移动到达反斜对角线上各点 $\{(0,24), (1,23), \dots, (16,8)\}$ 的总的可能路径数。



24. Q24 :完美的三振出局

24.1 问题描述

对喜爱棒球的少年而言，“三振出局”是一定要试一次的。这是一个在本垒上放置 9 个靶子，击打投手投来的球，命中靶子的游戏。据说这可以磨练球手的控制力。

现在来思考一下这 9 个靶子的击打顺序。假设除了高亮的 5 号靶子以外，如果 1 个靶子有相邻的靶子，则可以一次性把这 2 个靶子都击落。譬如，如图所示，假设 1 号、6 号、9 号已经被击落了，那么接下来，对于 “2 和 3” “4 和 7” “7 和 8” 这 3 组靶子，我们就可以分别一次性击落了。

求：9 个靶子的击落顺序有多少种？这里假设每次投手投球后，击球手都可以击中靶子。

24.2 解题分析

第一感就是深度优先路径搜索问题。因为本系列已经出现来太多这样的问题，比如 Q23, Q18, Q14, just to name a few.

24.2.1 DFS 算法流程

由于已经击落的靶子就不能再用了，所以靶盘状态用尚未被击落的靶子的列表表示即可。

算法流程如下(python-style):

```
Memo = dict()

Def explore(unStriked):
    If unStriked is empty:
        Return 1

    If unStriked is in memo:
        Return memo[unStriked]

    Count = 0

    遍历所有可能的靶盘的 next state: nextUnStriked, for each of one:
        Count = count + explore(nextUnStriked)
```

24.2.2 遍历下一个状态

深度优先搜索的同一类问题都可以直接套用上一节所示的框架套路，除了一些 problem-specific 的细节需要针对个别问题具体处理。本问题的要点在于如何给出下一个状态的遍历列表来。

本题解采用(比较笨拙的)办法如下：

首先，定义一个查找表，包括所有可能的 double-strike (一次击落两块) 的可能组合。

其次，搜索当前的 unStriked 的 2-combination's，并查找各自是否出现在 double-strike 列表中，如果在的话则表明这是一种可能的 double-strike 结果，由此导致一种 next-state

最后，所有 unStriked 的中靶子都可能被单独击落，也分别导致一种 next-state

代码参见 findNext(unStriked)

24.3 后记

本题说明其实是有一定模糊性(因为我一开始恰好掉到坑里去了)的。在关于一次能击落两块的情况下，题意其实是说当除了 5 以外，任意两块连着的靶子，击中其中任意一块时可能只掉下被击落的那一块，也可能把相邻的那一块带下去(但是两边都有相邻靶子的话，也只会带下一块)。这对应着游戏情况应该是球正好几种两个靶子的交界处附近，对于实际玩过棒球或者观看过类似电视节目的人来说可能是理所当然的，但是作为一道抽象的算法题目来说则很难说是理所当然的。

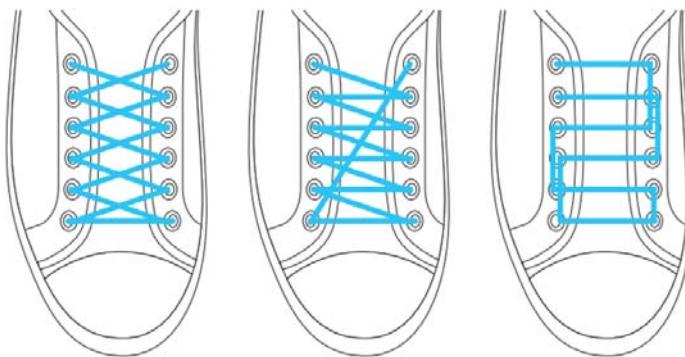
既然说到这儿，这道题目可以稍微做一下条件修改：任意两块连着的靶子，击中其中任意一块时一定会连同相邻的那一块一起掉下去(两边都有相邻靶子的话，也只会带下其中某一块)。在这种条件下，9 个靶子的击落顺序有多少种呢？

嗯，这道延伸题目算不算有点原创的意思？^-^

25. Q25: 鞋带的时髦系法

25.1 问题描述

即便系得很紧，鞋带有时候还是会松掉。运动鞋的鞋带有很多时髦的系法。下面看看这些系法里，鞋带是如何穿过一个又一个鞋带孔的。如下图所示的这几种依次穿过 12 个鞋带孔的系法很有名（这里不考虑鞋带穿过鞋带孔时是自外而内还是自内而外）。



这里指定鞋带最终打结固定的位置如上图中的前两种系法所示，即固定在最上方（靠近脚腕）的鞋带孔上，并交错使用左右的鞋带孔。

问题：鞋带交叉点最多时的交叉点个数。譬如上图左侧的系法是 5 个，正中间的系法是 9 个。

25.2 解题分析

深度优先路径遍历问题。

本系列中已经出现过很多类似问题，比如 Q24, Q23, Q18, Q14, just to name a few. 此类问题都可以直接套用类似的框架套路，除了一些 problem-specific 的细节需要针对个别问题具体处理，比如说，如何进行状态表示，如何给出下一个状态的遍历列表，等等。

本文在路径遍历的基础上进一步要求找出各种系法（即各种路径）中的交叉点数，而不仅仅是给出可能的路径数，这就要求（1）在路径遍历的同时要记住每条路径的历史；（2）对每条路径中的交叉点进行数数。

由于要记住每条路径的历史，所以 memoization 技巧不能用了（待确认）。

25.2.1 状态表示方法

考虑用 `left` 和 `right` 两个列表来表示当前状态，分别表示左右两列剩下的鞋带孔。由于鞋带孔不能重复穿过，基于这种表示方法，在深度搜索过程中也免去了“是否已经访问过”的判断。

除此之外，还需要：

(1) 指明下次应该轮到穿哪一列。activeCol: 0—left column; 1—right column

(2) 上一次穿过的是哪个孔: lastPole

必须从左列第一个孔开始(从右列开始结果一样, 假定这种对称的系法算同一种系法), 而且最后一个必须是右列第一个孔, 因此初始状态为:

`left = [1,2,3,4,5], right=[1,2,3,4,5], activeCol=1, lastPole=0`

右端最后一个孔是确定性的, 因此在深度优先搜索中不进行处理(因此以上 right 中不包括 0), 但是在搜索到终点进行交叉点计数时要把最后穿过 right-pole-0 的 edge 加入 pathHist (参加下一节流程) .

25.2.2 DFS 算法流程

算法流程如下(`python-style`):

Def explore(left: List, right: List, activeCol:int, lastPole:int, pathHist:List):

If left and right are both empty:

Add the last edge (from left to right-pole-0) to pathHist

判断统计交叉点个数

Remove the last edge from pathHist

Return 1

Count = 0

遍历所有可能的下一次穿孔位置, for each of one (要区分从左到右和从右到左):

Add the current edge to pathHist

Count = count + explore(···)

Remove the current edge to pathHist

Return count

25.2.3 遍历下一个状态

在问题中即寻找可以下一个穿过的鞋带孔。这里需要分从左到右和从右到左的处理, activeCol 即为此目的而设。activeCol 为 1 表示接下来是要从左到右; activeCol 为 0 表示接下来是要从右到左。另外, 还需要知道上一次穿过的孔(即 lastPole), 这样才能决定从哪个孔到哪个孔。

25.2.4 交叉判断

简单地来说, 每个 edge 由两个数(x,y)决定, x 是左边鞋带孔位置, y 是右边鞋带孔位置。两个 edges

的 x 坐标的相对大小和 y 坐标的相对大小关系是反的的话，则表示交叉。参见 `isCross()`。

25.2.5 运行结果

14400

Number of cross = 45, tCost = 0.300(sec)

`[[0, 5], [1, 5], [1, 4], [2, 4], [2, 3], [3, 3], [3, 2], [4, 2], [4, 1], [5, 1], [5, 0]]`

第一个数字是可能系法总数，最后一个结果是具有最大交叉点的系法的鞋带孔穿越顺序。

25.3 后记

其实，总的可能系法总数可以很简单地直接计算出来，即 $((N - 1)!)^2$ 。理由就留给小伙伴们自己思考了。本题中因为要给具体的路径历史（即鞋带孔穿越顺序），所以必须做穷尽的搜索。但是能直接计算出以上总的可能路径总数，对于程序调试是有帮助的。

26. Q26：高效的立体停车场

26.1 问题描述

最近，一些公寓等建筑也都配备了立体停车场。立体停车场可以充分利用窄小的土地，通过上下左右移动来停车、出库，从而尽可能多地停车。

现在有一个立体停车场，车出库时是把车往没有车的位置移动，从而把某台车移动到出库位置。假设要把左上角的车移动到右下角，试找出路径最短时的操作步数。

举个例子，在 3×2 的停车场用如图 13 所示的方式移动时，需要移动 13 步。

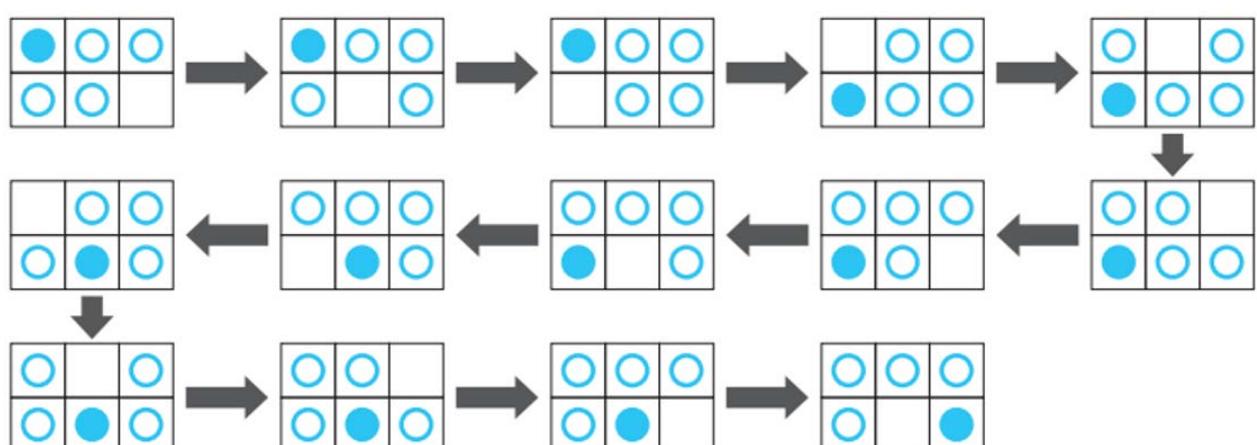


图 1 车从左上角移动到右下角的示例 1 (13 步)

不过，如果用如下图所示的移动方法，则只需要移动 9 步。

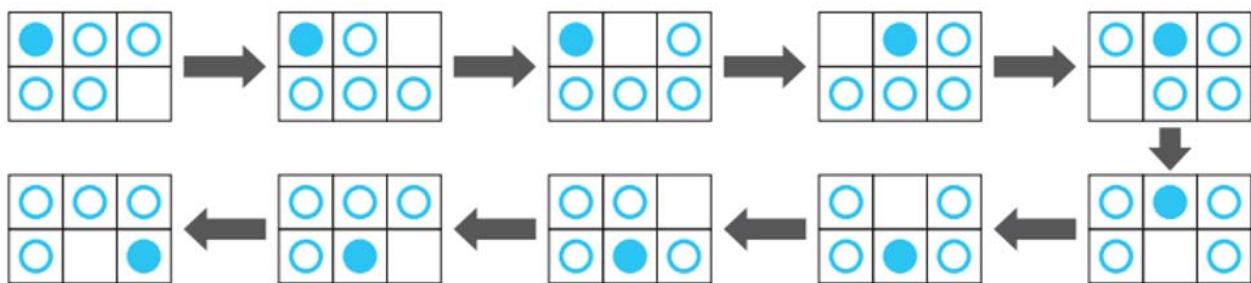


图 2 车从左上角移动到右下角的示例 1 (9 步)

问题：

求在 10×10 的停车场中，把车从左上角移动到右下角时按最短路径移动时需要的最少步数。

26.2 解题分析

因为是求最短路径，所以可以用广度优先搜索策略来解决。

广度优先搜索有与深度优先搜索相同的三个基本要素：

- (1) 基本算法流程
- (2) 状态表示方法
- (3) 遍历下一个状态

其中(2)和(3)相关联的，合理的状态表示和遍历方法的选择是解决问题的效率的关键。

26.2.1 BFS 算法流程

算法流程如下(python-style)：

初始化：

创建 Queue, visited 对象

Add start node to Queue, together with its layer (for start node, it should be zero)

```
Add start node to visited

While (Queue is not empty):

    node = Queue.pop()

    if node is the target:

        return the corresponding layer

    Traverse all the neighbour nodes, for each of them:

        If neighbourNode is not in visited:

            Add neighbourNode to visited:

            Queue.push(neighbourNode)
```

26.2.2 状态表示方法

本题考虑以车和空位的位置坐标表示当前状态（或节点），再加上对应的层数，即 $[[x_1, y_1, x_2, y_2], \text{layer}]$ 。其中 $[x_1, y_1]$ 表示车的位置， $[x_2, y_2]$ 表示空位的位置。

车的其实位置为 $[0,0]$ ，空位的其实坐标为 $[N-1, M-1]$ ， N 和 M 分别为车库的行数和列数。

当车的位置移动到 $[N-1, M-1]$ ，即算到达目标。

26.2.3 遍历下一个状态

本问题搜索下一个状态（或者说移动方式）是一件麻烦事。

焦点是空位的移动。因为空位是一定移动的，车则不一定，车只有在与空位相邻的时候才有机会移动。要点如下：

- (1) 空位在中心位置的话，有 4 个方向都可以移动
- (2) 空位在边缘的话就会受到限制
- (3) 如果车恰好在空位某个方向的相邻位置，则空位往这个方向的移动是交换车和空位的位置

参见 `findNext()` 函数。本题解中 `findNext()` 函数写得比较冗长，暂时没有想到更简洁的写法。不过这个不是关键点，暂时将就一下。

27. Q27 :禁止右转—深度优先搜索

27.1 问题描述

本题来自《程序员的算法趣题》中的第 27 题。

在某些国家（如日本、英国）车辆是靠左通行的，开车左转比右转要舒服些。那么现在来想一下，在网格状的道路网络中，如何只靠直行或者左转到达目的地。

问题：在规整的矩形网格状道路网络中（如图 1），从左下角出发到右上角，对于任意的正整数组合 (m, n) ，总共有多少种可能的路线？条件是只能直行或左转（禁止右转），且已经通过的道路不能重复，但是允许道路有交叉。并且假定只能指定的矩形范围以内行驶。

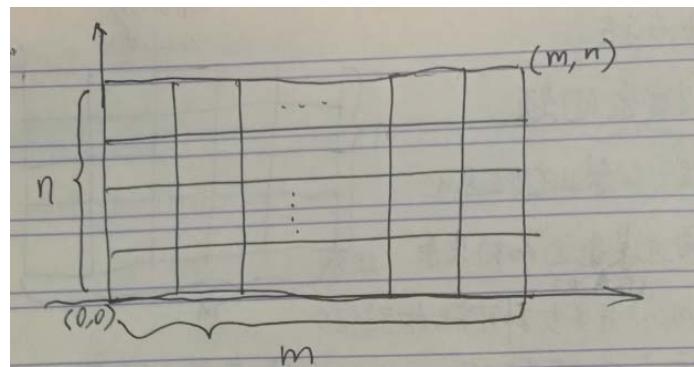


图 1 圈定范围内的矩形网格状道路网络

图中加上了直角坐标系，并将左下角标记为 $(0,0)$ ，右上角标记为 (m, n) ，并无必然性。仅为解题对照方便，且并不是一般性。

27.2 分析

先从一些简单的情况开始分析以获得一些直观认识。针对简单情况得到的分析结果也为程序调试提供了简单的 testcase。

显然，从出发点开始，第一次动作只能是向右走（注意，不可以右转，但是可以右行）。而且出发点

$\{m=1, n=1\}$: 显然只有一条路线。{右、上}

$\{m=2, n=2\}$: 有两条路线。{右、上、上、左、下、右、右、上}; {右、右、上、上}

$\{m=3, n=2\}$: 有四条路线

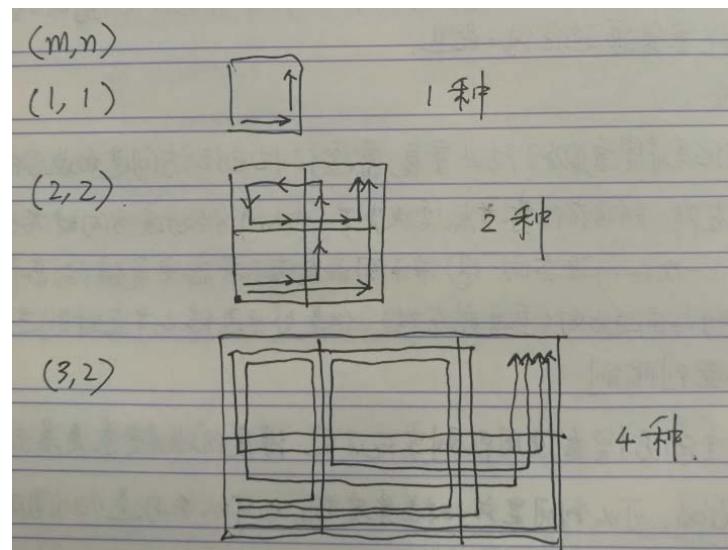


图 2 几种简单情况

注意，题目并不要求最短路径，所以允许绕弯兜圈，只要不重复通过同一条边即可。

这个问题可以用深度优先搜索（DFS）方法来解决。题目并不是要求找出一条特定的合法路径，而是要找出所有可能的路径数，所以不管有没有找到合法路径，都要进行回溯继续找下一条，直到穷尽完所有的可能性。

要点：

- (1) 在深度优先搜索中，需要记忆已经访问过的边以防止重复访问
- (2) 由于禁止右转，所以在每一个节点处选择可能的行进方向时，需要知道上一次的行进方向
- (3) 在中间的节点处，若不考虑不能的重复的限制的话有两种可行的前进方向（直进或左转），但是处于边界的节点处，选择就会受到限制

与广度优先搜索使用队列进行实现不同，深度优先搜索需要基于栈来实现。可以基于显示的栈来实现，也可以基于递归调用来隐式地实现栈处理。采用递归调用的实现方法，代码会显得非常紧凑，其代价则是执行效率的下降。

27.3 Solution#1—递归式实现

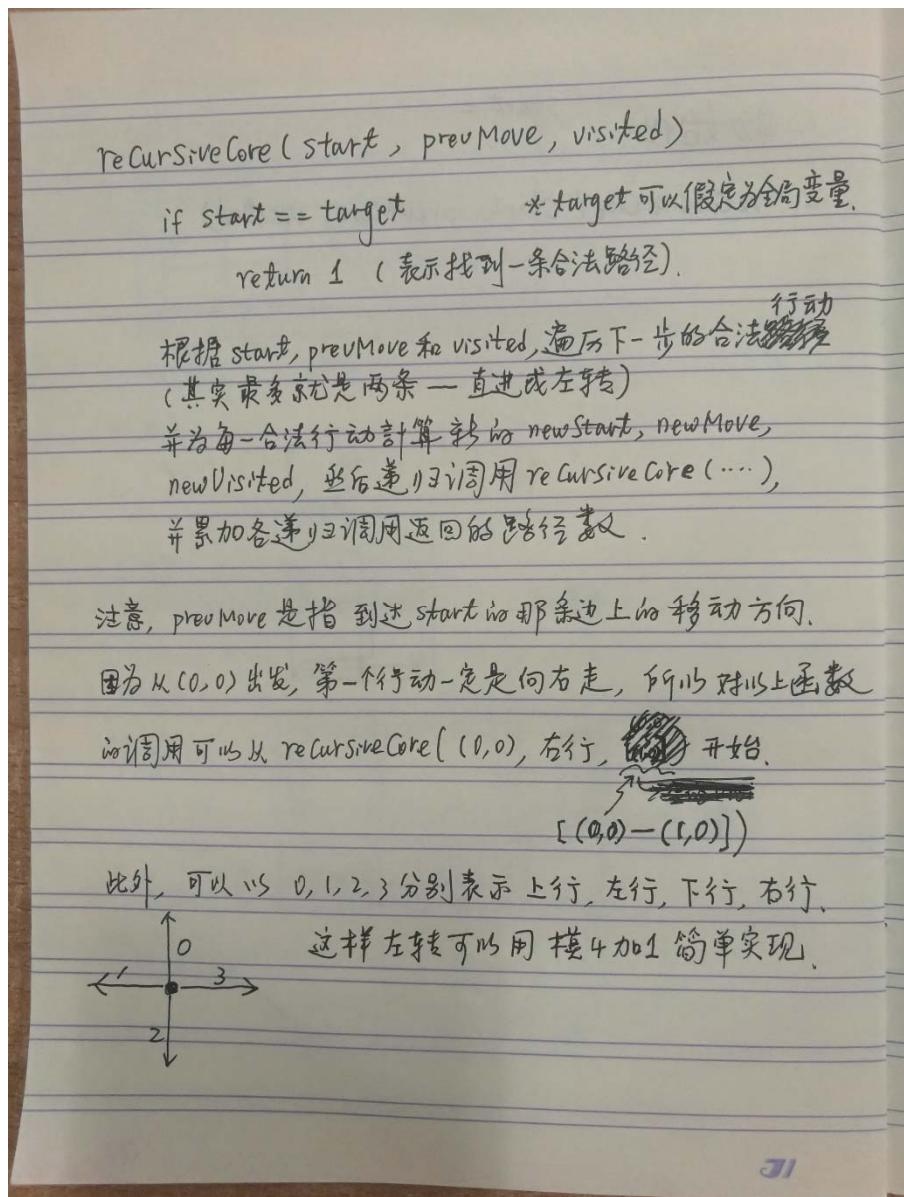


图 3 递归实现基本思路

27.4 Solution#2—非递归式实现

28. Q28:社团活动的最优分配方案

28.1 问题描述

某新建学校的校长, 要为学生规划要为他们准备哪些社团活动。学校里有 150 名想要运动的学生, 共有 10 种社团, 每个社团所需要的活动场地面积和人数如下所示:

社团	所需场地面积 (m^2)	人数
棒球	11000	40

足球	8000	30
排球	400	24
篮球	800	20
网球	900	14
田径	1800	16
手球	1000	15
橄榄球	7000	40
乒乓球	100	10
羽毛球	300	12

在总人数上限为 150 人的条件下，选择成立哪些社团，能够使得所需场地面积最大。

当然本问题的求解目标有点反直觉。一般的优化目标难道不应该是在场地面积的约束条件下最大化能够容纳的人数嘛。。。不过这个无所谓。

28.2 解题分析

每个社团要么选择、要么不选择，在某个约束条件下，达成另外一个指标的最大化，这是一个经典的 0/1 背包问题(0/1-knapsack problem, or knapsack problem w/o repetition)。

经典的背包问题的叙述中有以下几个要素，以及本题中各要素的映射关系如下所示：

要素	标准的 0/1 背包问题		本问题
Items	可选的物件	↔	可选的社团
Capacity	总容积或者总重量	↔	总人数上限
Cost	各物件的体积或者重量	↔	各社团的人数
	Value(各物件的价值)	↔	各社团所需场地面积
最大化目标	所选物件总价值	↔	所选社团所需场地面积总和

解决背包问题的标准框架当然是动态规划 (Dynamic Programming)。动态规划的根本要点是一个大的问题可以分解为性质相同但是“规模”较小的子问题，也即原问题具有良好的子问题分解结构。这种问题的最基本的解决方案是递归方法，但是递归调用的代价比较大，而动态规划则可以看作是解决递归问题的一种优化方法。

- What Is Dynamic Programming? Simply put, dynamic programming is an **optimization method for recursive algorithms**, most of which are used to solve computing or mathematical problems. You can also call it an algorithmic technique for solving an optimization problem by breaking it into simpler sub-problems.

不管用递归的方式求解，还是用动态规划求解，都涉及以下几个关键步骤：

- (1) 找出子问题
- (2) 列出原问题与子问题的递推关系式(recurrence equation)
- (3) 找出 baseline cases 并求解, as the initial or boundary condition for solving the above-mentioned recurrence equation

令 areas 表示各项目所需场地面积数组, humans 表示各项目的人数数组, maxHuman 表示总人数限制, 数组下标对应以上项目表中各项目的序号(counting from 1 – 为了叙述的方便取从 1 开始计数)。

以 $f(I,J)$ 表示在候选社团为前 I 个社团 $\{1,2,\dots,I\}$, 最大允许人数为 J 的条件下的解。则本问题的子问题可以理解{可选社团项更少 and/or 最大允许人数更小}的求解问题。如下我们可以求得本问题解的递推关系式 (分考虑选择 or 不选择最后一项社团两种情况进行考虑):

- (1) 如果不选择最后一项(I)社团 $\rightarrow f(I-1,J)$
- (2) 如果选择最后一项(I)社团 $\rightarrow areas[I] + f(I-1,J-humans[I])$

而本题要求解最大面积，所以要在以上两种选择中取较大的那个作为正确选择，即：

$$f(I,J) = \max(f(I-1,J), areas[I] + f(I-1,J-humans[I]))$$

当然，以上还漏了一点，即当社团 I 的人数已经超过总人数限制 J 的话，事实上就只有(1)可以选择，因此以上递推关系式需修正为：

```

if humans[I] ≤ J
    f(I,J) = max(f(I-1,J), areas[I] + f(I-1,J-humans[I]))
else
    f(I,J) = f(I-1,J)

```

接下来，需要考虑初始化 (或者说边界条件)。很显然，当最大允许人数等于 0，或者可选社团项集合为空时，所得的最大面积为 0。即：

$$\begin{aligned} f(0,*) &= 0 \\ f(*,0) &= 0 \end{aligned}$$

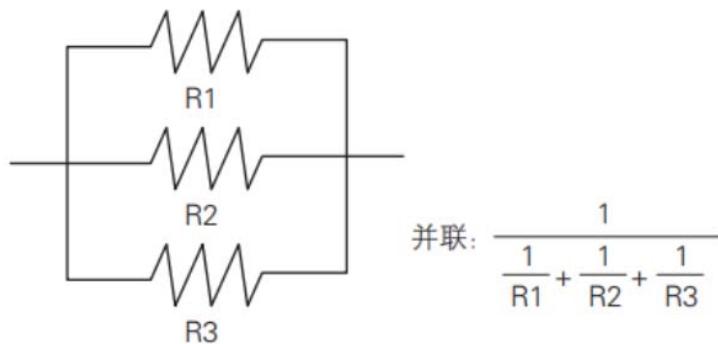
28.3 递归实现

本问题求解复杂度较低，采用递归方式的实现加上 memoization 可以在几乎可以忽略的时间中求解。

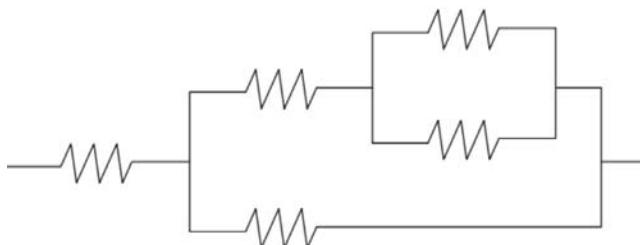
29. Q29: 合成电阻的黄金分割比

29.1 问题描述

我们在物理课上都学过“电阻”，通过把电阻串联或者并联可以使电阻值变大或者变小。电阻值分别为 R_1 、 R_2 、 R_3 的 3 个电阻串联后，合成电阻的值为 $R_1 + R_2 + R_3$ 。同样 3 个电阻并联时，合成电阻的值则为“倒数之和的倒数”。如下图所示：



现在假设有 n 个电阻值为 1Ω 的电阻。组合这些电阻，使总电阻值接近黄金分割比 $1.6180339887\dots$ 。举个例子，当 $n = 5$ 时，如果下图这样组合，则可以使电阻值为 1.6。



问题：求 $n=10$ 时，在所有可能的电阻网络中，所得到的电阻中最接近黄金分割比的数值，精确到小数点后 10 位。

29.2 解题分析

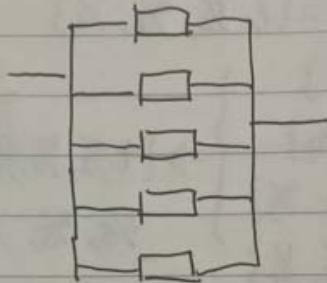
29.2.1 分割成两组

本题解的最关键的 insight 是对于任何电阻网络，总可以把它分割成两个组，两个组内部的拓扑结构任意，两个组之间或串联或并联。然后，每个组又可以同样继续分割下去，因此本题可以用动态规划的策略来解决。详细分析过程如下。

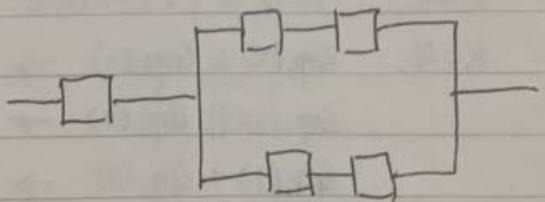
Q29.

Date: 2021.9.14.

Insight: N 个电阻不管以何种方式进行连接，总可以分割为非空的两组， Grp1 和 Grp2 . 两组之间或串联或并联。

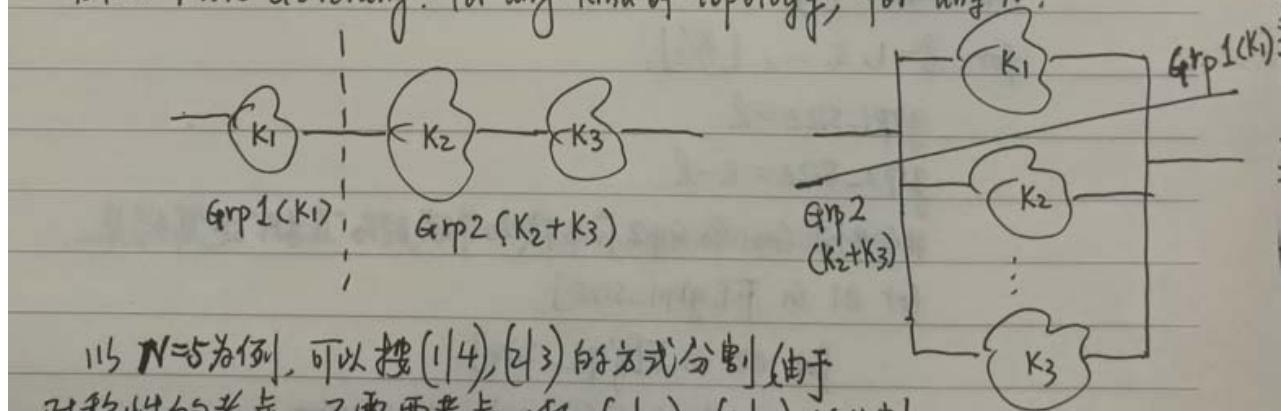
例1 $N=5$.可以看作是 $\text{Grp1}(1) \parallel \text{Grp2}(4)$ 也可以看作是 $\text{Grp1}(2) \parallel \text{Grp2}(3)$

.....

可以看作是 $\text{Grp1}(1) + \text{Grp2}(4)$

* $\text{Grp}x(x)$: 括号内部的 x 表示该组中共包含几个电阻，但是不管它组内是什么拓扑结构。

例2: More Generally. For any kind of topology, for any N .



以 $N=5$ 为例，可以按 $(1|4)$, $(2|3)$ 的方式分割 (由于

29.2.2 N=5 时的分割例

以 $N=5$ 为例，可以按 $(1|4)$, $(2|3)$ 的方式分割。由于对称性的考虑，不需要考虑 $(4|1)$, $(3|2)$ 的分割。

那么 $N=5$ 的情况有多少种可能的取值呢？

Case 1: $\text{Grp}_1(1) + \text{Grp}_2(4)$ For Case 1, 由 $\text{Grp}_1(1)$ 的各种可能的取值与 $\text{Grp}_2(4)$ 的各种取值的两两组合（即两个集合的直积，也叫克罗内克积）的和值。

Case 2: $\text{Grp}_1(1) \parallel \text{Grp}_2(4)$

Case 3: $\text{Grp}_1(2) + \text{Grp}_2(3)$

Case 4: $\text{Grp}_1(2) \parallel \text{Grp}_2(3)$ 对于 Case 2, Case 3, Case 4, 同理。

* 这里用 “+” 表示两组之间串联，
用 “||” 表示两组之间并联。
由此可以得到本问题的动态规划
求解方案。 deli 得力

29.2.3 可能的阻值计算例

以下为到 $N=4$ 为止的笔算计算例。

Date: 2021.9.14. Q29.

令 $F(k)$ 表示 k 个电阻构成的网络的可能取值的集合。

$K=1$, 显然 $F(1) = \{1\}$

$K=2$ $F(2) = \{2, 1/2\}$

$K=3$, $F(3) = \{1+2, 1+1/2, 1||2, 1||1/2\} = \{3, 3/2, 2/3, 1/3\}$

$K=4$, $\begin{aligned} \text{Grp}_1(2) + \text{Grp}_2(2) &\rightarrow \{4, 5/2, 1\} \\ \text{Grp}_1(2) \parallel \text{Grp}_2(2) &\rightarrow \{1, 2/5, 1/4\} \\ \text{Grp}_1(1) + \text{Grp}_2(3) &\rightarrow \{4, 5/2, 5/3, 4/3\} \\ \text{Grp}_1(1) \parallel \text{Grp}_2(3) &\rightarrow \{3/4, 3/5, 2/5, 1/4\} \end{aligned} \quad \left. \right\} \Rightarrow \{4, 5/2, 5/3, 4/3, 1, 3/4, 3/5, 2/5, 1/4\}$

用集合表示，编程语言的内建集合会自动完成去重。

29.2.4 算法实现流程

实现流程如下：

```

初始化: F[1] = {1}
for K=2, 3, ..., N,
    valueSet = {}
    for l=1, 2, ..., ⌊K/2⌋,
        grp1_size = l
        grp2_size = k - l
        # 分别求 grp1 和 grp2 在串联和并联时的“直积”运算结果。
        for e1 in F[grp1_size],
            for e2 in F[grp2_size],
                Add (e1+e2) to ValueSet
                Add (e1||e2) to ValueSet
    F[K] = valueSet
遍历 F[N] 求其中与黄金分割比值最接近的值。

```

29.3 代码及测试

运行结果：

```

golden_ratio=1.6180339887, valueWithMinDiff=1.6181818182,
tCost= 0.003(sec)
Totally there are 3158 kinds value for N = 10 element circuit network

```

由以上结果还可以看出，在N=10时总共有3158种可能的电阻值。那可能的拓扑结构数至少不小于这个数。因为有些不同的拓扑结构的电阻值可能是相同的。

29.4 后记

29.4.1 分割的洞见

这道题是到目前为止看完题目后觉得最没有头绪的题目。觉得无从下手。原书中给的提示没看懂要说啥，即便做完了这道题目再回头看依然没有看懂要说啥。“偷看”了几个网友的解答，也没有看出什么头绪（只上代码很难说有什么参考意义，这种问题直接读代码很难读懂）。在<https://blog.csdn.net/lanying100/article/details/116000347>读到“分割成两组”的思路，一开始也没有想明白，最后终于想明白了，觉得这个洞见确实妙极！想通了这一点后面就水到渠成了。最后，希望本文的解说能够让人更容易理解一些。

29.4.2 另一种思路

一开始我按照另外一种思路走，即把串联和并联看作是两种运算符，再加上括号，这样任何一种电路网络都可以表示为一个由若干个 1 为操作数，包含以上两种运算符以及括号的“算术”表达式。遍历所有可能的“算术”表达式然后评估表达式的值即可。

但是在“遍历所有可能的“算术表达式”这点上卡壳了。由于括号的位置拜访的自由度很高，而且还存在深度嵌套的情况，对于这种非常“非结构性”的东西要遍历没有想到什么好办法，暂时只好放弃了。但是觉得这个把问题转变为算术表达式求值的思路还是不错的。

30. Q30：插线板连接方式

30.1 问题描述

本题来自《程序员的算法趣题》中的第 30 题。

假设有双插口和三插口两种插线板。墙壁上只有一个插座能使用。而需要用电的电器有 N 台，试考虑如何分配插线板。举个例子，当 N=4 时，共有 4 种插线板连接方式（使用同一个插线板，不考虑插口位置的差异，只考虑插线板的连接方法）。另外，要是所有插线板上最后没有多余的插口。N=4 时的连接方式如下图所示。

求 N=20 时，插线板的插线连接方式有多少种（不考虑电源功率限制的问题）？

30.2 递归表达式

令 $f(N)$ 表示要用电的电器为 N 台时的插线连线方式数。

很显然， $f(1) = 1$ ，此时直接将设备插在墙壁的插孔上就可以了。

当 $N \geq 2$ 时，必须使用插线板来扩容。

第 1 级插线板有两种情况，即双口插线板或者三口插线板。

Case1: 考虑第1级插线板用双口插线板，假设双口插线板的两个插口下的子网络分别提供 k_1 和 k_2 个有效插口数（插线板自身也要消耗插口数，不算在有效插口数内）。则 k_1 和 k_2 必须满足以下条件：

$$\begin{aligned} k_1 &\geq 1 \\ k_2 &\geq k_1 \\ k_1 + k_2 &= N \end{aligned}$$

第一个条件源自于“不能有多余的插口”这一限制条件。进一步，由于“使用同一个插线板，不考虑插口位置的差异，只考虑插线板的连接方法”，即{插口1的子网络提供 k_1 个有效插口数，插口2的子网络提供 k_2 个有效插口数}的连接方法与{插口1的子网络提供 k_2 个有效插口数，插口2的子网络提供 k_1 个有效插口数}视为相同的方案，不重复计算。因此可以不失一般性地假定 $k_2 \geq k_1$ 。

由此可以得到递归表达式：

$$f_2(N) = \sum_{k_1=1}^{\lfloor N/2 \rfloor} f(k_1) * f(N - k_2)$$

Case2：考虑第1级插线板用三口插线板，假设三口插线板的三个插口下的子网络分别提供 k_1 、 k_2 和 k_3 个有效插口数（插线板自身也要消耗插口数，不算在有效插口数内）。则 k_1 、 k_2 和 k_3 必须满足以下条件：

$$\begin{aligned} k_1 &\geq 1 \\ k_3 &\geq k_2 \geq k_1 \\ k_1 + k_2 + k_3 &= N \end{aligned}$$

同理可得递归表达式：

$$f_3(N) = \sum_{k_1=1}^{\lfloor N/3 \rfloor} \sum_{k_2=k_1}^{\lfloor (N-k_1)/2 \rfloor} f(k_1) * f(k_2) * f(N - k_2 - k_3)$$

最后，总的递归表达式则为： $f(N) = f_2(N) + f_3(N)$

30.3去重(repetition removal)

但是按照上一节的递归表达式编程计算得到结果(74801991)与原书给出的结果(63877261)不一致！原因在于以上递归计算中漏掉了一些重复的情况。比如说，当一个双口插线板的两个插口分配了相同的负载（即 $k_1=k_2$ ）时，这个时候存在一些对称的配置，根据题设要求不能重复计算，需要排除掉。以下分双口插线板和三口插线板两种情况来分别考虑如何去重。

30.3.1 双口插线板的去重

30.3.2 三口插线板的去重

31. Q31:计算最短路径

31.1 问题描述

本题来自《程序员的算法趣题》中的第 31 题。

假设有正方形被划分为若干个边长为 1cm 的正方形方格。请思考沿着正方形方格的边从左下角到右上角再回到左下角 (*1) 往返的情况。这里往返程不能经过同一条边（但是允许往返路径有交叉点）。

求大的正方形的不同边长 N (cm) 时，总共有多少种最短路径？

*1 与原题略有不同—但是没有本质区别，只是作者习惯了迎合平面坐标系第一象限来思考

N=1 时，很显然有两条路径；N=2 时，有 10 条路径。如下图所示。

插入图片

31.2 解题分析

31.2.1 BFS or DFS?

第一感应该能够想到这是一个跟图遍历搜索相关的问题。“最短路径”容易让人联想到 BFS(广度优先搜索)，然而本题并不是 BFS 的菜。本题的重点在于要找出所有的最短路径（而最短路径的长度本身其实是确定性的，只要保证去程只向上或向右，回程只向下或向左，就能保证是最短路径），所以这道题可以用深度优先搜索算法来解决。

31.2.2 节点的表示

既然要作为图搜索问题来解决，那第一个问题就是这个隐含的图（implicit graph）的节点表示什么？题目要求往返程的边是不能重复的，但是交叉点是可以重复的，所以这里考虑用正方形方格的边来对应“图的节点”。每条边以嵌套的 tuple--((x0,y0), (x1,y1), direction) 来表示。 $(x0,y0)$ 和 $(x1,y1)$ 分别表示通过这条边时的起点和终点，这是一个有向表示。`direction` 用于指示通过当前这条边时是去程还是在回程—不是必须的，根据 $(x0,y0)$ 和 $(x1,y1)$ 也可以判断出这条边的通过方向。

31.2.3 如何避免往返程通过同一条边

上面说了“边”是以有向的方式表示的，但是题目的要求是往返程不能通过相同的边，不论方向。所以在通过对称以确定一条边是否被访问过时需要去除掉方向信息，只要两个端点相同即为相同的边。在以下代码中的实现方式是在将“边”加入 visited 时，是成对地加入，以存储的代价换取查询 visited 的便利。

但是 visited 的管理与通常的递归式深度优先搜索中的处理又有所不同。在以下代码实现中，visited 是作为参数传入 dfs()，并且在每次退出时将进入 dfs() 函数时添加的节点又弹出去了--To be frankly, 初始代码在这里掉坑里了。。。等彻底想清楚如何解释再来追加解释。

31.3 运行结果

运行结果如下：

```
N = 2, numSlns = 10, tCost = 0.0(sec)
N = 3, numSlns = 80, tCost = 0.0(sec)
N = 4, numSlns = 786, tCost = 0.007980823516845703(sec)
N = 5, numSlns = 8592, tCost = 0.10571455955505371(sec)
N = 6, numSlns = 100360, tCost = 1.3892908096313477(sec)
N = 7, numSlns = 1227200, tCost = 19.797149658203125(sec)
```

大概 N 增加 1，运行时间增加 15 倍的样子。要想搜索 N 更大时的情况，需要进一步优化实现，或者甚至从根本上调整实现策略。

32. Q32: 榻榻米的铺法

32.1 问题描述

考虑叫做“仪式铺法”的榻榻米铺法，这种铺法可以使相邻榻榻米之间的接缝不会形成十字，象征着吉祥。举个例子，如果一个房间看作由纵 3*横 4 个正方形方格构成，铺满这个房间需要 6 张榻榻米（榻榻米的大小是统一的 1*2 的长方形），则铺法如下所示（还有与这两个成上下或左右对称的图案此处略去）：

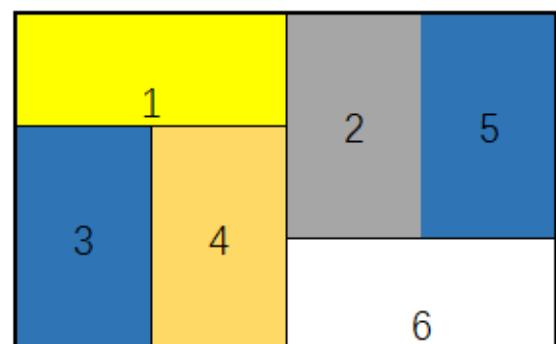
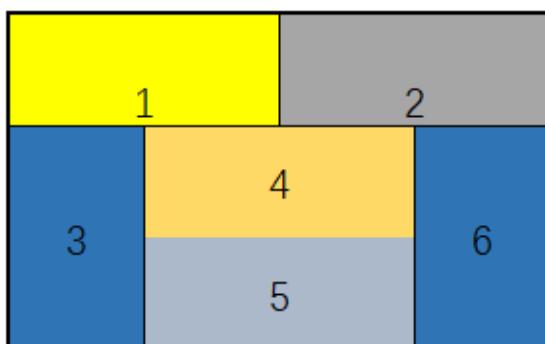


图 1 (3,4)的房间的两种铺法

求用 14 张榻榻米铺满大小为 4*7 的房间的铺法。

32.2 解题分析

深度优先路径遍历问题。

这一类问题的难点主要体现在（具体问题的形式千差万别）如何将问题转换为易于遍历搜索的问题表述形式，或者如何找到有效的遍历方式（包括节点状态表示，以及如何确定邻节点或者说子节点）。

32.2.1 如何判断能否铺

经过分析可以发现，构成非法铺法的一个关键特征是在某一个点的周围 4 个块均属于 4 块不同的榻榻米。如下图所示两个都是 NG 的铺法。



图 2 非法的铺法例

那如何把这种约束条件转化为容易判别的形式（换句话说人眼识别以上 NG 图案很容易，如何让计算机能做出判断）呢？比如说，如下图的状态，现在要判断带问号“？”格子是否可以铺下一张榻榻米（无论是纵还是横？）



图 3 “?”格子能够铺?

如前所述，非法的铺法中是指在某个交点周围的 4 个格子分属于不同的榻榻米。所以在判断“？”处是否能铺，作为充要条件（这里先不考虑触及边界的情况）就是判断“左上、上、左”这个三个格子是否不全部属于不同的榻榻米。换言之，只要“左上、上”属于同一榻榻米、**或者**“上、左”属于同一榻榻米即表明当前格子可以铺新的榻榻米（“左上、左”肯定分属不同的榻榻米不必判断）。反之，如果“左上、上”属于不同榻榻米、**且**“上、左”属于不同榻榻米则表明当前格子肯定不能铺。据此可以判断，上图中，左边不能铺，右边可以铺。

这也就自然地引出了给到当前状态为止已经铺好的榻榻米进行编号的考虑，以方便进行以上判断。

32.2.2 状态表示和遍历

【有点复杂的想法】

以整个房间的铺设状态作为图的节点，可以表示为 $H \times W$ 的矩阵（ H 为纵向高度， W 为横向宽度），矩阵元素取值 {0: 尚未覆盖；非零 IDX: 已经序号 IDX 的榻榻米被覆盖}。起始状态（即深度优先搜索树的根节点）为全 0 状态。终止状态为全非零状态。这样这个问题就转变成在一张图中搜索从某个节点（全‘0’状态）出发，到达另一个节点（全非零状态）的所有路径的问题。

然后，邻节点搜索可以这样考虑，遍历与移近覆盖的区域相邻（有共同的边）的格子，检查从各邻接格子是否能够铺设（因为是从左上往右下搜索的方式，针对每个格子只需要考虑向右铺或向下铺）。这种方法的好处是递归深度会小一些，但是已覆盖区域的邻接格子的查询似乎比较麻烦。

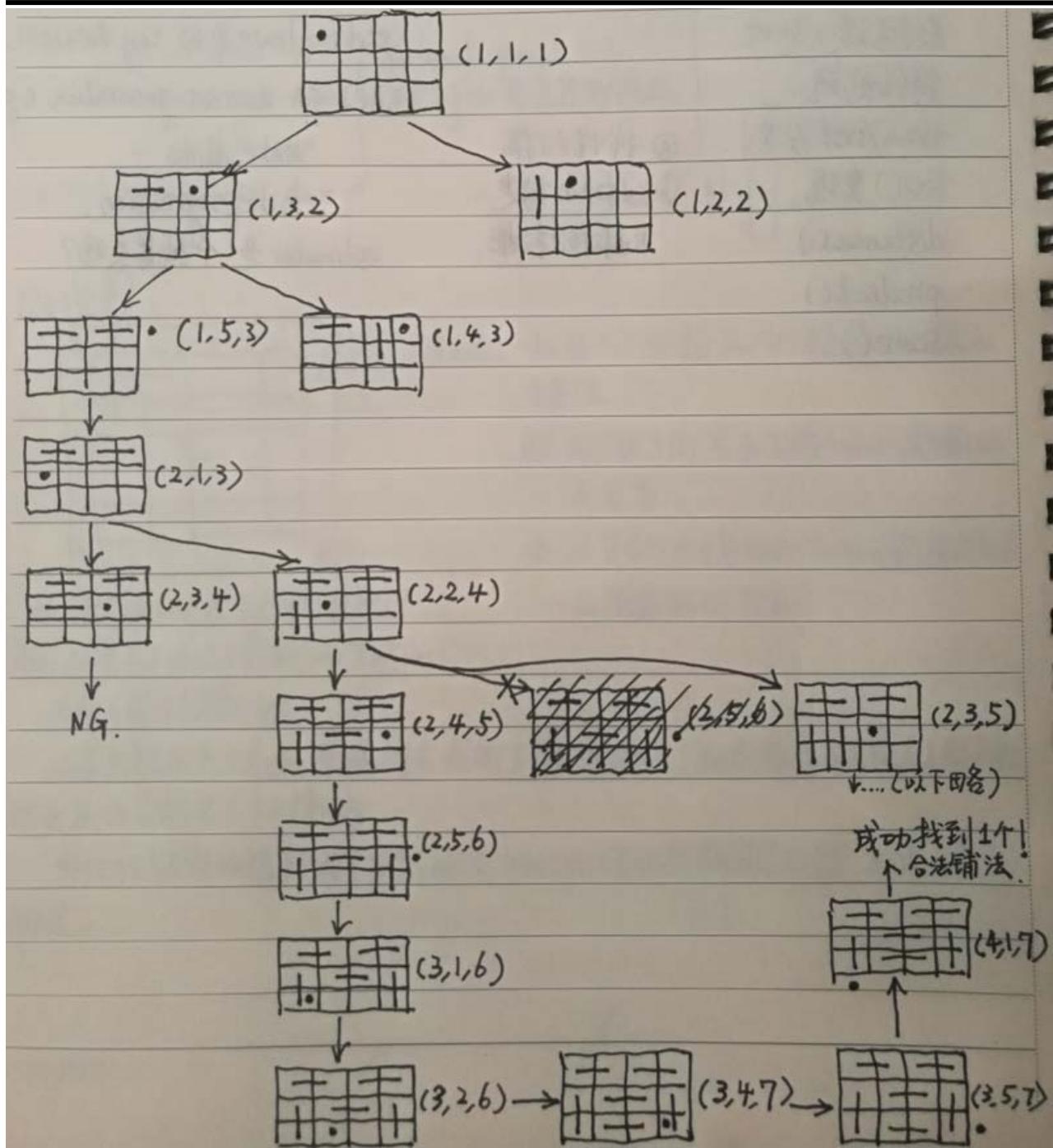
【逐行扫描】

原书中给出的解法（这道题原书给出的 Javascript 代码比 ruby 容易理解些好歹看懂了）很巧妙（当然要自己先撞墙了然后再读懂了才能体会其中妙处）。

简单地逐行（当然逐列也可以）逐格扫描过去，碰到边界就切换到下一行，碰到已经被覆盖的就跳过去，针对既非越界也没有被覆盖的格子结合当前已经铺就的状态进行是否能铺（参见上一节的描述）的判断。加上“围栏”技巧（参见下一节）以及给榻榻米编号的技巧，得到了一个非常简洁的实现。

这一解法的缺点可能是递归深度相对来说会比较大，当问题规模比较大的时候容易触及递归深度限制（有待确认）

以下为($H=3,W=4$)的递归搜索示意图（没有画出围栏，可以把格点坐标按照 matlab-style 的从 1 开始的方式理解；黑点表示当前探索的格子，旁边(x,y,idx)对应的递归调用的参数）



32.2.3 围栏

如原书中所述，对于棋盘类问题，很多时候，在问题界定的范围外侧加上围栏就可以简化边界条件的判定。在本题解中，在 $H \times W$ 格子矩阵外围追加一圈格子，并全部赋值为“-1”。在代码中虽然没有显式地(explicitly)用到“-1”的判断，但是在“if room[h, w+1]==0:”和“if room[h+1, w]==0:”判断中隐式地(implicitly)利用到了。因为追加了外围的一层，所以不必担心(w+1)和(h+1)越界而导致矩阵访问出错；另外，由于初始化为“-1”，所以在上述判断不会被误判为有效的可填的格子。

33. Q33: 飞车与角行的棋步

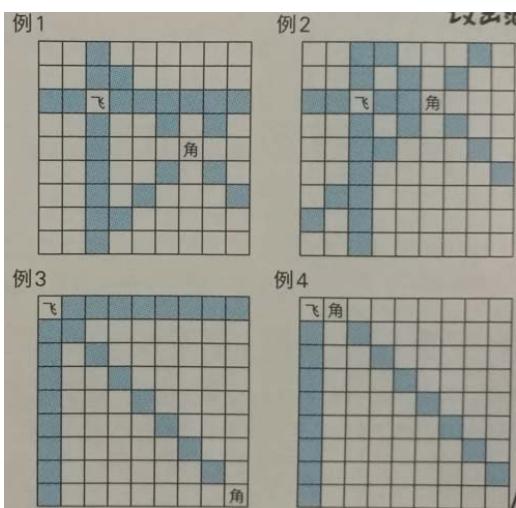
33.1 问题描述

将棋棋盘纵横各 9 格 ($9 \times 9 = 81$ 个格子)，假设在将棋棋盘上的任意两格内分别放入飞车和角行这两颗棋子。两颗棋子不能放在同一格，假设棋盘上没有其它棋子。

问题：将所有可能的棋子摆放位置都考虑在内，求两颗棋子的棋步范围内所有格子数之和。

注 1：所谓棋步范围，说成是“攻击范围”更容易理解吧。

注 2：飞车攻击范围为与自己同一行和同一列的所有格子，但是不能越过其它棋子。类似于中国象棋中的“车”。角行的攻击范围为位于与自己位置所谓的正反 45 度角对角线上的所有各点，但是不能越过其它棋子。在如下例子图中，例 1, 2, 3, 4 的攻击范围内的格子数总和分别为 24, 23, 23, 15。注意，在本题中，攻击范围不包含自身，也不包含另一颗棋子（假定都为同方棋子）



33.2 解题分析

这道题目感觉相当直白啊。

简单的暴力搜索就好，遍历{飞, 角}的所有各种可能的位置组合，因为棋盘为 $9 \times 9 = 81$ 个格子，所以总的位置组合数为 $81 \times 80 = 6480$ 中。

针对每种位置组合，统计两个棋子的攻击范围内的格子总数。这个算是本题的焦点。有以下几个注意点：

- (1) 在扫描各棋子的攻击范围时，从棋子自身出发分别有 4 个方向。对于飞车来说是{up, down, left, right}，对于角行来说是{right-up, right-down, left-up, right-down}

本题解中，就是傻傻地分别针对飞车和角行的各自 4 个方向进行查询。查询碰到对方棋子或者边界时就停止

- (2) 扫描各个方向时，碰到对方棋子时即停止（如以上例图 2、4 所示）
- (3) 有些格点处于双方共同攻击范围，需要注意不要重复计数

在本题解中，采用将格点置 1 的方式表明格点处于攻击范围，最后统计 2 维数组中的 1 的个数。这样就自然地解决了重复计数的问题。因为两次对同一格格点置 1 不会导致被重复计数。

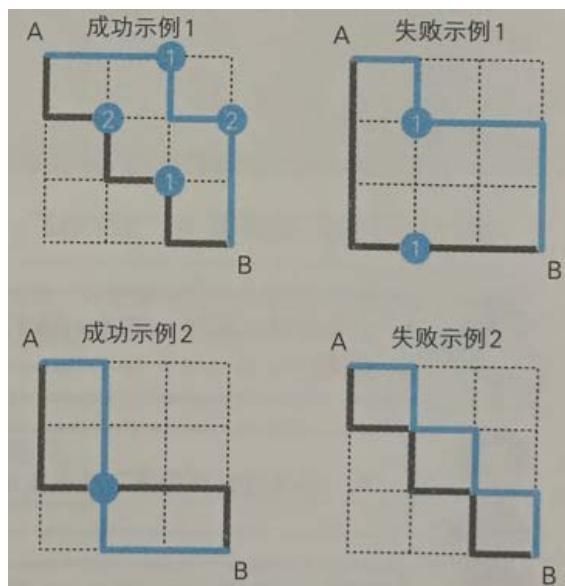
边界处理。对于棋盘类问题，很多时候，在问题界定的范围外侧加上围栏就可以简化边界条件的判定（原书语）。以下本题解在将棋盘设为 11×11 大小，并将四周格点值置为 2 即为此意。

34. Q34: 命中注定的相遇

34.1 问题描述

假设存在如下图的正方形，该正方形被划分成了边长为 1 的正方形的格。男生从 A 到 B，女生从 B 到 A，分别沿着最短路径以相同速度前行（两人同步，每次都横向或纵向各走一格）。如果符合以下情形，则判定为“命中注定的相遇”

- (a) 两次同时停在同一直线内的定点上（相互可见状态）
- (b) 在同一顶点交汇（相互接触状态）



边长为 3 时，几种成功和失败的例子如上图所示。

问题：边长为 6 时，发生“命中注定的相遇”的情况共有多少种？

34.2 解题分析

两人都是以最短距离行进，因此男生只有{向右，向下}两种选项，女生只有{向左，向上}两种选项，两人的组合动作因此就是四种：

- (1) {男:右；女:左}；
- (2) {男:下；女:左}；

(3) {男:右; 女:上};

(4) {男:下; 女:上};

每前进一步，进行是否符合条件的判断，如果是符合条件(a)，则直接判断成功；如果是符号条件(b)，将相互可见次数加1，如果相互可见次数到达2则判定成功；如果女生已经跑到男生的左上位置而还尚未满足条件的话，则可以判定失败；最后（作为边界条件），如果发生越界的话也判定失败。

这种类型的问题很适合于用递归的方式实现（唯一需要担心的是递归深度），算法流程如下所示：

递归调用函数 $f(\text{manX}, \text{manY}, \text{womanX}, \text{womanY}, \text{meet})$

前4格参数分别表示目前男女坐标，meet 表示到目前为止相互可见次数

如果跨越了边界，表示失败了，返回

如果女生到达了男生左上的位置，表示失败了，返回

如果两者坐标重合，表示成功：

$\text{total_count} = \text{total_count} + 1$ # total_count 定义为全局变量以简化实现

返回

如果两者的某个坐标相同，表示相互可见：

$\text{meet} = \text{meet} + 1$

if $\text{meet} == 2$:

$\text{total_count} = \text{total_count} + 1$ # total_count 定义为全局变量以简化实现

返回

针对4种可能的组合动作进行递归调用

$f(\text{manX}+1, \text{manY}, \text{womanX}-1, \text{womanY}, \text{meet})$ # 男:右; 女:左

$f(\text{manX}+1, \text{manY}, \text{womanX}, \text{womanY}-1, \text{meet})$ # 男:右; 女:上

$f(\text{manX}, \text{manY}+1, \text{womanX}-1, \text{womanY}, \text{meet})$ # 男:下; 女:左

$f(\text{manX}, \text{manY}+1, \text{womanX}, \text{womanY}-1, \text{meet})$ # 男:下; 女:上

男生和女生的初始坐标分别为(0,0),(N,M), meet 初始值当然是0，因此以 $f(0,0,N,M,0)$ 开始搜索即可（这里区分开矩形的长与宽，是考虑拓宽应对范围）

34.3 后记

这道题目算是最近几道问题感觉最为轻松的，难得地很快就想清楚了解决方案。

然而，**令人尴尬的是**，答案错了。更令人尴尬的是，“偷看”了原书答案以及CSDN上其他小伙伴们贴出来的答案，基本思路是一致的，运行别人的代码也的确给出了正确答案，可是反复对比看仍然没有看出自己的代码究竟错在哪儿了—**自信心受到了一万点暴击**。

但是，与“看到一个正确的答案后：喔，原来如此”相比，也许看一个错误的答案并揪出其中的八阿哥（BUG）更具有挑战性，也更有助益。所以，这里还是厚着脸皮贴出来供大家批判，并期待有人帮助我指出哪儿错了^-^.

34.4 Bug-Fix

34.4.1 分析

如上所述，在满足相遇的两种条件之一后即提前退出会导致大量的路径（满足相遇条件后的路径分岔）被漏算了。正确的做法是只要不出界，就必须一直走到底。由此可以看出在之前的错误的解答中的以下 4 个 return 的条件（如下图所示）中，第 2 个和第 4 个其实都是错的，都是不需要的。

第 4 个是错的，这个原因已经解释了。

其中第 2 个是错误的原因还需要再追加一点解释。第 2 个 return 是基于这样的考虑：(在错解中认为)一旦找到满足条件的就应该提前退出，而如果两者都已经擦肩而过（满足了“**manX > womanX and manY > womanY**”之后就不可能再相遇了）且还没有满足条件的话就不必继续搜索下去了。而现在要求不管是否已经满足条件，都必须每条路径走到底，这个 return 自然就不再需要了。因为任何一对路径不管是否满足相遇的条件下，最后一段都必然是满足“**manX > womanX and manY > womanY**”条件的。

```
if manX > N or manY > M or womanX < 0 or womanY < 0:  
    return # Cross the boundary  
if manX > womanX and manY > womanY:  
    return # Missed forever  
  
if manX == womanX and manY == womanY:  
    total_count += 1  
    # print(total_count)  
    return  
if manX == womanX or manY == womanY:  
    meet += 1  
    if meet == 2:  
        total_count += 1  
        # print(total_count)  
        return
```

进一步如以下代码所示统计条件和退出条件做了一些调整优化。

首先 `meet` 递增改为针对 X 坐标和 Y 坐标分别统计。X 和 Y 坐标同时相同（即题设中的相互解除状态）无非就是统计成加 2 而已（反正现在不比提前退出）。这样做的好处是统一了两种相遇的情况的统计。

此外，最终判断是否满足相遇条件是用 (`meet>=2`)，有人可能会有疑惑会有 `meet>2` 的情况吗？是的。因为统计是每走一步都统计的，考虑两者在某一点相遇后两者继续朝相反方向走，那两人的某一个坐标必定继续保持一致，因此会导致 `meet` 一直增加直到到达边界不得不转向为止。

34.4.2 后记 2

在很多的路径遍历搜索问题中的确是一旦找到目标（满足要求）就停止继续搜索下去的。前面的错解恰恰是掉在了这样的惯性思维所造成的陷阱中。

反过来说，完全通用的解题策略是不存在的。每一个实际问题都有它的一些独特的地方需要仔细分析 `case-by-case` 地处理。

35. Q35: 0 和 7 的回文数

35.1 问题描述

已知对任意正整数 n 而言，一定存在 n 的正整数倍的“仅由 0 和 7 构成的数”。

例) $n = 2$ 时， $2 \times 35 = 70$

$n = 3$ 时， $3 \times 2359 = 7077$

$n = 4$ 时， $4 \times 175 = 700$

$n = 5$ 时， $5 \times 14 = 70$

$n = 6$ 时， $6 \times 1295 = 7770$

这里我们思考一下 n 的正整数倍的“仅由 0 和 7 构成的数”中的最小值，且该最小值为回文数的情况（回文数在 Q01 中提过，即反过来读也是同一个数）。举个例子， $13 \times 539 = 7007$ 里的 7007 就是回文数。

问题：求位于 1~50 的所有满足以上条件的 n 。

这道题目是典型的“披着羊皮的狼”，一看似乎很简单，然而陷阱重重的那种。

35.2 解法 1—暴力搜索

第一感是，暴力搜索搞一搞就可以了吧。反正我的习惯也是从最傻（naive）的方法着手。。。

算法流程如下：

遍历 1~50 之间的整数，for each n from [1,50]:

遍历 m from 1 to infinity:

$k = m * n$

判断 k 是否只由“0”和“7”构成（或者是否只由“7”构成），如果是的话：

进一步判断它是不是回文数，如果是的话加入结果列表

以上流程之所以成立是因为如上所述“已知对于任意整数 n，一定存在 n 的正整数倍的仅由 0 和 7 构成的数”，否则的话，以上流程就有可能陷入无限循环。

实际上仅由 0 和 7 构成的数再加上要构成回文数，则必然首尾都是 7，因此可以排除掉偶数以及 5 的倍数，这样可以将搜索范围缩小一半。

以上流程虽然简单易懂，然而写出代码一运行，就会让你怀疑人生。如果没有以上一定存在的保证，你会怀疑它是不是陷入了死循环；知道了有一定存在的保证，你就只能怀疑程序是不是写错了。

```
In [25]: runfile('F:/AlgorithmPuzzles/Q35-0-and-7-palindrome.py',  
Number of ok_dict = 3, tCost = 790.8717994999997  
13 (539, 7007)  
39 (1813, 70707)  
49 (143, 7007)
```

单单针对 9 这个数字，在我的机器上跑了 80 多秒才跑出来。

仔细思考一下会发现，在以上正向循环中，针对 m 的遍历中，所得到的 $k=m*n$ 绝大多数根本就不满足后面的“仅由 0 和 7 构成”以及“构成回文数”的条件，换句话说，绝大多数的遍历计算都是无效的。如果换一个方向，先筛选能满足“仅由 0 和 7 构成”以及“构成回文数”的条件的数再来判断是否能被 n 整除的话，就可以避免绝大多数无效的搜索了。

35.3 解法 2—逆向思考

逆向思考的关键是先按照不同的位数，(1) 构成满足“仅由 0 和 7 构成（或仅由 7 构成）”的条件的

数; (2) 用这些数去试能不能被待检查对象 n(1~50 之间奇数且不能被 5 整除)整除; (3) 如果能的话, 再判断是否回文数, 如果是的话, 则这个 n 满足条件是答案之一; 否则, 这个 n 不满足条件应被排除。

针对以上流程遍历位数 m 即可。直到 1~50 间的数要么被判断满足条件要么不满足条件全部判断完毕就结束。

需要注意的一个坑是: 题目要求的是 n 的倍数中第一个“仅由 0 和 7 构成(或仅由 7 构成)”的条件的数、同时又是回文数。n 的倍数中可能有很多个满足“仅由 0 和 7 构成(或仅由 7 构成)”的条件的数, 而且其中也可能有多个还满足回文数的条件。但是只有第一个(即最小的那个)满足“仅由 0 和 7 构成(或仅由 7 构成)”的条件的数同时又是回文数的 n 才符合题目的要求。**这就是为什么是以上(1)—(2)—(3)的顺序, 而不是(1)—(3)—(2)了。**有兴趣的读者可以细想一下为什么**(1)—(3)—(2)不行。**

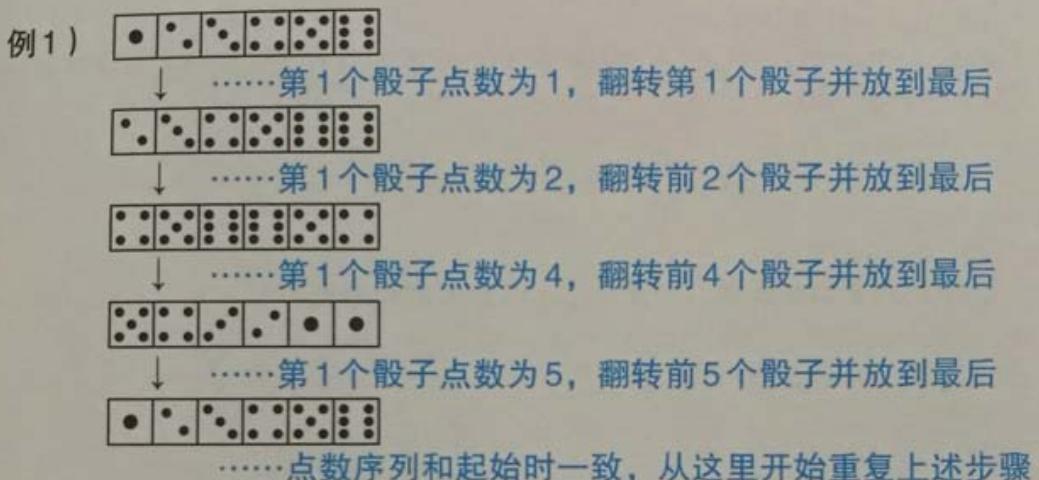
其中如果需要“仅由 0 和 7 构成”或“仅由 7 构成”之前进行切换的话, 只需要切换以下两条语句即可。

```
if item[0]=='0' or ('0' not in item):  
# if item[0]=='0':  
    ...
```

36. Q36: 翻转骰子

36.1 问题描述

这里有 6 个骰子排成一排，当第 1 个骰子的点数为 n 时，翻转前 n 个骰子并放到最后（假设翻转前后的点数之和为 7。也就是说，1 点翻转后为 6 点，2 点翻转后为 5 点，3 点翻转后为 4 点）。如果重复这个过程，就会出现同样的点数序列循环的情况。



例 2) $343434 \rightarrow 434434 \rightarrow 343433 \rightarrow 433434 \rightarrow 343443 \rightarrow \dots$ 点数序列和起始时一致，从这里开始重复上述步骤
 $443434 \rightarrow 343343 \rightarrow 343434$

例 3) $132564 \rightarrow 325646 \rightarrow 646452 \rightarrow 131325 \rightarrow 313256 \rightarrow \dots$ 点数序列和第 3 步时一致，从这里开始重复第 3 步及以后的步骤
 $256464 \rightarrow 646452$

例 4) $616161 \rightarrow 161616 \rightarrow 616166 \rightarrow 161611 \rightarrow 616116 \rightarrow \dots$ 数字序列和第 2 步时一致，从这里开始重复第 2 步及以后的步骤
 $161661 \rightarrow 616616 \rightarrow 161161 \rightarrow 611616 \rightarrow 166161 \rightarrow \dots$
 $661616 \rightarrow 116161 \rightarrow 161616$

可以注意到，像例 3 和例 4 这样，有些点数序列不会进入循环（132564、325646、616161 等）。

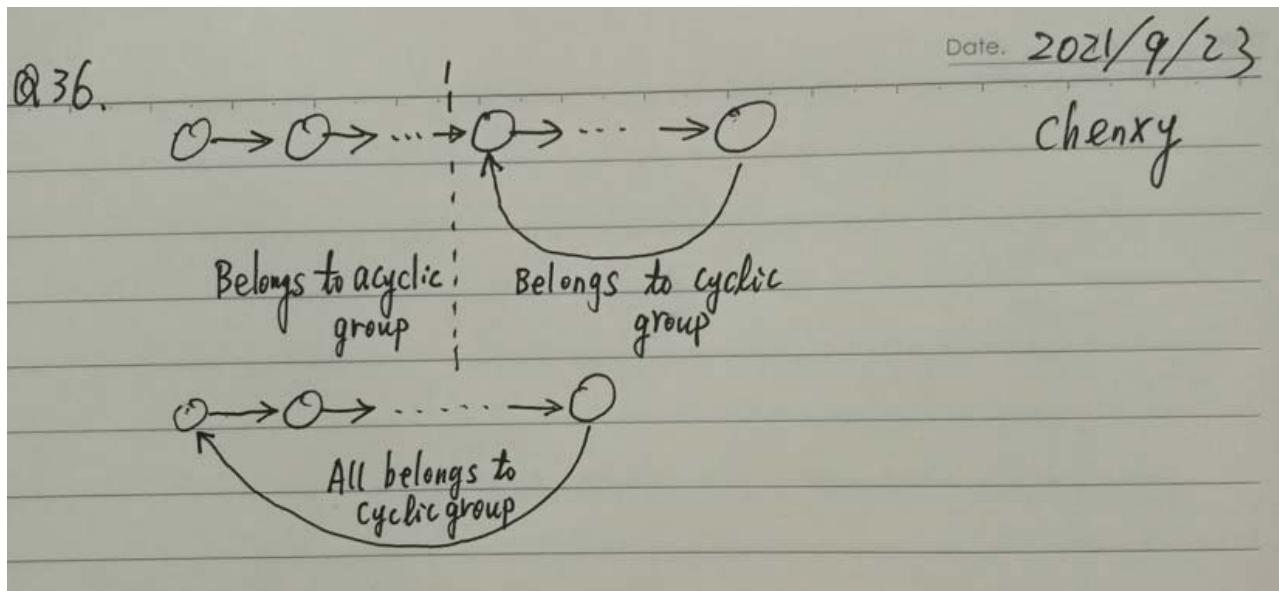
问题

求像上面这样未进入循环的点数序列的个数。

36.2 解题分析

6个骰子构成的排列构成的总的状态数为6的6次方即 $6^6=46656$ 种。

从任意的状态出发经过翻转骰子的操作更新状态，其历经的状态序列都必然是如下图所示：



其中上边代表一般性情况，下边可以看作是一种特殊情况。

为什么必然会成为上图这种情况呢？原因在于这是一个有限状态序列（如前所述最多只有46656种状态）。从任意状态出发，最多经过46656步以后，必然会回到前面已经经过的某个状态（根据组合计数中的“鸽笼原则”）。这个洞见（Insight）是解决本题的关键。

用acyclic存储已经判明的属于非循环的状态（即从它自己出发不会回到自身），用cyclic存储已经判明的属于循环的状态（即从它自己出发会回到自身）。

从任意状态出发进行翻转更新，将历经的状态序列存储在statest中：

途中（包括一开始它自身）不管是碰到已判明是acyclic的还是cyclic的状态，都可以判断从出发状态到此刻为止的状态都是属于acyclic（为什么？请大家仔细品一品），将它们加入到acyclic

如果途中某个状态已经存在于statest（即上图所示情况），则如上图所示这表明在状态序列中到该状态之前的所有的状态都属于acyclic，其后（包括它自己）都属于cyclic，将这些状态分别加入acyclic和cyclic

对所有46656种状态进行遍历（分别以它们为起始状态）重复以上过程即可。

Acyclic和cyclic存储已经判明的结果，是为了避免重复搜索。毕竟，比如说，一旦你从某个状态出发已经找到了一个loop，那这个loop中的所有的状态都属于循环状态了，自然不必再针对其中每个状态重新搜索一次。

37. Q37: 翻转 7 段码

37.1 问题描述

计算器、运动计时器等所用的“7段显示屏”(7-segment display)是使用如图8所示的7个部分的亮灭来显示1个数字的(这里有A~G这7个比特，对应比特为1时亮灯，为0时灭灯)(表3、图8)。

表3 用于显示各个数字的比特的序列

n	A	B	C	D	E	F	G
0	1	1	1	1	1	1	0
1	0	1	1	0	0	0	0
2	1	1	0	1	1	0	1
3	1	1	1	1	0	0	1
4	0	1	1	0	0	1	1
5	1	0	1	1	0	1	1
6	1	0	1	1	1	1	1
7	1	1	1	0	0	0	0
8	1	1	1	1	1	1	1
9	1	1	1	1	0	1	1

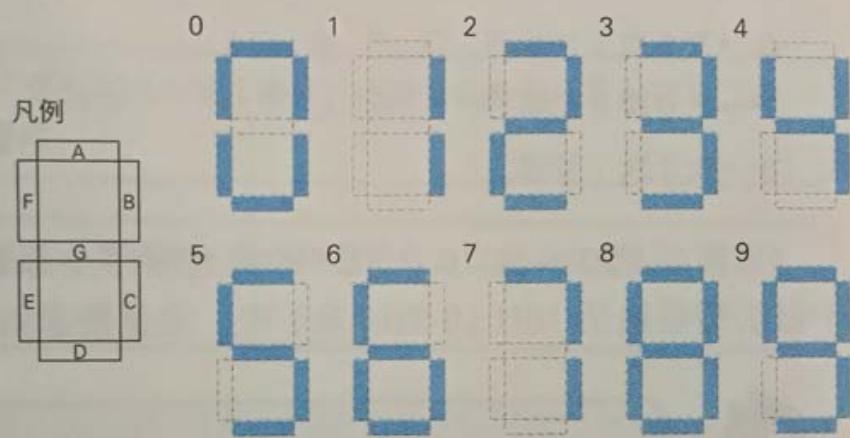


图8 7段显示屏的亮灯示例

现在假设我们要使用这样的显示屏分别依次显示0~9这10个数字。显示当前数字时，如果应亮灯部分与显示上个数字时相同，则依然保持亮灯；同样地，如果应灭灯部分相同，也依然保持灭灯，也就是说，这里是通过只切换有变化的部分的灯的亮灭来显示下一个数字的。

问题：求把10个数字全部显示出来时，亮灯/灭灯的切换次数最少的显示顺序，以及这个切换次数。

注：原题从答案来看是不包括第一个数字显示时从全黑到显示该数字所需要的亮灭切换次数的。不过这个不影响解题，但是可能会影响答案。

37.2 解题分析 1—暴力搜索

从暴力搜索(原书中使用“全量搜索”这个词很贴切)着手。

10个数字的不同排列顺序共有 $10!=\text{factorial}(10)=3628800$ 种，针对每一种排列顺序进行切换次数的统计即可。

本题解中用numpy array存储各数字显示所需要的灯亮灭表示矩阵，每行表示一个数字，表示使用7

段码对应的二进制表示。

切换次数的计算可以理解为两个二进制序列之间的汉明距离，即两个二进制序列之间的不同的数的个数。汉明距离可以用异或操作实现。

此外，各数字的二进制表示之间的汉明距离总共只有 100 个（连自己与自己之间的距离也算上，虽然不用。考虑到查找实现的便利， (k,j) 和 (j,k) 也分开来算，虽然它们是相等的）可以预先计算好，这样可以避免在遍历搜索时的大量的重复计算。

37.3 解题分析 2

先对本文题进行子问题分解并求其递推关系式。

假定尚未点亮（访问）的数字列表为 `unvisited`，上一次点亮（访问）的数字为 `prev`，到目前为止累计的切换数为 `toggleSum`。

记从这个状态开始到将 `unvisited` 中所有数字显示完所需要的最小切换数为 $f(\text{unvisited}, \text{prev}, \text{toggleSum})$ 。

接下来，可以从 `unvisited` 中任选一个数字作为下一次显示，而最小切换数则为这多条路径中切换数最小的那一条。由此可以得到递推关系式如下所示(u : `unvisted`; p : `prev`; s : `toggleSum`):

$$f(u, p, s) = \min_i f(u[0:i, i+1:end], u[i], s + \text{dist}[p, u[i]])$$

这一递推关系式似乎难以用动态规划的方式来实现（凭感觉而已，还不知道如何解释清楚，留待后面回头再看），这里先用递归的方式解决。

37.4 后记

运行时间有点长，需要考虑进一步优化。

这个问题其实可以看作是，具有 10 个节点的全连接无向图，每条边的权重值代表两个节点所代表的数字的 7 段码显示的二进制表示之间的汉明距离。因此本题就转化为就该图中任意一条最长无重复路径的权重总和。这其实就是著名（恶名昭彰）的旅行商问题。这是一个 NP 问题，但是只有 10 个节点还可以对付。接下来考虑用递归+Memoization 的方法来试一试会不会变得更快一些。

38. Q38: 填充白色

38.1 问题描述

把 4×4 的方格分别涂成黑色和白色。对任意方格，当选中一个方格的时候对该方格所在的行和列全部进行反色（白→黑，黑→白）填充处理（其他行列不变）。

只要反复进行这个处理，无论初始状态如何，一定能使所有方格全部变为白色（图9）。这里我们要按照能以最多次数把所有方格都变成白色的顺序来选择方格（不能重复选择同一方格）？

例)

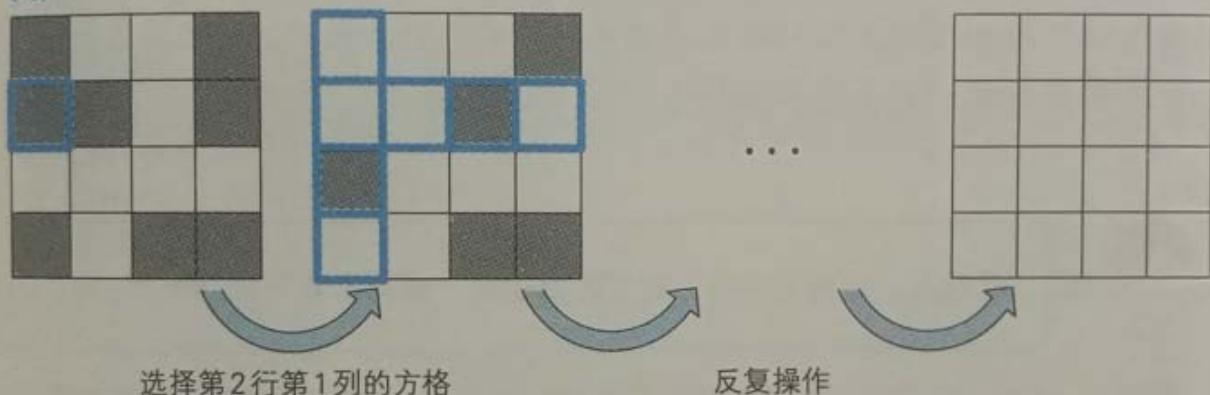


图9 反转方格颜色

问题

总共有 2^{16} 种初始状态。

请思考这种选择方格次数（反色操作的次数）最多的初始状态，并求这个最多次数是多少。举个例子，如图10所示的情况只需要3次操作就能把全部方格变为白色（图10）。

这与上面的“以最多次数...”的要求不是矛盾吗？

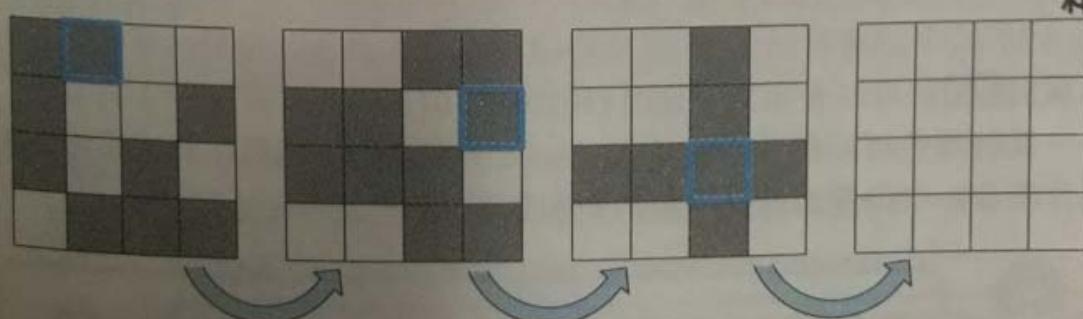


图10 全部填充为白色时的示例（方格选择次数为3次）

题设中所说的“。。。要按照能以最多次数把所有方格。。。”的提法疑似笔误，应该为最少次数。这个与后面问题中的“最多”并不矛盾。先求每种状态下到达全白状态所需要的最少翻转次数，然后再比较所有各状态(到达全白状态)所需最少翻转次数的最大值，相当于一个 Maxmin 问题(更常见的形式是 Minmax 问题)

38.2 解题分析

把每种盘面状态看作是一个节点（共有 $2^{16}=65536$ 中状态/节点，本系列中通常把节点和状态交换使用），把各状态到达全白状态所需要最少翻转次数视为该节点到达全白节点的距离，本题可以看作是在由这 65536 个节点构成的图中距离全白节点最远的点。当前这里隐含了一个前提，就是说这个图是全连接图，或者说任意一个状态出发经过有限次翻转操作后都能够到达全白状态。

38.2.1 Naïve Approach

作为 Naïve approach，遍历从每个状态出发，然后搜索它们到全白状态的距离（即所需翻转次数），然后再进行比较。这样做当然也可以，但是将会导致巨大的重复和冗余搜索。作为一种改进方案，可以采用“递归+memoization”的策略，将已经搜索得到的结果记忆下来，较长距离的节点的搜索可以利用较短距离的节点的搜索结果。这样相比上述 Naïve approach 虽然可以得到巨大的性能提升，但是仍然不够好。

38.2.2 最大路径问题—广度优先搜索

由于翻转操作是可逆的，上述的图是一个无向图，A 和 B 两个节点的距离从 A 向 B 搜索与从 B 向 A 搜索会得到相同的结果。采用逆向思考（本系列中已经出现了多道基于逆向思考策略的题目了，等做完了本系列考虑做一次全面的各种算法策略的总结），从全白状态反向搜索到达各节点的距离，寻找其中最大值，这个问题就转化为从固定起点开始的图搜索中的最长路径搜索问题了，这类问题的经典策略是广度优先搜索。

注意，不仅仅最短路径搜索可以用广度优先搜索，最长路径搜索也可以用广度优先搜索。

在最短路径搜索中，是一旦找到目标点就停止搜索。与之不同的是，在最长路径搜索中，要遍历所有的点，最后到达的那个节点就是距离最大的节点。

38.2.3 节点状态表示及翻转操作

4×4 的棋盘共有 16 个方格，其黑白状态可以用 16 比特的二进制数来表示。

而翻转操作则可以以 bit-wise 异或运算来实现。对于所选中的每一格方格，其对应的翻转操作可以表示为一个 16 比特的二进制数，称之为掩码。让掩码与表示当前状态的 16 比特二进制数进行 bit-wise 异或运算即等价实现了题设所要求的翻转处理。16 个方格对应 16 种掩码可以提前计算好备用。

如下图所示为一个运算示意图，包括棋盘状态、操作掩码的二进制表示及运算结果（'0b'开头表示这

是一个二进制表示):

Q38. chenxy. 2021.9.24

	0	1	2	3
0	1	0	0	1
1	1	1	0	0
2	1	0	0	1
3	0	1	1	0

选择方格 [1,2] 作为翻转的枢轴点，
它所对应的翻转模式为

0	0	1	0
1	1	1	1
0	0	1	0
0	0	1	0

当前盘面状态:

$ob1001_1100_1001_0110.$

对应掩码可表示为 $ob0010_1111_0010_0010$

因此，翻转后的盘面状态为:
 $(ob1001_1100_1001_0110)$
 或 $(ob0010_1111_0010_0010)$
 $ob1011_0011_1011_0100$

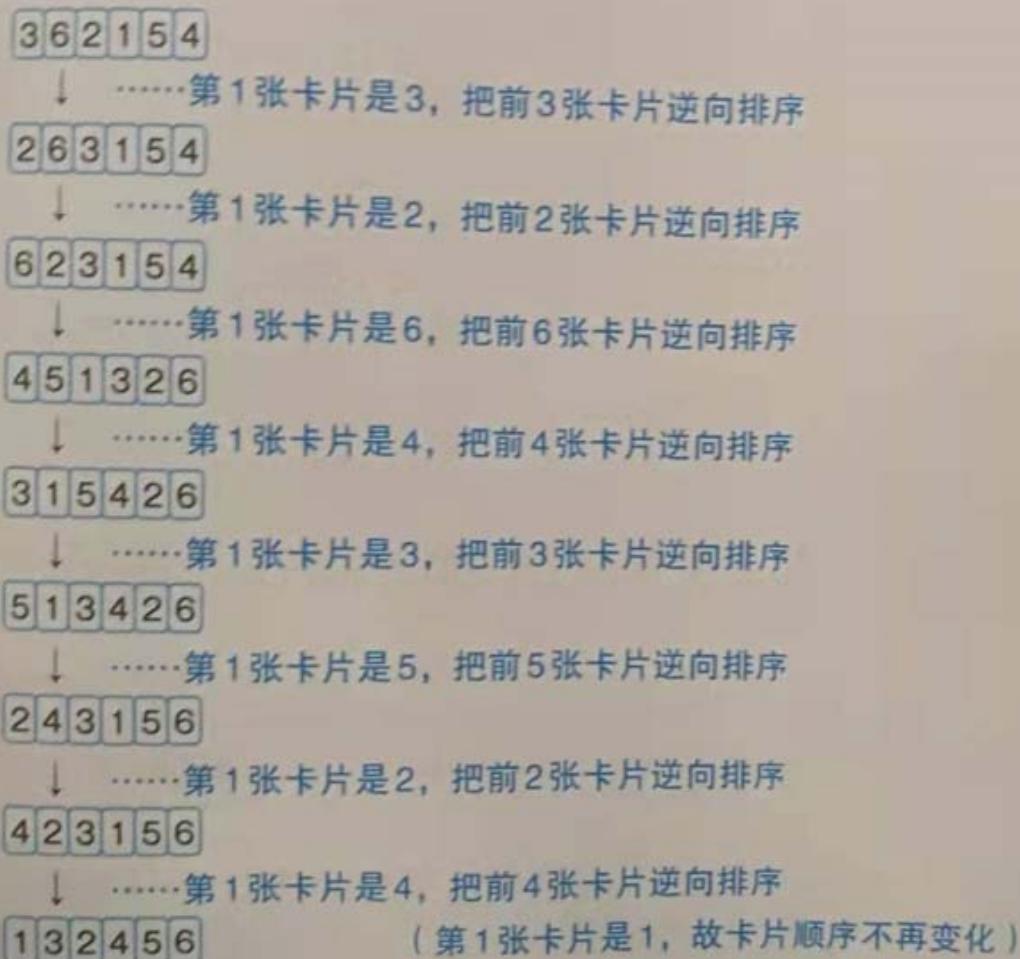
38.3 后记

本题是到目前为止第一道 coding 完未经调试直接运行正确的题目，有点小小的得意^-^。
 算法编程解题确实也是一个“无他、唯手熟尔”的事情。前面有些题目做的非常费劲，是因为不管是针对问题本质的洞见、算法策略、Python 编程等都很不熟悉（就是对各种套路和飞刀不熟悉），但是随着对这些技能的逐渐熟练的掌握，现在的解题就一点一点地变得轻松起来了。

39. Q39: 反复排序

39.1 问题描述

假设有标注了 $1, 2, 3, \dots, n$ 各个数字的 n 张卡片。当第 1 张卡片的数字为 k 时，则把前 k 张卡片逆向排序，并一直重复这个操作。举个例子，当 $n = 6$ 时，如果由“362154”这个序列开始，则卡片的变化情况如下。



这种情况下，卡片顺序一共变化 8 次后就无法继续变化了。

问题

求当 $n = 9$ 时，使卡片顺序变化次数最多的 9 张卡片的顺序。

39.2 解题分析

把每种排序状态看作是一个节点（共有 $9! = 362880$ 种状态/节点，本系列中通常把节点和状态交换使用），把各状态到达“终点”状态所需要最少重排次数视为该节点到达“终点”的距离。到此为止，本问题似乎跟 Q38 是完全相同类型的问题。但是，本问题与 Q38 相比有一个根本性的差异：不存在一个统一的终点。Q38 中的统一的终点是“全白”状态。而本问题中，从任意状态出发，它对应的“终点”状态是以 1 开始的排列，总共有 $8! = 40320$ 中可能的“终点”状态！所以不能单纯地像 Q38 那样采样逆向思考来解决问题。

39.2.1 Naïve Approach—正向全量搜索

作为 Naïve approach，遍历从每个状态出发，然后搜索它们到某个“终点”状态的距离（即所需重排次数），然后再进行比较。算法流程如下：

遍历所有的 1~9 的排列，for each of one(denoted as start):

```

Step = 0
While start[1] != 1:
    根据题设条件进行翻转
    If maxstep < step:
        Maxstep = step
        Maxorder = start

```

39.2.2 缩小搜索范围

根据题设的重新排列规则，任何一个排列状态，只要它满足以下条件：它的第 k 个数恰好为 k ，则它肯定可以由将前 k 个数逆序排列得到的状态按照题设规则重新排列而得。因此它肯定距离最远的状态。比如说， $[1,3,4,6,5,2,7,8,9]$ 的第 5 个数恰好是 5，它可以由 $[5,6,4,3,1,2,7,8,9]$ 根据题设规则重排而得。满足这样条件的排列就可以从搜索列表中排除出去，可以节省一定的搜索时间。算法流程如下：

遍历所有的 1~9 的排列，for each of one (denoted as start):

```

如果 start 的任意第 k 个数恰好为 k，则跳过；否则继续以下搜索：
Step = 0
While start[1] != 1:
    根据题设条件进行翻转
    If maxstep < step:

```

Maxstep = step

Maxorder = start

39.2.3 以递归的方式实现

从任意状态开始的搜索过程也可以用递归的方式实现。递归函数的实现流程如下：

Def recur(cur, start, step):

If cur[0]==1:

 到达终点

 更新 maxstep 和 maxorder

 根据题设规则计算下一个状态 cur→nxt

 递归调用：recur(nxt, start, step+1)

39.2.4 反向搜索

虽然，前面说了不能像 Q38 那样单纯从单一状态逆向搜索来求解。但是，毕竟可能的终点状态数只是 $8!$ ，是所有可能状态数 $9!$ 的 $1/9$ ，所以从每一个可能的终点状态进行逆向搜索还是大大缩小了搜索范围（？）。不过事情并没有这么简单。正向搜索的话，虽然起点比较多，但是从起点到终点是单一路径（即每个状态向下一个状态转移是唯一的）；而反向搜索的话，则从一个状态可能到达多个状态（对应于正向搜索中，可能从多个状态出发到达同一个状态，比如说[2,1,3]和[3,2,1]根据题规则都是到达[1,2,3]）。所以反向搜索的起点数虽然少，但是从的搜索复杂度不一定比正向搜索低。定量的分析很难（超出了本渣的能力范围），所以是骡子是马拉出来遛一遛，不妨写出代码来比一比。流程如下所示：

遍历所有的 1~9 的排列，for each of one (denoted as start):

 如果 start 的第一个数不为 1 则跳过；否则继续以下搜索：

 以 start 为起点执行反向的广度优先搜索

 根据搜索结果更新 maxstep 和 maxorder

40. Q40: 优雅的 IP

40.1 问题描述

可能本书大部分读者都清楚，IPv4 中的 IP 地址是二进制的 32 位数值。不过，这样的数值对我们人类而言可读性比较差，所以我们通常会以 8 位为 1 组分割，用类似 192.168.1.2 这种十进制数来表示它（图 1）。

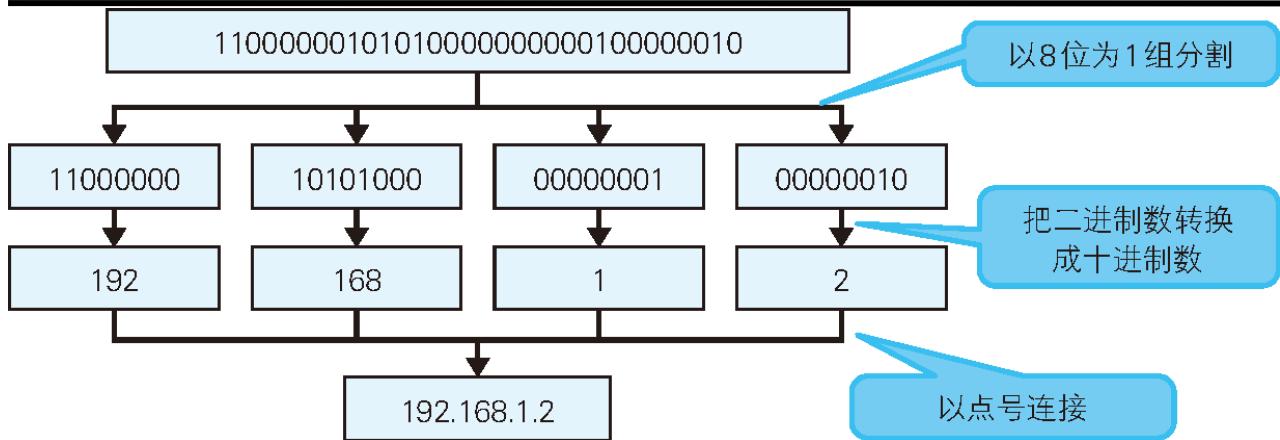


图 1 IP 地址 (IPv4)

这里，我们思考一下十进制数 0~9 这 10 个数字各出现 1 次的 IP 地址（像正常情况一样，省略每组数字首位的 0。也就是说，不能像 192.168.001.002 这样表示，而要像 192.168.1.2 这样来表示）。

问题：求用二进制数表示上述形式的 IP 地址时，能使二进制数左右对称的 IP 地址的个数（用二进制数表示时不省略 0，用完整的 32 位数表示）。

40.2 解题分析

40.2.1 笨办法

10 个数共有 $10!$ 种排列。IPv4 的第一个字段允许为 0 吗？如果不允许的话，则应该是 $9 \times 9!$ 。注意，题中的要求是省略每组数字首位的 0，但是单独的 0 至少对于后面几个字段是允许的。这里先假定第一个字段也允许为 0，反正只是定性的分析，差异不大。

将每种排列分割成 4 段，相当于在 9 个位置中插入 3 个分割点。如果第一个数是 0，则 0 后面必须放一个分割点，则总共有 $C(8,2)$ 种分割点放置方式；如果第一个数不是 0，则分割必须不出现在 0 的前面，因此总共有 $C(8,3)$ 种分割点放置方式。综合考虑有 $C(8,2)+9 \times C(8,3)$ 种分割点放置方式。

然后再进一步去判断这每一种分割方式所生成的 IP 是否符合题设的要求（每个字段必须在 [0,255] 之间，且转换成 32 比特二进制表示后左右对称）。

因此总共需要检查 $(10!) \times (C(8,2)+9 \times C(8,3)) = 1930521600$ 种可能的 IP。这是一个巨大的数字，有没有办法削减搜索范围？

40.2.2 逆向思考

IPv4 的每个字段为 8 比特，十进制表示范围是从 0~255。以 A、B、C、D 分别表示第 1、2、3、4 个字段的十进制表示的话，如果使得整体用 32 比特表示的二进制为对称的话，则必然 A 和 D、B 和 C 作为 8 比特的二进制表示分别构成对称对（顺便说一下，原书的提示不知道是不是翻译的问题，说的非常含糊）。

容易引起误导)。因此，事实上只要对两个 8 比特数的组合进行扫描，对每一种组合在进一步判断十进制表示是不是刚好包含 0~9 各 1 个的 10 个数字。共有 $256 \times 256 = 65536$ 种可能的组合，这个组合数远远地小于上一节的方案。

以下代码按照这种思路来实现。其中有一些小巧的细节，说明如下：

1. 十进制数转换成二进制数用 `bin()`，但是 `bin()` 的输出前面包含'0b'前缀。所以后跟[2:]去除这个前缀。

```
Abin = bin(A)[2:]
```

2. 求对称的二进制序列。由于 `bin()` 的输出可能不满 8 比特长。但是在判断二进制序列是否相互对称时是考虑双方都是 8 比特 (不足 8 比特的先头补 0)，因此在求与 A 互补的 D 的二进制表示时首先用 `Abin[::-1]` 取 Abin 的 reverse，然后再在尾部根据需要补 0 (因为 A 是头部补 0)

```
Dbin = Abin[::-1] + (8-len(Abin))*'0'
```

3. 将二进制转换成十进制，`int()` 函数有参数只是原字符串的进制数，二进制的话则设为 2。

```
D = int(Dbin,2)
```

4. 判断是否是刚好 0~9 各包含一个的 10 个数。将 A、B、C、D 变成字符串然后串联起来，然后再转换成 set (因为 set 的元素是 unique 的，自动剔除重复的元素)，因此最后只要判断该 set 的长度是不是 10 即可。

```
len(set(combinedStr)) == 10
```

41. Q41: 只用 1 个数字表示 1234

41.1 问题描述

这里我们思考一下通过四则运算，只使用 1 个数字来表示某个数的情况。例如 1000 这个数，如果只用 1，则可以用 7 个 1，即 $1111 - 111$ 来表示；如果只用 8，则可以用 8 个 8，即 $8 + 8 + 8 + 88 + 888$ 来表示；如果只用 9，则可以用 5 个 9，即 $9 \div 9 + 999$ 来表示。

假设我们只能使用四则运算符（+、-、×、÷），不能使用改变运算优先度的括号，而运算顺序同数学上的运算法则，即“先乘除后加减”。此外，使用除法运算时结果只取整数（譬如 $111 \div 11 = 10$ ）。

问题

求只用 1 个数字表示 1234，且要尽可能少地使用该数字时，使用哪个数字才能使该数字出现个数最少呢？最终的算式又是怎样的呢？



四则运算我们在 Q02 中学习过。不过如果只用 1 个数字，会不会无论用多少个该数字，也没办法表示最终的结果呢？感觉会陷入无限循环。



如果用“1”来表示最终的结果，那么通过简单的加法（“最终结果”个 1 相加）就能完成。如果使用其他数字，那么最多使用“最终结果”的两倍”那么多的数字，应该也就可以了。例如，如果使用数字“9”，则可以通过“最终结果”个 “ $9 \div 9$ ”相加来表示最终结果。

41.2 解题分析

除了最“笨”的暴力搜索以外，没有找到什么头绪。

这个问题涉及到几个维度的搜索：

- (1) 用哪个数字
- (2) 用几个
- (3) 如何构成表达式

这种多个维度的搜索问题，维度的搜索顺序很重要。错误的维度搜索顺序可能会导致额外的时间甚至导致陷入死循环。比如说，本题如果以选择哪个数字为第一搜索维度的话，有些数字可能会不管用多少个都无法构成符合条件的表达式，因此就会陷入无限循环。

本题的最关键的一点是要用最少个数的数字。因此应该以个数为第一搜索维度。

算法流程如下：

K = 1

While 1:

For n in [1,2,3, ..., 9]:

由 k 个 n 以及{+,-,*,/,"}构成任意可能的表达式

评估以上得到的表达式确定是否有满足条件(等于 1234)，如果是则结束搜索，否则继续

K = K + 1

有几个细节需要注意：

- (1) 关于除法是采用整数除法，在 python 中有“//”表示整数除法，可以方便使用
- (2) 允许多位数的存在。包括多位数的遍历有不同的方法，本题中采用的方式，假设另有一个空的运算符”，如果两个数字之间插入一个空运算符“就表示构成了两位数。构造多位数的话则一次类推。这样做的好处是空操作符“和其他 4 个有效运算符可以统一对待，因此由 k 个 n 构成的表达式时就相当于在 k 个 n 的(k-1)个位置上任意插入不同的运算符，处理起来非常方便。这一构造方法最先用于 Q2 的解答，请参考。
- (3) 构造好表达式后，如何评估表达式的值。可以自写代码实现但是比较麻烦。Python 中有 eval()实现了这一功能，本题的焦点不在于表达式的值如何评估，所以在本题解中就偷懒使用 eval()了。对于如何自写代码评估感兴趣的话，可以参考 Q2 的题解。

41.3 后记

只想到了暴力搜索的方法，好在运行时间似乎还在可以接受的范围。至于是否还有更好的解法，在“偷看”答案之前，（反正已经有了一个解）还是让“思考”再飞一会儿^-^.

42. Q42: 30 人 31 足游戏

42.1 问题描述

本题来自《程序员的算法趣题》中的第 42 题。

假设有 $2n$ 张扑克牌，每次我们从中抽取 n 张牌（不是分散地抽取，而是抽取连续的一沓牌）放置到牌堆的顶上。然后重复这个操作，直到牌的顺序和最初顺序相反。

请问，当 $n=5$ 时，要使 10 张牌逆序排列最少需要经过多少步？

42.2 分析

$N=1$. 显然只需要 1 次操作就能将排序颠倒。

$N=2$. 假定初始序为 [1,2,3,4]，假定左边表示在排队上面的位置。则可以通过以下操作步骤将原排序颠倒：[1,2,3,4] \rightarrow [2,3,1,4] \rightarrow [3,1,2,4] \rightarrow [2,4,3,1] \rightarrow [4,3,2,1]，即 4 次操作可以将原排序颠倒（参见以下附图 1）。但是，能证明 4 次是最少需要的次数吗？

$N=3$ 的情况要仅靠纸和笔来计算出来已经超出了一般人的大脑承受范围了。

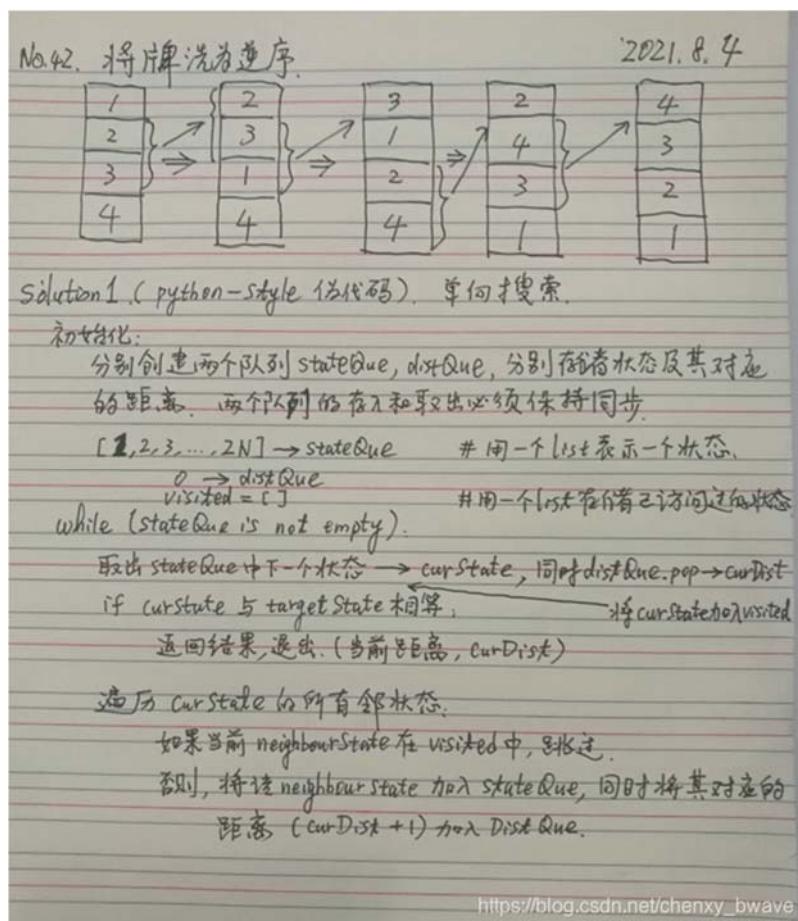
牌堆的每一种排序代表一种状态，当有 $2N$ 张牌时有 $(2N)!$ 种状态。当 $N=3$ 时，状态数已经增长至 $(6!) = 720$ 了，所以用手算的方式求解几乎不可能了。

问题可以转化为从初始状态（不失一般性可以记为 $[1,2,3,\dots,2*N]$ ）通过题目所约定的操作到达终止状态 $[2*N, 2*N-1, \dots, 3, 2, 1]$ 所需要的最小步数。这其实也就是图的两个节点之间的最短距离的问题。解决这个最短距离问题的“标准”算法是广度优先搜索（BFS: Breadth First Search）。

广度优先搜索的几个要素：

- (1) 状态表示。本题中可以用 $2*N$ 个数的一个排列来表示一种状态，为了计算给出最短距离，还需要记录某个状态与初始状态的距离。
- (2) 状态之间的转移，或者说求一个节点的邻节点。在本题中，从每一种排列出发总共有 N 种抽取连续 N 张牌的方式，每种抽取方式得到一种新的状态。因此每个节点有 N 个邻节点。
- (3) 已访问节点的记忆。这是为了避免重复访问已经访问过的节点。

42.3 BFS 算法描述



42.4 初始实现

运行结果:

N = 1, ans = [1, 2], tCost = 0.0(sec)

N = 2, ans = [4, 11], tCost = 0.0(sec)

N = 3, ans = [7, 841], tCost = 0.029889583587646484(sec)

N = 4, ans = [8, 28889], tCost = 29.551127672195435(sec)

ANS 的第一项代表所需要的步骤数, 而第二项代表所访问过的节点或者状态数。

结果分析:

N=4 的耗时相比 N=3 增长了约 1000 倍! 定性分析的话, 一方面是所需要访问节点/状态数大幅度增长了; 另一方面, 每个状态的表示长度也从 6 变为 8, 因此单次存取和状态比较的时间也变大了。但是这个似乎不能完全解释 1000 倍的比例。

按照这个增长速度往下走的话，很显然 $N=5$ 的运行计算将会变得难以忍受的长。需要从两个方面考虑优化：(1) 算法的优化；(2) 实现方式的优化，比如说存储中间数据的存储数据结构等的优化。

另外， $N=3$ 时，总共只有 $(2^N)=720$ 种节点状态，为什么访问节点状态数会达到 841？这是不是意味着当前的基本实现本身还存在 bug？

42.5 Bug-fix and Optimization

42.5.1 When to update visited?

在深度优先搜索的非递归实现中，一个容易犯错误的坑是更新 visited 的时机。

在以上实现中，在出栈的时候将从栈中取出的节点（状态）加入到 visited，而这正是导致总的访问节点状态数竟然超过了总的可能状态数的情况。实际上应该在入栈的时候就同时加入到 visited 中去。在算法教科书中这一点肯定在算法流程介绍中已经列出来了，但是由于显得那么理所当然其实并不一定会注意到这个细节或者想到为什么要这样做呢？只有在实际实现中碰到这个问题并且踩过坑才会认真地思考为什么。以下简要解释一下。

在广度优先搜索中，节点是分层的，每一层在栈中挤在一起。考虑当前层的节点有 N_1, N_2, \dots, N_k ，在从栈中依次读出 N_1, N_2, \dots, N_k 处理并将下一层的节点 M_1, M_2, \dots, M_x 时需要判断它们是不是 visited 过。存在这样的可能，比如说， N_1 的邻(或子)节点有与 N_2, \dots, N_k 重叠的，这样就会如果是在出栈时才加入到 visited 中的话，那么这个子节点就会被再次加入到 visited，从而造成重复。

42.5.2 What to implement visited?

Visited 需要进行大量的查询。在初始实现中是采用 list 来实现的。

在 python 中，dict()的查询要远远地快于 list 的查询。

用 dict()替换 list 实现 visited 后，运行速度提高了接近三个数量级！

当然，初始实现中节点状态是用 list 表示的，而 dict 不接收 list 作为 key，所以优化实现中改为用 tuple 来表示状态。

42.5.3 Queue or deque?

用 collections.deque 替换 queue.Queue 后，运行速度也得到了一定的提高。

42.6 优化实现：代码及测试

43. Q43:让玻璃杯水量减半

43.1 问题描述

有 A, B, C 这三个大小各不相同的玻璃杯。从 A 杯装满水、B 和 C 空杯的状态开始，不断地从一个杯子倒到其它杯子里去。假设不能使用任何辅助测量工具，且倒水时只能倒到这个杯子变为空，或者目标杯子变为满。重复这样的倒水操作，使 A 杯剩余水量是最初的一般。举个例子，如果 A、B、C 的初始容量分别为 8、5、3，则可以通过以下操作序列使得 A 的水量变为 4：(A to B), (B to C), (C to A), (B to C), (A to B), (B to C), (C to A)。读者可以自行手动演算验证。

在 B 和 C 的容量互质，且满足 $B+C=A$, $B>C$ 的条件下，当 A 为 10~100 之间的偶数时，请问能使得“倒水操作后 A 杯水量减半”的 A、B、C 容量的组合有多少个？

43.2 解题分析

图搜索问题，深度优先递归搜索（随口杜撰的名词大杂烩。。。做了一些题后一些概念开始在脑子里乱炖到一块儿了，后面要适时地总结整理夯实一下地基巩固一下训练成果）！本系列中有相当多题目都是这一个类型，用同样的套路就可以解决，后面有时间要回头来做一次总结。相比之下，原书还提供了另外一种更为精妙的解法，但是那个是只适用于当前题目的特定技巧，没有通用性。

图搜索问题的过程的关键就是构建搜索树，这一类问题的通用解题思路的要点：

- (1) 节点状态表示
- (2) 邻节点（或子节点）搜索
- (3) 路径历史记忆以及判断邻节点是否在路径历史中

通用很重要！灵光一现的解题技巧（可遇而不可求）就留给天才们去做好了。掌握了一个通用技巧，你可以确保碰到一个同类型的问题你有一个重型坦克般的保底的解决方案，虽然有时候不免显得笨拙，但是没有什么能拦得住！

43.2.1 节点状态表示

本题节点状态很简单，就是当前三个杯子中的水量，用列表[a,b,c]表示即可。

43.2.2 邻节点搜索

邻节点搜索就是指搜索从当前状态/节点能够去往的下一个节点/状态，这些邻节点在搜索树中就对应着当前节点的子节点。所以这里邻节点和子节点是可以互换使用的等价概念。

但是邻节点要避免回到当前路径上已经到达过的节点，因为那样的话就形成了环路（loop），破坏了树的结构。如何防止形成环路参见下一节。

43.2.3 路径历史记忆以及判断邻节点是否在路径历史中

与单纯的深度优先搜索（for reachability judge only）不同的是，本类问题需要搜索所有可能的路径（呃。。。后来发现我误解了题目，主动提高了解题要求，不过油多不坏菜，就按‘误解’后的扩展版本来解吧，反正扩展版本包含了原题的答案），不同路径有可能共享一部分的节点或者甚至一部分 edge。所以在搜索过程中需要记住当前搜索路径的历史，而不是一个全局的 visited，因为只用于防止本路径形成环路。每条路径的搜索需要维护自己的路径历史。

在本题解中，用 python dict 来存储路径历史。但是由于 python dict 不能使用 list 作为 key，所以将表示状态的列表[a,b,c]转换为 tuple 后再用作 dict 的 key。那为什么不直接用 tuple 表示节点/状态呢？这是因为 tuple 的值不能修改，对于在处理过程需要更新状态值时不太方便。当然这些都不过是细枝末节。

在每次递归调用时，将当前节点/状态加入 pathHistory，然后在退出本次递归调用时又将进入本次递归调用时加入的当前节点/状态清除掉。这相当于伴随着递归调用的隐式栈，并行地维护了一个显式的路径历史堆栈。我还没有想清楚这个是不是不可避免的，或许有什么办法可以回避掉。。。有时间再琢磨琢磨。

44. Q44: 质数矩阵

44.1 问题描述

在 n 行 n 列的方格内逐位填写 n 位数的质数，要求不仅横向数字（左→右）是质数，纵向数字（上→下）也要是质数，但相同的质数不能出现多次（只能使用 n 位数的质数，且排除 0 开头的数字）。

举个例子，当 $n = 2$ 时，如图 15 所示，①和②的情况符合要求。
①中的质数是 11、13、17、37，而②中的质数是 23、29、37、97，分别使用了 4 个质数。在③中，17 和 73 都出现了 2 次，因此不符合题意。

①	②	③(NG)	④
1 3 1 7	2 3 9 7	1 7 7 3	1 2 7 3 1 3 1 1 3

图 15 当 $n = 2, n = 3$ 时的示例

问题

求当 $n = 3$ 时，符合要求的数字排列方式有多少种？例如，我们可以使用 113、127、131、211、313、733 这 6 个质数组成上述④的排列方式（另外，矩阵沿对角线翻转后即使质数不变，排列方式也要另外计数）。

44.2 解题分析

一共 9 个格子，每个格子都从 $\{0,1,2,\dots,9\}$ 中可以任选数字的话，有 10 的 9 次方（ 10 亿！）中可能的组合。就这样直接遍历的话会需要很长的时间，需要考虑缩小搜索范围的策略。

考虑 9 个格子中的数字（按行优先排列）分别记为 a_0, a_1, \dots, a_8 。

由于质数肯定不能是偶数，所以个位数不能为 $\{2,4,6,8\}$ 。同样也不能为 5。因此 a_2, a_5, a_6, a_7, a_8 的可选范围缩小为 $\{1,3,7,9\}$ 。题设要求限定为三位质数，故首位数字不能为 0，所以 a_0, a_1, a_2 不能为 0。只有 a_4 的选择范围仍然为 $\{0,1,2,\dots,9\}$ ，这样总的可能组合数缩小为：

$$9^3 * 4^5 * 10 = 7464960$$

仅为原始的 10^9 的 $(1/144)$ 。

进一步，还可以利用 early-stop 的策略来提高搜索速度。因为总共有 6 个质数要考察，安排遍历的顺序使得可以尽量早地进行质数判断，如果发现不满足条件，就可以从某条搜索路径提前退出。比如说，已

经安排了 a0,a1,a2 后，就可以先判断(a0,a1,a2)构成的三位数是否为质数；接下来，对 a3,a6 进行遍历，就可以对(a0,a3,a6)构成的三位数是否为质数进行判断。据此，考虑由外到内的遍历顺序为 a0-a1-a2-a3-a6-a4-a5-a7-a8.

题设还要求质数不能重复。因此在以上遍历过程中，将搜索路径上得到的满足条件的质数记录下来，新的质数还要判断是否已经存在与该列表中，如果有不满足条件的也可以提前退出。这样做比把 6 个质数都凑齐了再判断是否有重复可能能够节约一些时间。在以下代码中，用一个 list 来实现一个 stack(FIFO)，每得到一个新的满足条件的质数就加入到栈中；相应地，退回到上一层的时候要把本层加入到栈中的数要退出（**因此代码中 prime_lst.append() 与 prime_lst.pop() 是成对出现的**）。这个有点类似于深度优先搜索中的入栈退栈处理。因此本题解也可以用递归的方式实现，但是由于针对每个数的搜索范围不完全一致，所以递归方式也有点额外的麻烦。

44.3 后记

本题也有多种方案可用，但是没时间，暂时先只给这一个题解，后面有时间再回头补充。

本题的难度感觉比之前很多题目都要低（甚至低得多）啊，考虑到本书的编排的原则是从易到难安排的，感觉有点奇怪。或者只是由于不同人的思维习惯导致难和易的感觉只是一个相对的感觉？

45. Q45: 排序交换次数的最少化

45.1 问题描述

排序可以说是算法的基础，其实现方法有很多。这里我们先不关注处理速度，来思考一下交换次数最少的排序方法。

举个例子，假设要通过反复交换 1、2、3 这 3 个数字中的 2 个数字，来得到升序有序数列。那么初始数列不同，排序时需要的最少交换次数也不同。

例) 1, 2, 3 → 不用交换

1, 3, 2 → (2 和 3 交换) → 1, 2, 3 (1 次)

2, 1, 3 → (1 和 2 交换) → 1, 2, 3 (1 次)

2, 3, 1 → (1 和 2 交换) → 1, 3, 2 → (2 和 3 交换) → 1, 2, 3 (2 次)

3, 1, 2 → (1 和 3 交换) → 1, 3, 2 → (2 和 3 交换) → 1, 2, 3 (2 次)

3, 2, 1 → (1 和 3 交换) → 1, 2, 3 (1 次)

也就是说，如果是 1、2、3 这 3 个数字，那么最少交换次数之和为 7。

问题

求对于由 1~7 这 7 个数字组成的所有数列，执行以最少交换次数求得升序有序数列的处理时，这些最少交换次数之和。

45.2 解题分析

考虑：N 个数字的每种排列看作是一个节点，邻节点是指能通过交换任意两个位置的数得到的新的排列。这样，所有 N! 个排列一个连通图。能以最少交换次数到达升序有序排列（记为 B）的数列（记为 A）就等价于从 A 代表的节点在这张图中到达 B 对应的节点的最短路径长度。

进一步，“交换任意两个位置的数”是可逆的操作，这是一个无向图。因此，从节点 A 到达节点 B 的最短路径长度，等于从节点 B 到达节点 A 的最短路径长度。

所以本题求解的其实就是在这种图中，从节点 B 点其它所有各节点的最短路径长度之和。而求最短路径长度的标准解法就是广度优先搜索。从节点 B 出发通过广度优先搜索遍历所有节点，记录下每个节点的层数（距离），最后求和即可。

广度优先搜索（BFS）的基本流程（即便在本系列也出现过很多次）这里就不再赘述（不熟悉的伙伴可以参阅前面的题解）。

在一般的 BFS 流程中，用 visited 只需要记录已访问过的节点，而无需记录其对应的距离。本题解在最后统一进行距离求和，所以必须将每个节点的距离记录下来，最自然的做法当然是在 visited 中将节点和距离信息一起记录下来，因此在本题解中用 dict() 实现 visited（一般只记忆节点的话用 set() 即可）。

46. Q46: 唯一的 0x 序列

46.1 问题描述

在 n 行 n 列的矩阵中排列 \circlearrowleft 和 \times ，并统计各行各列的 \circlearrowleft 的个数。举个例子，当 $n = 3$ 时，我们可以像图 16 中的①这样统计个数。

①	②
$\circlearrowleft \times \circlearrowleft \rightarrow 2$	$\times \circlearrowleft \circlearrowleft \rightarrow 2$
$\times \circlearrowleft \circlearrowleft \rightarrow 2$	$\circlearrowleft \times \circlearrowleft \rightarrow 2$
$\circlearrowleft \times \times \rightarrow 1$	$\circlearrowleft \times \times \rightarrow 1$
$\downarrow \downarrow \downarrow$	$\uparrow \uparrow \uparrow$
2 1 2	2 1 2

图 16 统计 \circlearrowleft 的个数并重新排列的示例 1

然后，反过来根据①的计数结果重新排列每行每列的 \circlearrowleft 和 \times 。这样，我们就可以得到像图 16 中的②这样的结果，即 \circlearrowleft 的个数和①相同，但位置排列不同。

不过，如果是像图 17 中的③的情形，即便根据统计结果反过来重新排列，也只能排列在与原来一模一样的位置上（图 17 中的④）。

③	④
$\times \times \times \rightarrow 0$	$\times \times \times \rightarrow 0$
$\times \circlearrowleft \times \rightarrow 1$	$\times \circlearrowleft \times \rightarrow 1$
$\times \circlearrowleft \times \rightarrow 1$	$\times \circlearrowleft \times \rightarrow 1$
$\downarrow \downarrow \downarrow$	$\uparrow \uparrow \uparrow$
0 2 0	0 2 0

图 17 统计 \circlearrowleft 的个数并重新排列的示例 2

当 $n=4$ 时，像上述例子一样，根据统计结果重新排列 \circlearrowleft 和 \times 的位置，只有一种排列方式的 \circlearrowleft 和 \times 的

排列一共有多少种呢？

46.2 解题分析

因为是对 O 计数，可以用 1 代表 O，用 0 代表 x，这样原矩阵就转化为一个二进制矩阵。

以下采用暴力搜索法。

对 $N \times N$ 的所有可能的二进制矩阵进行 N 行和 N 列的，所得的 $2 \times N$ 个值形成的排列 $\{r1_sum, r2_sum, \dots, rN_sum, c1_sum, c2_sum, \dots, cN_sum\}$ 构成这个矩阵的 signature。然后查询值对应唯一的矩阵的 signature 的个数。可以在遍历所有矩阵时，对各种 signature 出现的次数进行计数，最后计数值为 1 的 signature 个数即为所求结果。signature 出现的次数可以用哈希表来存储，在 python 中就是 dict()。

$N \times N$ 的所有可能的二进制矩阵种类数为 2^{N^2} , $N=4$ 时为 65536，随着 N 增大急剧增大。

算法流程如下所示：

```
sigCount = dict()
```

遍历 $N \times N$ 的所有可能的二进制矩阵，for each of one:

分别计算 N 个行和以及 N 个列和得到

```
sig={r1_sum, r2_sum, ..., rN_sum, c1_sum, c2_sum, ..., cN_sum }
```

```
If sig in sigCount:
```

```
    sigCount[sig] += 1
```

```
else:
```

```
    sigCount[sig] = 1
```

统计 sigCount 中值为 1 的 key 的个数

46.3 后记

有两个可能改进方案：

- (1) 用二进制的形式来表示矩阵，以位操作的方式实现行和以及列和计算
- (2) 矩阵中任意一个子矩阵的 4 个顶点按对角线分为两组，一组为全 0、另一组为全 1 的情况下，很明显可以构成出和它所对应相同的 signature 的不同矩阵，因此可以排除在搜索范围之外

以后回头来补上这些改进解。

47. Q47: 格雷码循环

47.1 问题描述

“格雷码”^①是一种数字编码方式，其特征是任意相邻的代码只有1个位元^②不同。举个例子，一般的2进制数表示的1要变为2时，是由001变为010，需要改变2位；3要变为4时，则是由011变为100，需要改变3位。而用格雷码时，这两种情况都只需要改变1位（表5）。

下面我们试试“把n进制数转换成格雷码，把得到的编码结果看作n进制数，再一次转换成格雷码”，并一直重复这个过程，直到转换为与初始值相同的值。

举个例子，当n=2，初始值为100时，是“100→110→101→111→100”这样一个循环，重复转换4次后得到初始值。同样地，当n=3，初始值为100时，转换过程则是“100→120→111→100”，重复3次后得到初始值。

问题

求当n=16时，从808080开始转换，最后得到808080所需的转换次数，以及从abcdef开始转换，最后得到abcdef所需的转换次数。

47.2 解题分析

这道题目的焦点是任意M进制数的格雷码的生成。这个确实是有难度的问题，因为它已经不是常规的算法设计问题。要求对格雷码的生成算法有了解，尤其是针对任意M进制的格雷码的生成并不是一般的二进制格雷码的生成算法的简单推广。

以本问题为契机我特意调查了一下二进制格雷码以及任意M进制格雷码的生成算法。详情参见。。。

我自己基于格雷码的定义给出的基于深度优先搜索的任意M进制格雷码的生成算法由于（是预先一次性生成N位M进制的所有格雷码字）需要巨量的存储，因此并不适用于作为本题的解决方案。对于M=16，N=6的情况，总共有 $M^N=16^6=2^{24} \sim 10^9$ ，这个存储量有点惊人。

表5 格雷码示例

10进制数	2进制数	格雷码
0	000	000
1	001	001
2	010	011
3	011	010
4	100	110
5	101	111
6	110	101
7	111	100

原书中给出一个提示，基于二进制格雷码生成的异或转换算法的扩展得出任意 M 进制格雷码生成算法。。。真的有醍醐灌顶之感！

另外，在 https://en.wikipedia.org/wiki/Gray_code 中给出了一种基于迭代的方式生成 M 进制格雷码的算法。本文先基于这种算法给出题解。关于任意 M 进制格雷码生成的“异或转换”算法后面再来补充。

M 进制格雷码的详细迭代生成算法这里不再解释（事实是我没有看懂这个算法是怎么设计出来的，因此无法解释，感兴趣的伙伴自己就着代码慢慢品味吧）。

整体的算法流程如下：

初始化：

start = 0x808080 or 0xABCD or any other numbers

M: M 进制

N: 数字位数

cur = start

step = 0

while 1:

 step = step + 1

 curGray = baseM_toGray(cur)

 将 curGray 视为 N 位 M 进制自然码所代表的值赋给 cur

 If cur == start:

 Break

48. Q48: 翻转得到交错排列

48.1 问题描述

这里有围成圆形的 $2n$ 张卡片。最开始是 n 张白色卡片和 n 张黑色卡片分别连续排列。接下来，反转连续 3 张卡片的颜色（白色卡片反转成黑色，黑色卡片反转成白色），并重复这样的操作，直到黑色卡片和白色卡片交错排列（反转颜色的卡片张数固定为 3）。

举个例子，当 $n = 3$ 时，如图 19 所示，通过两次反转颜色操作就能达到目标。

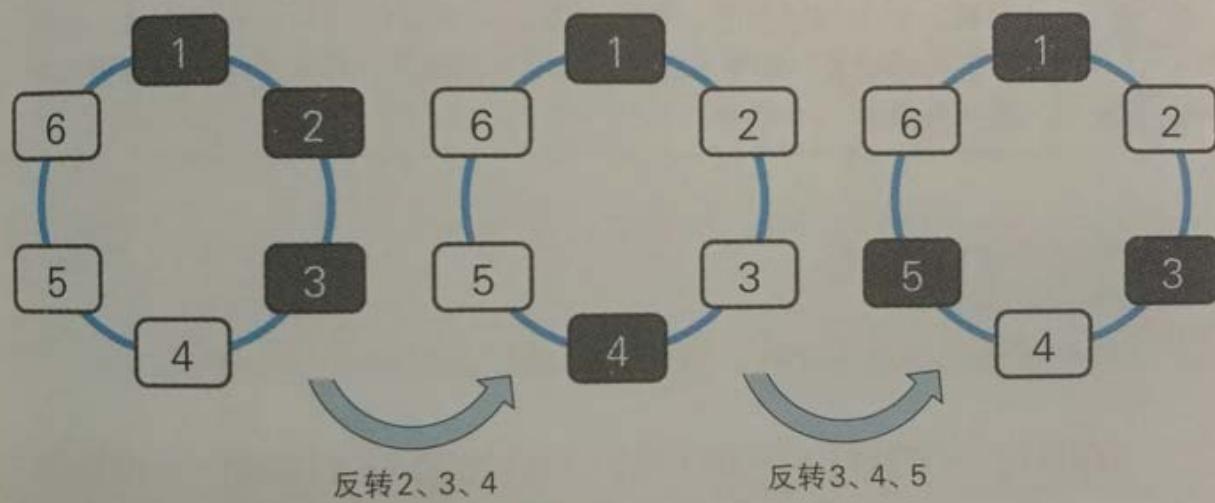


图 19 当 $n = 3$ 时

问题

求当 $n = 8$ 时，使黑色卡片和白色卡片交错排列所需的最少反转次数。

48.2 解题分析

本题关键在于围成一圈的排列以及翻转运算如何表示。

48.2.1 圆圈排列的表示

从某处剪开形成线性排列是常用套路。具体从哪里剪开看处理方便。

由于初始位置是黑白各连续 N 个，用 1 表示黑，0 表示白。标示 2^N 个位置分别为 pos₀, pos₁, ..., pos_(2N-1)。令初始状态中，pos₀~pos_(N-1)为 1（即黑色卡片），pos_N~pos_(2N-1)为 0（即白色卡片）。

从 pos₀ 和 pos_(2N-1)中间剪开，这个排列可以用一个 $2N$ 比特的无符号整数{pos_(2N-1), pos_(2N-2), ..., pos₀}表示，约定 pos₀ 为 LSB，pos_(2N-1)为 MSB。

由此可得：

初始状态为 0b000...0111...1 = 2^{**N-1}

终止状态（黑白或 01 交错）有两种，分别是：0b1010...10, 0b010...101

48.2.2 翻转运算

在圆圈上翻转连续 3 张牌，可以用一个 2^N 比特的有 3 个比特为 1 的整数（称为掩码）与表示排列状态的整数进行按比特异或运算得到。

在每个状态下，“翻转连续 3 张牌”的可能位置有 2^N 种，对应的整数为 7, 14, ...。但是要注意翻转位置跨越剪开的那个位置时的处理。比特数比较少的时候，用手动计算出掩码也是可以的。也可以找出规律用代码来实现（这样代码可扩展性更好）。本题解中用以下方式来求 2^N 种掩码：

```
mask    = 2*N*[0]
for k in range(2*N):
    mask[k] = (7 << k) // (2**16) + (7 << k) % (2**16)
```

$(7 << k)$ 对应于没有位宽限制时从 7 (0b111) 出发的线性左移。求余部分代表没有移出 2^N 比特部分的值。移出的部分在循环左移中挪到了最右端，对应于以上整除部分。

48.2.3 算法流程

基于以上讨论，这个问题就转化成了图搜索问题中的最短路径问题了，可以用广度优先搜索来解决。

算法流程如下：

初始化：

start = 0b0...01...1

target1 = 0b0101...01

target2 = 0b101...010

2^N 个 mask 初始化计算

q, visited 分别用作队列和存储已访问节点

step = 0

q.push((start,step)) #队列中存储状态及距离

将 start 加入 visited

while (q 非空):

 cur, step = q.pop()

 如果 cur 为两个 target 之一，退出搜索，返回 step 值作为结果

 For k = 0,1,...,2*N-1:

 nxt = mask[k] ^ cur

 if nxt not in visited:

 q.push((nxt,step+1)) # Don't forget step increment!

 visited.add(nxt)

49. Q49: 欲速则不达

49.1 问题描述

假设存在如图20所示的长方形，该长方形被划分为了边长为1厘米的正方形方格。假设从A移动到B时，只能在同一条直线上移动2次（可以在同一条路径上往返，但这种情况也算作2次）。另外，这里规定要沿着方格的边移动，且允许交叉通过同一个点。求这种条件下从A到B的最长路径。当长方形宽3厘米，长4厘米时，“OK示例1”的移动距离为7厘米，“OK示例2”的移动距离为13厘米。而像“NG示例”这样，经过同一条直线3次以上是不允许的。

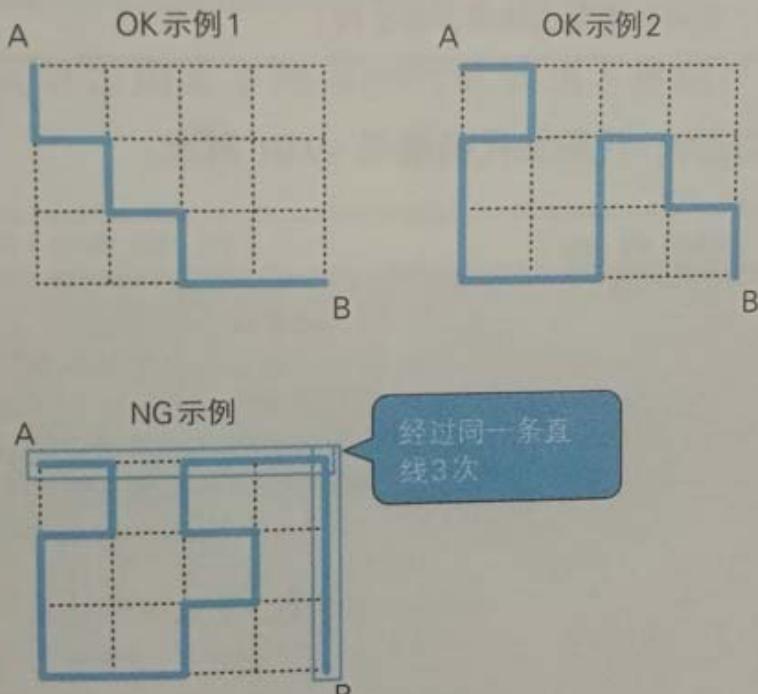


图20 移动路径示例

问题

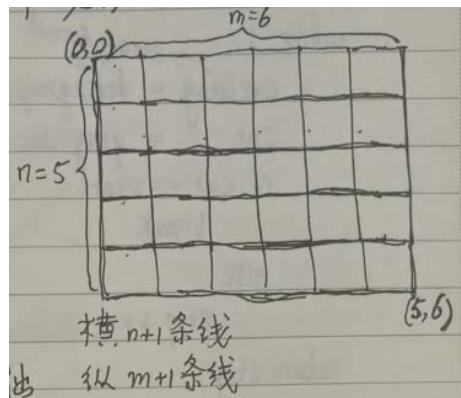
求当长方形宽5厘米，长6厘米时的最长移动距离。

49.2 解题分析

本题是要找最长路径，应该广度优先搜索和深度优先搜索都可以。本文先考虑深度优先搜索。

本题与以前的类似的题目的区别在于约束条件的不同，比如说之前有类似的题目的约束条件是不能

重复通过一条边，或者路线不能交叉，等等。本题的约束条件是在同一条直线上不能移动两次。因此在深度优先搜索过程中需要记录当前搜索路径中每条直线上已经移动过的次数，并结合边界条件来确定从当前位置出发的下一步可能的移动方向。



移动网格如上图所示，以 n 和 m 分别表示纵向宽度和横向长度，出发点坐标为 $(0,0)$ ，目标点坐标为 (n,m) 。横向和纵向分别有 $(n+1)$ 条直线和 $(m+1)$ 条直线，分别用 H 和 V 表示 $(n+1)$ 条横线和 $(m+1)$ 条纵线已经移动过的次数。

深度优先搜索算法流程如下：

```
def dfs(curpos, H, V, steps):
    if curpos == target: # 表示当前路径到达终点，更新 maxsteps
        if maxsteps < steps:
            maxsteps = steps
    # 分上下左右 4 个方向探索是否可以移动
    # Up
    如果可以向上移动的话：
        更新位置为 nxtpos, 更新 H 和 V,
        然后执行递归调用：dfs(nxtpos, H, V, steps+1)
        还原 H 和 V # 很容易忘记，相当于退栈处理
    The almost-identical procedure for Down, Left, Right, respectively
```

最后，将 $curpos$ 初始化为 $(0,0)$ ， $steps$ 初始化 0， H 和 V 分别初始化为适当 size 的全零数组，然后调用 $dfs()$ 到最终退出后所得到的 $maxsteps$ 即为所求结果。在以下代码实现中， $maxsteps$ 指定为全局变量，这样避免了参数传递的麻烦。

以上流程中，“还原 H 和 V ” 比较容易忘记，这个相当于函数递归调用退回时的退栈处理。只不过递归调用的退栈处理是编译器进行了自动管理。但是如果像 $steps$ 那样的传递的话，由于没有更新当前递归

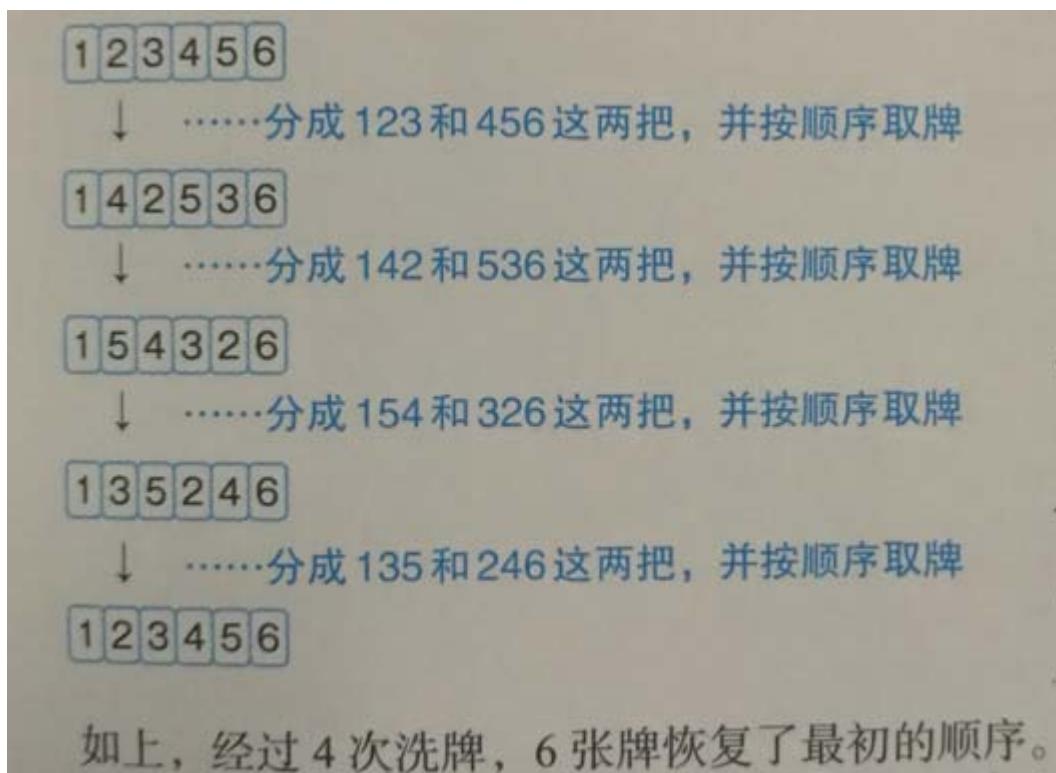
层次的值，所以就不需要还原处理。

50. Q50: 完美洗牌

50.1 问题描述

假设有 $2n$ 张牌，牌上写有用来区分每张牌的文字。把牌叠起来，从正中间把牌分成两把，然后分别从最上面开始，按顺序一张张地取出牌并堆叠起来，我们称这种操作为“洗牌”。

洗几次后，牌会变成最初的顺序。譬如当 $n = 3$ 时，假设 6 张牌上分别写了 1~6 的数字，那么通过以下顺序洗牌可以使牌变成最初的顺序。



问题：对 $2n$ 张牌洗牌，并求当 $1 \leq n \leq 100$ 时，一共有多少个 n 可以使得经过 $2(n-1)$ 次洗牌后，恢复最初顺序？分两种情况考虑：

Case1: $2(n-1)$ 次洗牌后，牌恢复最初顺序

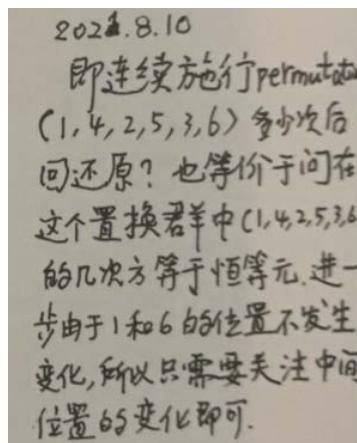
Case2: $2(n-1)$ 次洗牌后第一次恢复顺序

Case2 可以看作是 case1 的一种特殊情况。Case1 的意思是，如果在 m {其中 m 为 $2(n-1)$ 的因子} 次洗牌后回复最初顺序即可。

50.2 解题分析

50.2.1 思路 1

第一感是以下这个‘高大上’的想法：



呃。。。虽说如此，群论只学了一丢丢。。。等我先把群论学一学再来看这条路能不能走通。本系列其实出现过好几道可以用群论来解决的问题。群论学习从开始到放弃经历过好多次了，这次带着问题去学习看看能不能走得远一些。

50.2.2 思路 2

(没有别的更炫的法子时) 蛮干就是了。。。

针对每一个 n ，从初始状态（初始状态是什么并不重要）开始，以迭代的方式进行以上 permutation 操作，并判断是否回到了最初状态。

算法流程如下：

遍历 $n = 1, 2, 3, \dots, 1000$:

Start = [1, 2, ..., 2*n] #任意初始状态都可以，当然这个是最自然的

Cur = start

Cnt = 0

While 1:

 对 cur 执行 permutation

 If cur 与 start 相等 and $(2(n-1)\%cnt) == 0$:

 将 n 加入结果列表

```
break  
If cnt > 2(n-1):  
    break
```

以上是对应 case1 的，将条件 $((2(n-1)\%cnt)==0)$ 修改为 $((2(n-1)==cnt))$ 就对应 case2 了。

实现的要点在于如何实现 permutation。如果置换模式用一个列表表示，对数组用这个列表进行取值的话就可以简单地实现 permutation 了。比如说，原数组为 $a = [1,2,3,4]$, $p = [0,2,1,3]$ ，则 $a[p]=[0,3,2,4]$ 。然而，Python 中的 list 不支持这种访问，如果这样访问的话，会报告以下错误：

`TypeError: list indices must be integers or slices, not list`

本题解中使用了 numpy array 来实现以上这个 permutation 方式。

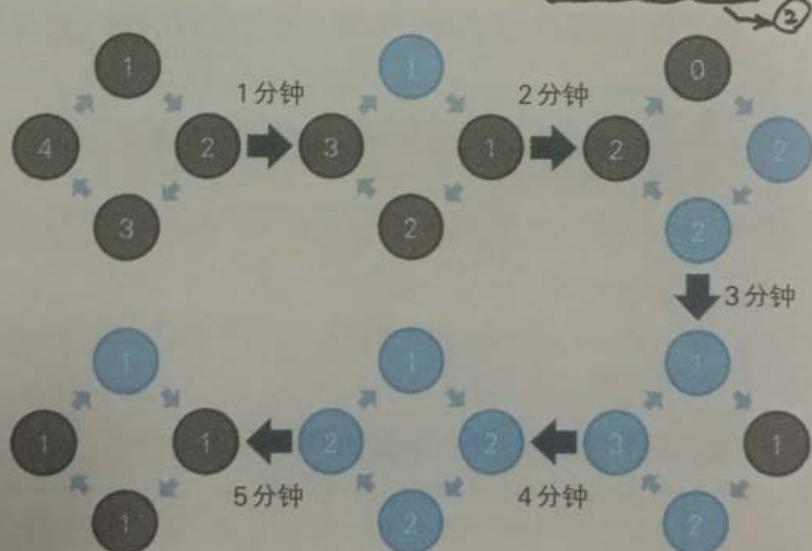
51. Q51: 同时结束的沙漏

51.1 问题描述

假设有 N 个沙漏，分别能计时 $1 \sim N$ 分钟。把这些沙漏围成一个圆，每隔 1 分钟倒挂（即上下颠倒）一次。然后，倒挂沙漏时的起始位置会“依次顺时针移动”，并且沙漏的倒挂个数会根据起始位置上的沙漏的计时分钟数不同而不同（计时 1 分钟时，倒挂 1 个沙漏；计时 2 分钟时，倒挂 2 个沙漏；计时 N 分钟时，倒挂 N 个沙漏）。
→ 应为“是 1 分钟沙漏时倒挂”

如果一开始所有沙漏的上半部分都装有沙子，那么倒挂时会出现所有沙漏的沙子同时向下落的情况。举个例子，当 $N = 4$ 时，按照 1 分钟、2 分钟、3 分钟、4 分钟的顺序排列沙漏，其起始位置为能计时 1 分钟沙漏的时候，如图 21 所示，经过 6 分钟后所有沙子会同时向下落（图 21 展示了 0~5 分钟后（倒挂后）沙漏中上半部分的沙量，而蓝色部分是倒挂着的沙漏。如下图 21 所示，1 分钟后所有沙子都会向下落）。

但是，如果以 2 分钟、4 分钟、3 分钟、1 分钟的顺序排列，当以 2 分钟的沙漏为起始位置时，无论经过多久都不可能出现所有沙子同时向下落的情况。
③



①、②、③、④ 应该翻译有误吧？ 图 21 当 $N = 4$ 时

问题

④

求当 $N = 8$ 时，使所有沙子同时向下落的 8 个沙漏的排列方法共有多少种（即便是同样的排列顺序，只要倒挂沙漏时的起始位置不同，就当作不同的情况计数）？

补充

所谓“即便是同样的排列顺序，只要倒挂沙漏时的起始位置不同”} 指的是下面这种情况。

不知所云：“按照 1 分钟、2 分钟、3 分钟、4 分钟的顺序排列，从 1 分钟的沙漏开始倒挂”以及“从 2 分钟的沙漏开始倒挂”这 2 种情况要分别处理 (计作 2 种排列方法)。又因为沙漏要围成一圈，所以“按照 1 分钟、2 分钟、3 分钟、4 分钟的顺序排列，从 1 分钟的沙漏开始倒挂”和“按照 4 分钟、1 分钟、2 分钟、3 分钟的顺序排列，从 1 分钟的沙漏开始倒挂”这两种情况可以看作是一种排列 (计作 1 种排列方法)。

感觉牛头不对马嘴

思路

51.2 原题的表述

首先，我认为这道题目存在严重的表述问题（是原文的问题还是翻译的问题呢？）。

题干部分多次出现“同时向下落”的说法，稍有常识就知道只要每个沙漏的上半部分都有沙子，那就不是“同时向下落”的情况吗。结合上下文猜测应该是说“沙子同时漏完”的意思。

第一段话的括号里的“计时 1 分钟时，倒挂一个沙漏；计时 2 分钟时，倒挂两个沙漏；计时 N 分钟时，倒挂 N 个沙漏；”也是显而易见的理解错误。难道即是 N+1 分钟时，倒挂 N+1 个沙漏吗？哪来的 N+1 个沙漏呢？猜测应该是说：当前倒挂操作的起始沙漏为 1 分钟沙漏时则倒挂一个，为 2 分钟沙漏时则倒挂两个。。。依此类推。

“补充”说明的第一段所说的跟第二段说的根本就不是一回事。第一段是说想解释倒挂沙漏起始位置不同看作是不同排列，而第二段解释了两种看上去不同的排列（由于圆的对称性的特性）其实是同一种排列，牛头不对马嘴。不过这个有可能不是翻译的问题，而是原文就有问题。

51.3 解题分析

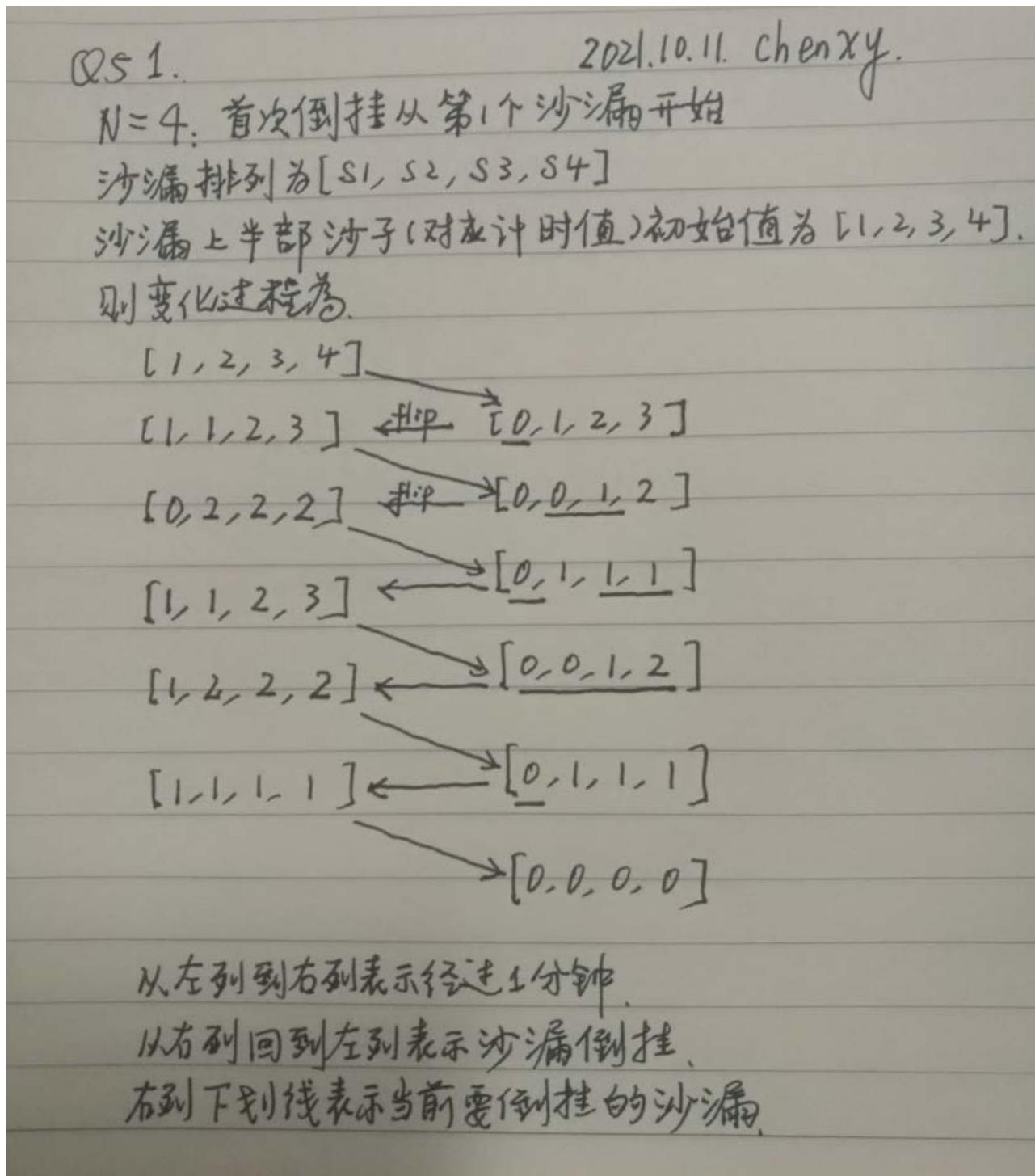
没有什么花哨（没想到什么花哨），唯有暴力破解。问题的焦点在于如何表示不同的排列状态以及如何处理沙漏翻转。

51.3.1 转换为线性排列

N 个沙漏，圆排列总共有 $(N - 1)!$ 种（与之相对，线性排列的场合是 $N!$ 种）。但是，对于每个圆排列，

第一次沙漏倒挂操作的起始位置共有 N 种（其后的沙漏倒挂操作的起始位置是按顺时针旋转），所以{圆排列，首次沙漏倒挂起始位置}组合起来的话就又回到 $N!$ 种了。所以可以把沙漏圆形排列还原成线性排列，只不过在线性排列中首次沙漏倒挂总是从排在首位的沙漏开始。只不过，要注意(1)沙漏倒挂起始位置是循环的，(2)连续倒挂多个沙漏时，存在跨越首尾边界的情况，即从尾部回到头部。

以下为一种情况的状态变化示例。



52. Q52: 糖果恶作剧

52.1 问题描述

假设有 N 种糖果，每种糖果有 M 颗。不同种类的糖果有不同颜色的糖纸和不同味道。同种糖果的糖纸可以区分，但是糖果本身无法区分。每颗糖果都用自己匹配的糖纸包裹叫做完全匹配；每颗糖果都用与自己不匹配的糖纸包裹叫做完全错配；介乎于两者之间的叫部分错配。

请问，当 $N=5$, $M=6$ 时有多少种完全错配的情况？

注意，即便同种糖果的糖纸也是可以区分，而同种糖果之间不可区分。比如说，‘苹果味的糖纸 1 包裹草莓味的糖果 1’与‘苹果味的糖纸 1 包裹草莓味的糖果 2’就不可区分算作相同的组合；‘苹果味的糖纸 1 包裹草莓味的糖果 1’与‘苹果味的糖纸 2 包裹草莓味的糖果 2’则由于糖纸不同因而是可以区分的。

52.2 解题分析

在 $M=1$ 时，这个问题实质上是一个“错排问题”。等价于“ n 个人互换礼物，每个人最终拿的都不是原本自己送出的礼物的组合方式一共多少种”的问题。

在 $M>1$ 时问题变得要复杂得多。考虑动态规划策略，先进行子问题分解。

考虑用 $candy$ 表示当前各种糖果尚未分配的数量的数组； $paper$ 表示各种糖纸尚余数量的数组。各数组的序号可以理解为各种糖果/糖纸的种类序号。以 $candy$ 和 $paper$ 联合表示当前分配状态，记为 $\{candy, paper\}$ 。比如说， $candy=[3, 3, 3]$ 表示共有 3 种糖果，且每种有 3 颗糖果未分配， $paper=[3, 2, 1]$ 则表示 0 号糖纸还有 3 张，1 号糖纸还有 2 张，2 号糖纸还有 1 张…

为了方便考虑（但并不失一般性），考虑接下来取 $candy$ 中尚未分配的糖果中种类序号最低的一枚糖果（可能有多枚，但是同种糖果不能区分所以任取一枚）进行分配，假设为 k 号糖果，那它可以分配给哪种糖纸呢？假定分配 j 号糖纸，首先不能是 k 号糖纸，即 $j \neq k$ ；其次，该 j 号糖纸必须还有未分配的，即 $paper[j] > 0$ 。做完这枚糖果分配后，分配状态需要如下更新： k 号糖果数要减一； j 号糖纸数也要减一。然后可以基于更新后的 $\{candy, paper\}$ 进行递归调用（solving the sub-problem）。

当前状态 $\{candy, paper\}$ 下的分配数等于对所有满足条件的 j 的子问题解之和。由此可得处理流程（伪代码）如下：

date: 2021.8.26. 假设有 N 种糖果，每种为 M 个 chenxy

`explore(candy, paper)`. Candy 和 paper 均为长度为 N 的数组。
 if candy 变为全零:
 return 1.
 Candy[K] 表示糖果#K 还未安排的个数
 paper[K] 表示 K 号糖纸还剩下(或可用)的
 张数.
 (*) 表示全部安排完毕.

~~(接下来搜索序号最低的尚未发放的糖果)~~

for k in range(len(candy)).
 if candy[k] > 0:
 break
 (接下来把 k 号糖果放置到任意空的 k 号糖纸中去)

sum = 0
 for j in range(len(candy)).
 if j != k && paper[j] > 0: (不同于 k 且仍有剩下的糖纸)
 candy[k] -= 1
 paper[j] -= 1
 sum += explore(candy, paper)
 candy[k] += 1, paper[j] += 1. (复原现场, 准备下一个 explore)

Candy = N * [M]
 paper = N * [M] 即 $\underbrace{[M, M, \dots, M]}_N$ } 初始化 candy 和 paper
 return explore(candy, paper)

但是以上流程中尚未考虑 memoization, 实现时需要追加进去, 否则的话运行时间会变得无法承受。

【吐槽】老实讲这道题断断续续想了两三天理不清, 原书没有解说直接上来一段代码, 看半天也没看懂, 不是因为 Ruby 语言的问题, 而是确实没有看懂他的思路(廉颇老矣。。。但是不服气)。还是得以自己能够理解的方式想清楚、用自己的语言能解说清楚、并用代码实现了才能有最大的收获。从这个意义上来说, 原书解说匮乏以及代码看不懂倒是一件好事, 逼着自己只能硬着头皮干。。。

53. Q53: 同数包夹

53.1 问题描述

有分别标了数字 $1 \sim n$ 的两副牌，共 $2n$ 张。把这些牌排成一排，确保两张 1 的中间夹一张牌，两张 2 的中间夹 2 张牌，……，两张 n 的中间夹 n 张牌。

请问，当 $n=11$ 时，有多少种排列方法？

53.2 分析

考虑有 $2n$ 个位置，按顺序地且按照以上规则的约束，安放每个数字的两张牌，直到 $2n$ 张牌将 $2n$ 个位置占满。这样的话这个问题可以看成（转换成）图搜索问题中的路径遍历问题。即从“空”状态开始到达“满”状态的所有不同路径的数量。这可以通过深度优先搜索来解决。

但是要注意的是，这里的终止状态并不是唯一的，所有的“满”状态都是终止状态。所以这不是从图中的某个起始节点到另一终止节点之间的路径遍历，而是遍历搜索从“空”状态节点到达所有合法的“满”状态节点的可能路径。

深度优先搜索的要点如下所述。

53.2.1 节点（状态）表示

以长度 $2n$ 的数组表示，“空”状态指数组中的元素为全“0”。“满”状态指数组中的元素全部为非零的状态，但是并非所有的“满”状态都是合法的状态。但是在本题中单独的状态还不能完全表示一个节点，还要加上接下来要插入的数据才能唯一地表示节点信息，即 $\{state, m\}$ 。

53.2.2 邻节点搜索

基于当前状态 $curState$ 和接下来要插入的数据 m ，可以按如下方式确定邻节点：从数组头部开始搜索，搜索到两个相隔 m 个位置都为空的时候，即可认为找到一个合法的邻节点 $\{newState, m+1\}$ ， $newState$ 为将 m 插入前述两个位置，而接下来要插入的数也相应地更新为 $m+1$ 。

53.3 递归实现：从小到大插入

运行结果：

```
N=2, ans = 0, tCost = 0.0(sec)
N=3, ans = 2, tCost = 0.0(sec)
N=4, ans = 2, tCost = 0.0(sec)
N=5, ans = 0, tCost = 0.0(sec)
N=6, ans = 0, tCost = 0.0010356903076171875(sec)
N=7, ans = 52, tCost = 0.0029900074005126953(sec)
N=8, ans = 300, tCost = 0.02193927764892578(sec)
N=9, ans = 0, tCost = 0.1495983600616455(sec)
N=10, ans = 0, tCost = 1.1013171672821045(sec)
N=11, ans = 35584, tCost = 8.87007999420166(sec)
```

令人意外的是， $N=5, 6, 9, 10$ 居然没有合法的安排方式！是程序的 bug 吗（ $N=11$ 的结果与原书的结果一致，所以应该概率比较低）？**如果是正确的运行结果的话，那么有没有可能从数学上证明满足什么条件才能有合法的安排方式呢？**

另外，运行时间略长，需要进一步优化。

53.4 递归实现：从大到小插入

53.5 递归实现：二进制存储

53.6 非递归实现

54. Q54: 偷懒的算盘

54.1 问题描述

算盘在国外也流传甚广。这里我们用算盘进行加法运算。假设要求 1~10 的和，那么简单地说，按顺序计算 “ $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$ ” 的和就可以得到答案。

用算盘计算时，请注意移动的算珠的个数。比如计算 “ $8 + 9$ ” 时，首先要移动 “个位的 4 个算珠”，然后移动 “十位的 1 个算珠” 和 “个位的 1 个算珠”。也就是说，计算 “ $8 + 9$ ” 时一共需要移动 6 个算珠（图 24）。

而 “ $9 + 8$ ” 的时候则是首先移动 “个位的 5 个算珠”，然后移动 “十位的 1 个算珠” 和 “个位的 2 个算珠”，一共移动 8 个算珠（图 25）。

如上所述，计算顺序不同，珠算的时候要移动的算珠个数也不同。

问题

求 1~10 的和时，使移动的算珠个数最少的计算顺序是什么样的呢？此时要移动的算珠个数是多少呢？

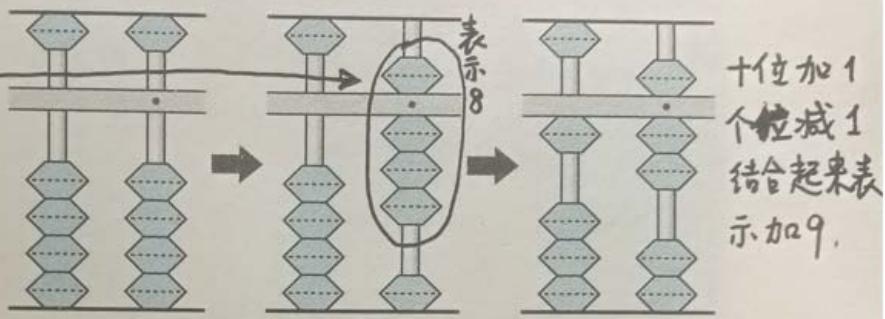


图 24 计算 $8 + 9$ 时

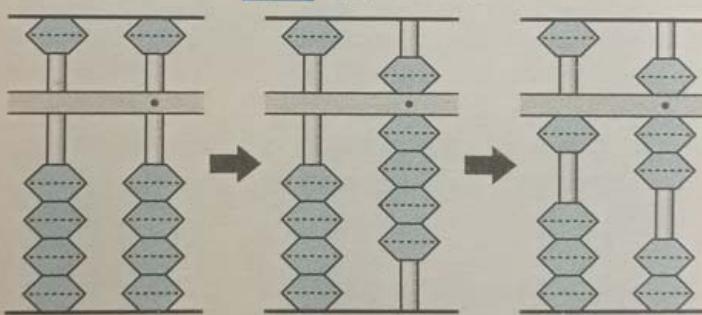


图 25 计算 $9 + 8$ 时

54.2 解题分析

关键点在于把算盘上算珠的摆放位置看作是数字的一种表达方式。算盘分上下段，在每个数位上，

上段一个算珠表示 5，下段一个算珠表示 1。因此每个数位上的数在算盘上的表示可以用两个数字来表示： $(x//5, x\%5)$, for $x=0,1,2, \dots, 9$. 第 1 个数表示上段的表示，第 2 个数表示下段的表示。比如说 8，用上段一个算珠表示 5，用下段 3 个算珠表示 3. (参见上面的图例)。

本题要求计算的是 1 到 10 的和，总和只有 55，所以只需要考虑两位数，因此总共可以由 4 个数字来表示它在算盘上的表示： $(x//50, (x\%50)//10, (x\%10)//5, (x\%10)\%5)$.

基于以上考虑，进行一次加法所需要的算珠移动次数就是比较加法执行前后算盘上的两个数的表示的差分，即比较 4 个位置上的算珠个数的差异，对这些差异进行求和即得所需要移动的算珠的总次数。

这里的关键是不要被神秘的算珠划拉的动作所迷惑。。。我刚看到这道题目时脑海中模糊显现的就是手指在算盘上灵活而诡异的划拉动作。不用关心算珠划拉的过程，只要关心算珠起始位置和最终位置即可！

54.3 代码及测试

太慢了！总共有 $10! = 3628800$ 种排列，所以暴力搜索需要非常长的时间。需要考虑优化策略。比如说动态规划策略。。。一时还没有想清楚。。。且听下回分解^-^

54.4 递推关系

由于 10 个数每个只用计算一次，考虑已经执行了若干个数的运算（它们构成 `visited` 集合），此时算盘上的计算结果为 `curSum`，完成之后剩余的运算所需要的最少算珠移动数与之前若干个数的运算顺序（以及相应的算珠移动数）是无关的，仅与 `curSum` 有关。由此可以得到本问题的子问题分解结构，如下所述。

令 $f(\text{unvisited})$ 表示（在当前已经求得的和—即 `visited` 中的数的和—的基础上）执行剩余的 `unvisited` 的数的运算所需要的最少算珠移动数。注意，`unvisited` 与 `visited` 是互补的，或者说一一对应的。则有以下递推关系成立：

$$f(\text{unvisited}) = \min_{k \in \text{unvisited}} (f(\text{unvisited} - k))$$

其中，`unvisited` 表示一个集合，“`unvisited-k`”则表示从 `unvisited` 中删除一个元素 `k` 的操作。

基于以上递推关系，可以以正向的标准的动态规划的方式实现，也可以以递归加 Memoization 的方式实现，鉴于后者代码显得更加简洁，以下代码采用后者。

因为 10 个数每个只能用一次，所以实际上 `visited` 或者 `unvisited`（以下代码中是用 `visited`）可以用 10 比特的二进制数来表示。

54.5 代码及测试 2

以上包含两种同为“Recursion+Memoization”策略的实现方式，后一种又比前一种还要再快 4 倍。相比原始的暴力破解方案则快了三个数量级。

search1() 与 search2() 为什么会有 4 倍的速度差异呢？

Search1() 将到目前 visited 为止的算珠移动次数通过函数接口传递，然后在到达目标（即所有数都运算完的状态）进行最小算珠移动次数判断，并且以 {visited, moves} 作为状态表示用于 memo 记忆。而 Search2() 则只计算从 visited 往后的最小算珠移动次数，因此只需要基于 visited 进行 memo 记忆。

很显然，search1() 所需要考虑的 {visited, moves} 所表示的状态数会远远大于 search2() 的仅由 visited 所表示的状态数。或许这就是两者运算速度相差 4 倍的原因吧。

55. Q55: 平分蛋糕

55.1 问题描述

假设要公平地分蛋糕。不过单纯地对半分有点无聊，所以我们将采用以下切法。

蛋糕是 $m \times n$ 的长方形，“初始状态”如图 26 所示，可以根据 1×1 的方格沿着直线切蛋糕。因为是直线切分，所以每次切分一定会切分为两半。

这里，两人轮流切分蛋糕，切蛋糕的人吃掉两半中小的一块，而剩下的蛋糕由另一个人切，切完也是吃小的一块，然后重复这个过程。如果切分出来的是两块同样大小的蛋糕，则吃其中任意一块。最后一块不切，直接由下一个切蛋糕的人吃掉。

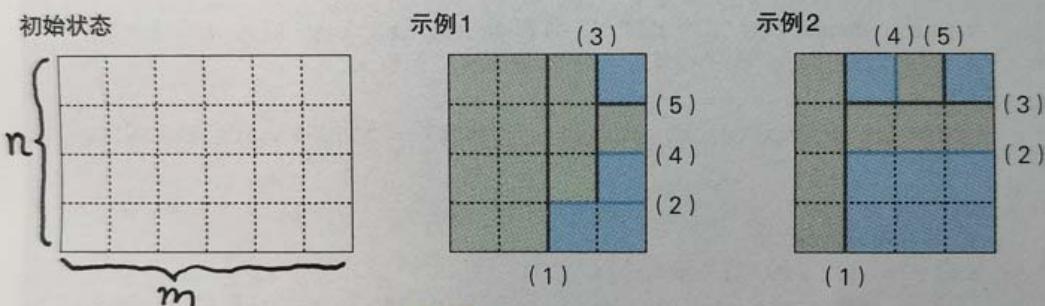


图 26 蛋糕切分示例

举个例子， 4×4 的正方形蛋糕可以用示例 1 和示例 2 的切法。这里我们思考一下使两人最终吃掉的蛋糕的量相同的切分方法。据图可知，采用示例 1 这种切法时，灰色部分明显比较多，而示例 2 的切法才能使两人最终吃掉同样多的蛋糕。

问题

$$\begin{matrix} m & n \\ \swarrow & \searrow \end{matrix}$$

如果是 16×12 的长方形蛋糕，那么当有一种切法使两人吃掉的蛋糕同样多时，切掉的蛋糕的长度是多少（上图示例 2 中，(1) 是 4 格，(2) 是 3 格，(3) 是 3 格，(4) 是 4 格，(5) 是 1 格，因此切掉的蛋糕共 12 格）？

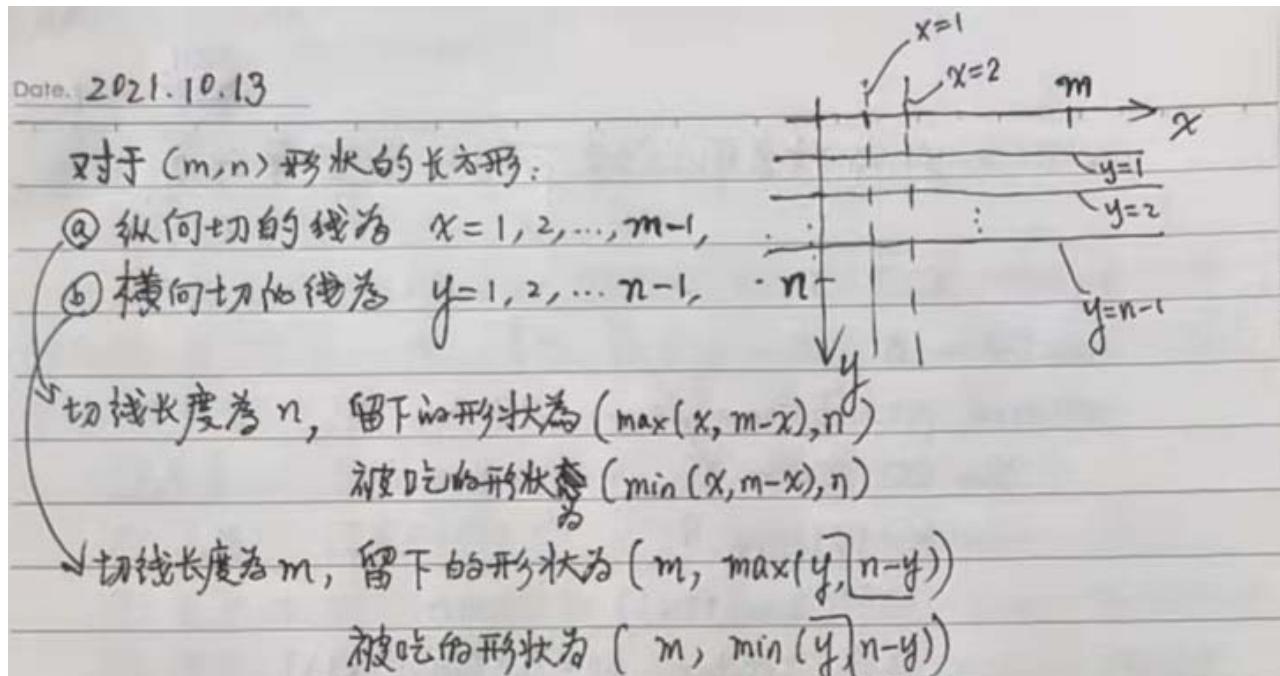
条件

能使两人吃掉的蛋糕同样多的切法有很多种，这里规定求其中切掉部分的长度最短的一种切法。

55.2 解题分析

这个题目第一感就是动态规划。

对于 (m, n) 形状 (如下图所示, m 代表横向的长度, n 代表纵向的高度)。为了方便起见, 参照以下图右上角这样的坐标说明。



每切一刀并吃掉小的那一块后留下的仍然是一个长方形。

但是由于需要确保从头到尾两个人分到的蛋糕一样多, 所以子问题并不单纯由长方形的尺寸决定。还要考虑进去当前轮到谁切, 以及两个各自已经分到的蛋糕大小。

55.2.1 初始算法流程

考虑“递归+Memoization”的实现方式, 得到算法流程如下所示。

```

memo = dict()
BIGINT = 10000 # 足够大即可

def search(m, n, who, eat):
    if m == 1 and n == 1:
        eat[who] = eat[who] + 1
        if eat[0] == eat[1]: # 满足条件的终态
            return 0
    else:
        return BIGINT # 不满足条件的终态, 返回很大的数会在择小比较中淘汰
        # 这个也要存放到 memo 中去。
    if (m, n, who, eat[0], eat[1]) in memo:
        return memo[(m, n, who, eat[0], eat[1])]

minlen = BIGINT
for x = 1, 2, ..., m-1: # 纵向切
    eat[who] += n * min(x, m-x)
    curlen = n + search(max(x, m-x), n, 1-who, eat)
    minlen = min(minlen, curlen)

for y = 1, 2, ..., n-1: # 横向切
    eat[who] += m * min(y, n-y)
    curlen = m + search(m, max(y, n-y), 1-who, eat)
    minlen = min(minlen, curlen)
    memo[(m, n, who, eat[0], eat[1])] = minlen

```

初始状态为 $\text{who}=0, \text{eat}=[0,0]$, 因此调用 $\text{search}(M,N,0,0,0)$ 即可。

55.2.2 优化

进一步考虑优化。

【优化 1】

由于总是吃掉较小的那一块。

因此, 比如说纵向切的时候, $x=1$ 与 $x=(m-1)$ 其实是等价的。横向切的情况也同理。这样切分的次数可以大约减小一半。

【优化 2】

最终只关心两者分到的是不是相同，并不需要关心各自分分到的绝对量（当然在本题中两者平分的话所分的量也是确定的），因此可以用一个变量 diff 来去掉 eat[2]。在跟踪统计 diff 时，第 1 个人的份加进去，第 2 个人的份则减掉，即可。

进一步，考虑每一步分掉的量为 z，每一步用($diff = z - diff$)的方式跟踪的话，连 who(表示当前轮到谁切分)都可以省掉了。这是原题解给出的技巧，确实小巧精致！

【优化 3】

由于对称性，(m,n)与(n,m)两种形状其实也是等价的，由此可以进一步减少需要计算的子问题个数。

考虑以上优化策略后，得到以下代码。

55.3 代码及测试

程序运行的速度倒是足够快了，但是熟悉的尴尬又重现了。原书给的答案是 46，以上结果是 47。有些结果一看就是十万八千里的离谱，那就知道肯定根本的机制有问题，这种问题通常反而容易解决。本题这样差 1。。。这是一个根本性的问题呢，还是一个小的纰漏呢？很难判断。这种问题可能反而很难解决。

之所以总是敢厚着脸皮把不完全正确的题解贴出来，是因为我认为从纠错中学习是一种重要的学习路径。即便是出题者也不一定是每道题一上来就是漂亮的正解，可能也需要经过纠错、优化的迭代才能得到最终呈现在读者面前的漂亮的正解（书籍出版由于容量的约束不可能把所有的过程都呈现出来）。

老问题：谁能帮我指出错在哪儿呢？

56. Q56: 鬼脚图中的横线

56.1 问题描述

假设我们要在鬼脚图^①中划横线使上方和下方相同的数字相连。但是，这里要根据给定的上下数字用最少的横线连接来形成鬼脚图。
横线只能连接相邻两条竖线，不能越过几条竖线连接。

例) 上方数字：1、2、3、4
下方数字：3、4、2、1

上面这个示例对应的就是图1中的排列方式1。这时最少的横线条数是5条。同样地，最少需要5条横线的还有排列方式2里的“4、3、1、2”这个排列。排列方式3也是用5条横线连接的，但也可以像右侧这样只用3条横线连接。也就是说，当下方数字的排列为“3、2、1、4”时，需要的最少横线条数就不是5了。

问题

② 从上方数字到下方同一数字的路径只能以阶梯形
向下走。③ 不同路径可以共享横线。④ 碰到横线
时必须顺着横线走。⑤ 左右可以折返。
给定7个数字（即竖线有7条）时，最少需要10条横线的下方数字的排列着横线走
方式共有多少种（这里规定，即使可以在不同位置使用横线连接，但是只要下方到隔壁的
数字的排列顺序一致，就算同一种排列方式）？

上方数字 1, 2, 3, 4, 5, 6, 7

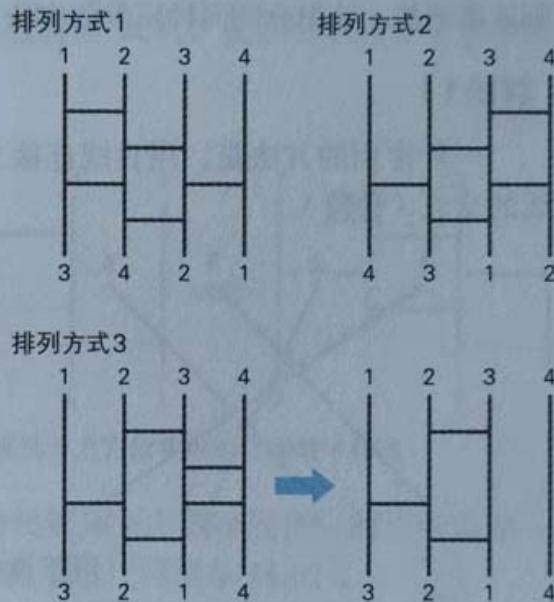


图1 鬼脚图示例

① 起源于日本室町时代（1336年—1573年）的游戏，常用于抽签等。玩法是先在纸上画平行的竖线，竖线条数与参加抽签的人数相等。以竖线一端为起点，另一端为终点。在起点处空出写人名的地方，并在终点处写上抽签的项目。接下来在相邻的竖线之间任意划横线，但横线不得跨越两条以上的竖线。为确保公平，其他参加者也可自由添加横线，但此时需将终点处折叠，隐藏抽签的项目。最后每人选一个起点（注意不能重复选）开始往下走，有横线时必须转弯，即顺着横线走到隔壁的横线。最后到达的终点即自己抽中的项目。——编者注

竖

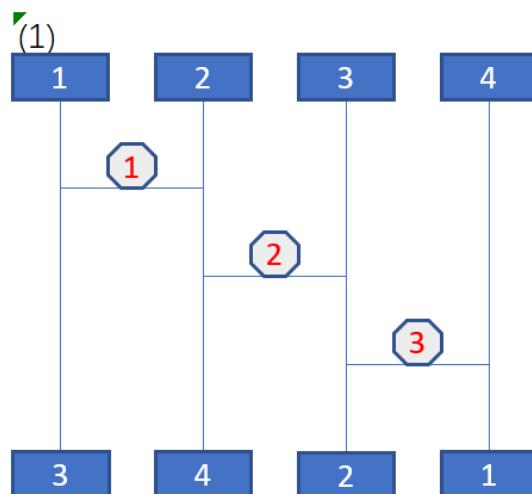
56.2 思路 1—贪心策略

感觉非常没有头绪。先做一些实例计算分析。

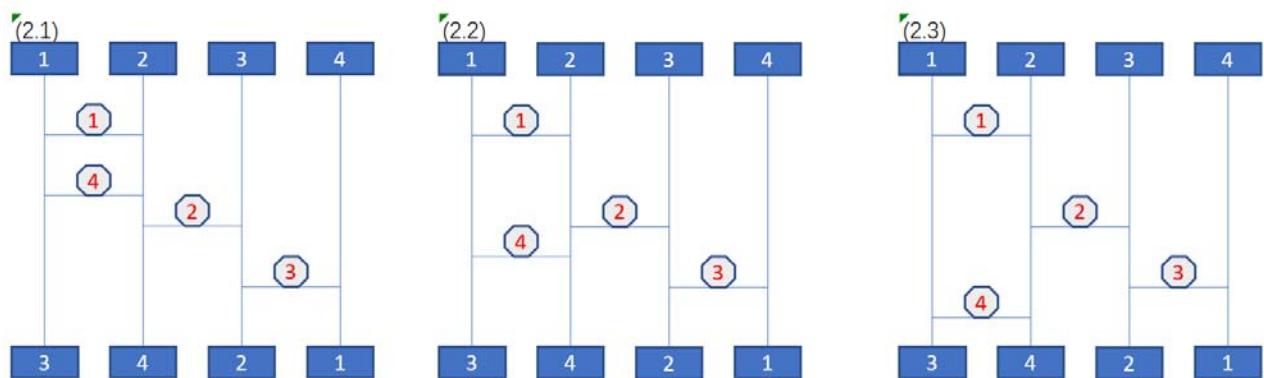
考虑图 1 的 $\{1234 \rightarrow 3421\}$ 的例子。为了说明方便，以 U* 表示上边的数字，D* 表示下边的数字。以“纵 1”表示第一根纵线，余类推。以“横*”表示所添加的横线，横线上的数字(图中 8 边形中的数字)表示横线添加的序号。

假定按照类似于贪心算法的方式来绘制鬼脚图。

比如说 U1 要到达 D1，必须横跨中间两根纵线，因此可以画出对应的横线如下图所示：

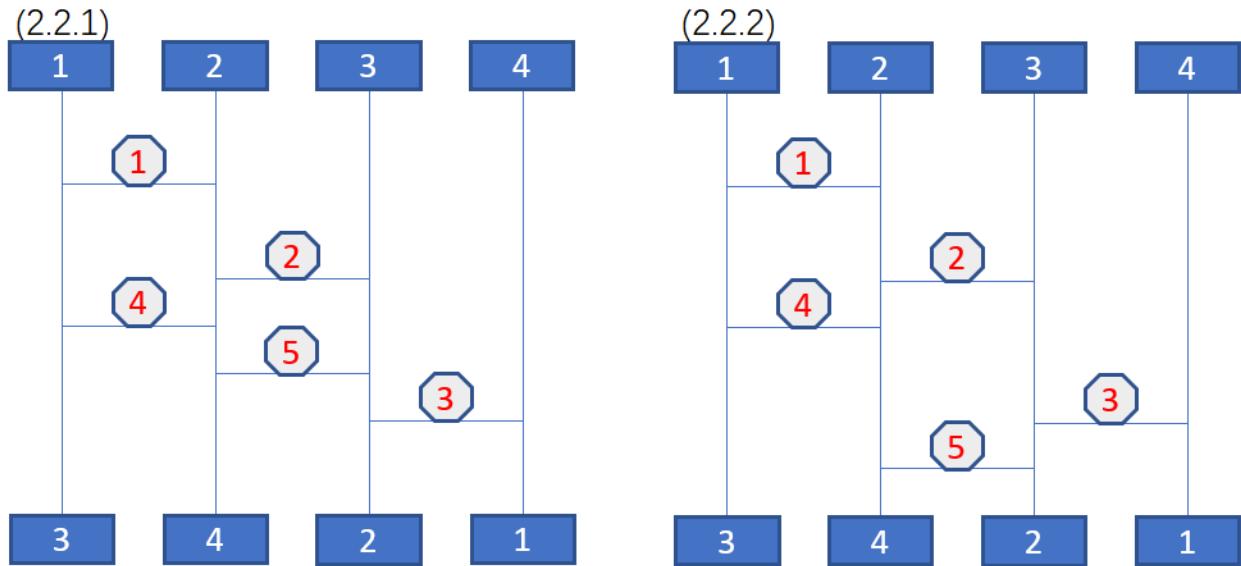


接下来考虑 $U2 \rightarrow D2$. $U2$ 往下走时先碰到横 1 横跨到纵 1，然后肯定必须再加一根横线回到纵 2。但是这根横线添加的（相对于其它已经添加的横 2、3 的）纵向相对位置是与后续动作相关的，所以必须分情况考虑。显然这里有三种可能的情况，如下所示：



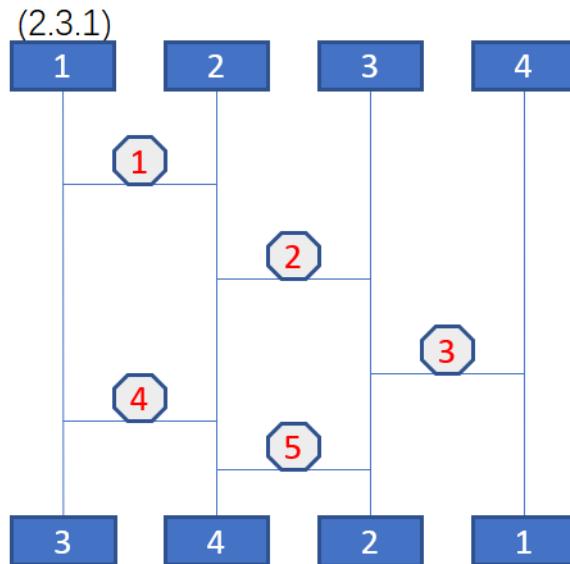
上图中的(2.1)由于横 4 的加入导致原来的 U1 到 D1 的路径发生了变化，所以予以排除（注 1：暂时按照这种规则进行）。

图(2.2)中 $U2$ 经过横 1、横 4 后回到纵 2，然后需要再添加一条横线到达 $D2$ ，同样有相对于横 3 的上下相对位置的两种选择。但是如果添加到横 3 上面（横 4 与横 3 之间）的话，会导致原 $U1$ 到 $D1$ 的路径发生变化所以排除(2.2.1)。因此只有一种选择，如下图所示 (2.2.2)：



接下来继续考虑(2.2.2)，我们会发现在(2.2.2)中 U3 和 U4 已经可以借助既有横线到达目的点了：U3-横 2-横 4；U4-横 3-横 5。因此我们得到{1234→3421}的一个鬼脚图解。

接下来再回头考虑(2.3)的继续。U2 要到到达 D2 还要追加一条线，由于横 4 已经比横 3 低了，所以追加横 5 只有一种可能。如下图所示：



然后我们同样发现，在(2.3.1)中 U3 和 U4 已经可以借助既有横线到达目的点了：U3-横 2-横 4；U4-横 3-横 5。因此我们得到{1234→3421}的第二个鬼脚图解。

以上我们基于类似于贪心算法的思路得到了{1234→3421}的两个鬼脚图解。但是这里存在两个问题：

- (1) 如何确保，或者说如何证明，以上贪心算法总能得到最优解？
- (2) 以上鬼脚图绘制规则由人手动来做的话似乎是一目了然的，但是如何让计算机按照这个规则来搜索呢，或者说如何编程呢？

试图总结以上鬼脚图绘制贪心策略如下：

(R1) 按顺序考虑 U_1, U_2, \dots 的到达对应 D^* 的路径

(R2) 考虑 U_k 的路径时不得破坏 $U_i (i < k)$ 的已经确认好的路径

(R3) 针对 U_k 的路径规划：

a) $V = k$; #表示 U_k 的初始位置在纵 k 上

b) 沿 V 下行，

i. 如果碰到横线，沿横线跨越到对应纵线 ($V=V-1$ or $V=V+1$)

ii. 如果从跨越到 V 的点往下没有横线，

1. 如果 V 即目的纵线，针对 U_k 的路径规划结束，退出

2. 如果 V 非目的纵线，根据当前所在纵线与目标纵线的相对位置绘制相应的横线（目标纵线在左侧则向左画横线，反之向右画横线）。绘制横线时要考虑与（自本纵线上一个横线立点以下的）既有的其它横线的所有可能的（上下）相对位置，但是如果碰到破坏规则(R2)的话则放弃

以上可以看作是一个基于贪心策略的深度优先搜索问题。虽然前进了一步，但是依然没有解决前面提及的两个问题：如何证明最优化或非最优化？如何编程？甚至，以上规则难以显而易见地判断是正确的。。。不过作为一种思路记录下来总归是有价值的

56.3 思路 2

仍然考虑图 1 的 $\{1234 \rightarrow 3421\}$ 的例子。

在之前的讨论中我们已经得到了 $\{1234 \rightarrow 3421\}$ 的两个鬼脚图连线方式，如下图所示的(2.2.2)和(2.3.1)。

56.3.1 从鬼脚图出发进行变形

考虑将横线缩为一个点进行变形（类似于拓扑学中的那种变形）会得到什么呢？如下图所示：

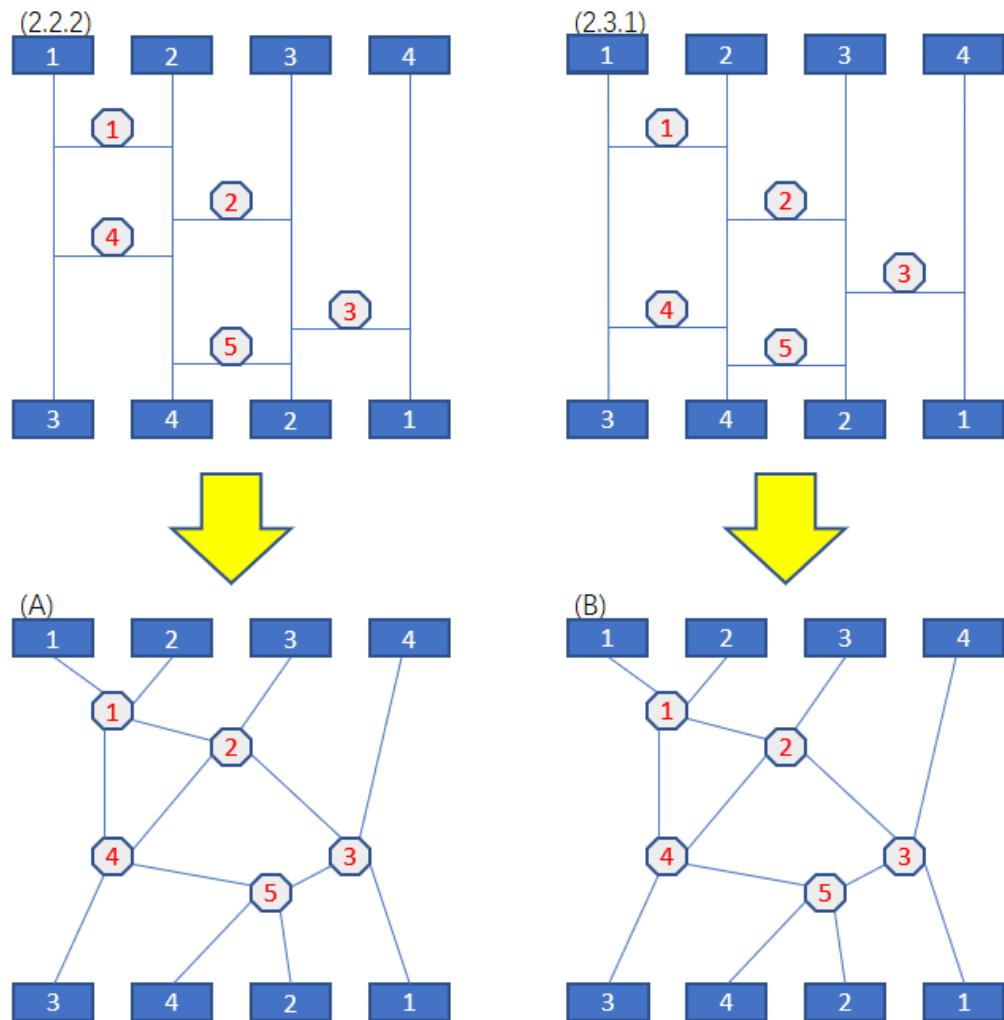


图 1

进一步将线扯扯直，让每一对 $U_k—D_k$ 的路径变成直线，我们会发现以上图(A)和图(B)会得到相同的图，如下图所示（先扯直 $U_1-D_1, U_3-D_3, U_4-D_4$ ，得到(C)，然后再挪动以下位置将 U_2-D_2 也扯直得到(D)，以下我们称这种图为交叉图）：

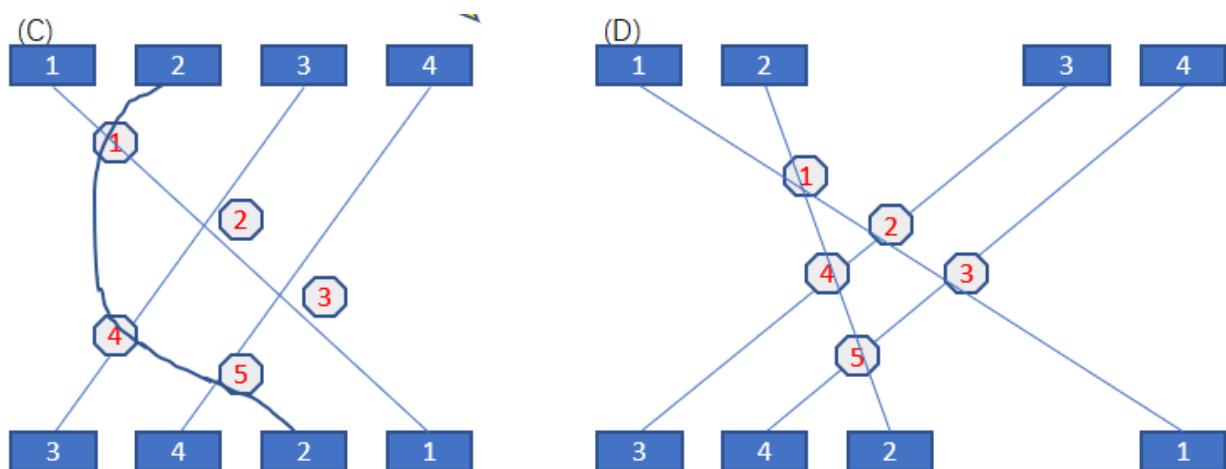


图 2

这个意味着什么呢？是不是意味着原问题可以转变为“上下两排之间对应数字之间连线，求最小的交叉点的个数”的问题呢？

56.3.2 反向变换

注意，从以上图(D)的交叉图出发遵照前述鬼脚图的规则是可以恢复出原鬼脚图的（恢复的方法参见以下说明）。下面我们从一个一个与图(D)略微不同的交叉图(E)（这种不同仅仅在于上下两排点的相对间距不同导致直线连接后的交叉点的位置不同而已）出发看看反向能恢复出什么来。

首先从 U1 (记得前面我们提过 U 表示 Upper, D 表示 Down, Uk 表示上边的 k 节点, Dk 表示下边的 k 节点) 出发，因为 U1 到 D1 经过(1), (2), (3)三个交叉点，我们知道每个交叉点对应一条横线，因此可以先画出三条横线，而且横 2 位置在横 1 之下，横 3 在横 2 之下。

接下来，考虑 U2 的路径。U2 的路径先碰到交叉点(4)，对应鬼脚图上横 4 的线，但是横 4 是应该往左画还是往右画呢？首先横 4 必然是在横 2 上面（如果不是的话，U2 出发后就应该先碰到交叉点(2)）；其次，如果往左画的话，要么在横 1 上面，要么在横 1 下面。在横 1 下面画不行，因为这样破坏了 U1 的路径。在横 1 上面画也不行，因为这样的话，U2 在(4)之后就应该是碰到(1)而不是如左图中的(2)。所以，横 4 只能是往右画，而且必须在横 1 上面。然后，U2 在(4)之后碰到(5)，这对应着在右图需要追加一条新的横 5，显然它只能往右画（因为 D2 在右边）而且必然在横 3 下边（不然的话 U2 在(4)之后就会碰到(3)）。

接下来，考虑 U3 的路径。此时图上已经有了横 1, 2, 3, 4, 5。很显然 U3 经过(4)和(1)可以到达 D3，这个在作图和右图都成立。因此不需要追加新的横线。

接下来，考虑 U4 的路径。左图中 U4 先碰到(3)然后是(5)，右图上也已经存在这一条路径了。

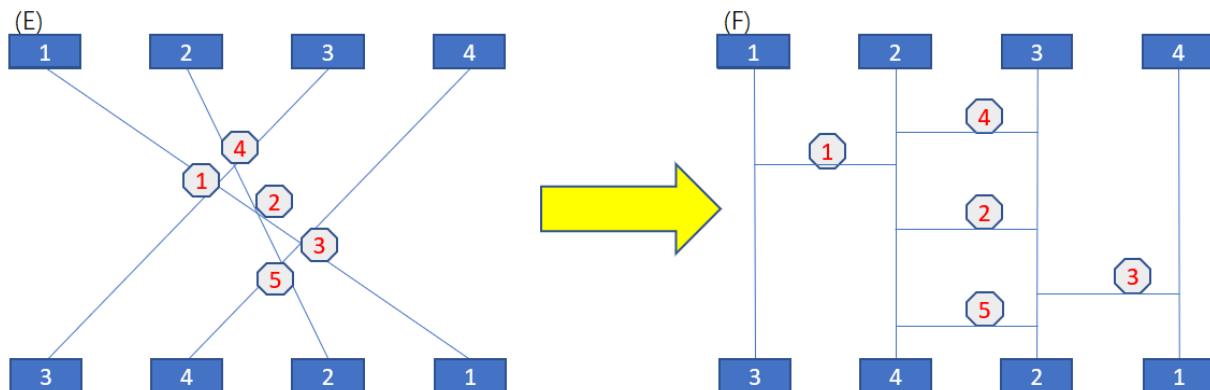


图 3

好了，大功告成，我们的确从一个新的交叉图恢复出了对应的鬼脚图。而且，surprisingly，这个鬼脚图与我们之前的讨论所得到(2.2.2)和(2.3.1)都不同，我们得到了一个新的鬼脚图！

56.3.3 进一步探索

如上所述，图(D)和图(E)的差异仅在于上下两排节点的间距稍微不同而已。进一步，显而易见的是，不管从图(D)出发还是从图(E)出发，总可以使多个交叉出现重叠，这种情况意味着什么呢？还能不能恢复出对应的鬼脚图，重叠的交叉点算多个点还是算一个点？如果算一个点的话，是不是意味着出现了横线更少的鬼脚图呢？

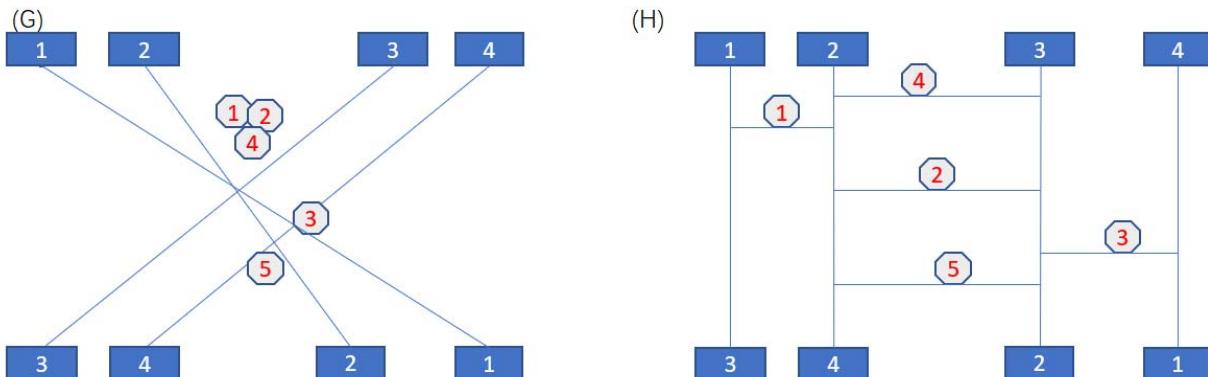


图 4

如上图所示，在图(G)中，经过调节上下两排点的相对间隔后，交叉点(1)(2)(4)重叠了。让我们看看基于这个图能恢复出什么来？

首先从 U1 出发，因为 U1 到 D1 经过(1-2-4)（代表上图中三条线公共交叉点）和(3)。我们知道每个交叉点对应一条横线，但是如果把(1-2-4)看作一条横线的话，那么在 U1 和 D1 之间只有两条横线，而在鬼脚图中 U1 和 D1 之间隔着 3 格，不可能只经过两条横线到达。由此我们知道多重交叉点必须看作是多个交叉点，不能看作是一个。因此我们仍然把(1-2-4)看作是(1)和(2)和(4)（编号是无所谓的），那么 U1 到 D1 的路径可以是(1)-(2)-(3)也可以是(1)-(4)-(3)，由于编号是无所谓的，不失一般性，我们就仍然考虑是(1)-(2)-(3)。这样的话我们仍然可以先画出三条横线，而且横 2 位置在横 1 之下，横 3 在横 2 之下。(这个与上一节相同)

接下来，考虑 U2 的路径。此时图中已经有了横 1、横 2 和横 3。由于是三个重叠的交叉点，U2 的路径是算碰到(1)和(2)和(4)中的哪个呢？显然(2)可以排除，因为横 2 在横 1 下面，U2 不可能在遇到横 1 之前先碰到横 2。

考虑 U2 先碰到的是交叉点(4)，对应鬼脚图上横 4。那么必然有横 4 在横 1 上面。但是横 4 是应该往左画还是往右画呢？首先横 4 必然是在横 2 上面（如果不是的话，U2 出发后就应该先碰到交叉点(2)）；其次，如果往左画的画，要么在横 1 上面，要么在横 1 下面。在横 1 下面画不行，因为这样破坏了 U1 的路径（与上一个例子的道理相同）。在横 1 上面画呢也不行，因为这样 U1 就会先碰到横 4 因而破坏了 U1 的路径。结论是横 4 不能往左画，只能是往右画，而且必须在横 1 上面。然后，U2 在(4)之后碰到(5)，这对应着在右图需要追加一条新的横 5，显然它只能往右画（因为 D2 在右边）而且必然在横 3 下边（不然的话 U2 在(4)之后就会碰到(3)）。

接下来，考虑 U3、U4 的路径就和上一节的例子一样了。由此得到了图(H)，其实和图(F)是相同的。

本节的讨论得到的结论是，多重交叉点必须作为多个不同的交叉点来处理。

56.3.4 小结

今天的讨论感觉前进了一大步，问题似乎转变成了一个更容易进行算法处理的问题。不过我仍然没有完全想清楚这种对应意味着什么，目前也没有能力给出稍微严谨一点的数学证明。暂且贴出来看看有没有高手能够指点一下^-^.

56.3.5 实现

虽然前面所提出的问题没有解决，但是我猜测这里所做的问题转换是正确的，姑且先不管它在数学意义上的严谨证明，先把它实现了再说了。

如前所述，问题转变成了上面一排数字和下面一排数字（相同数字之间）连线后总的交叉点的个数的问题。

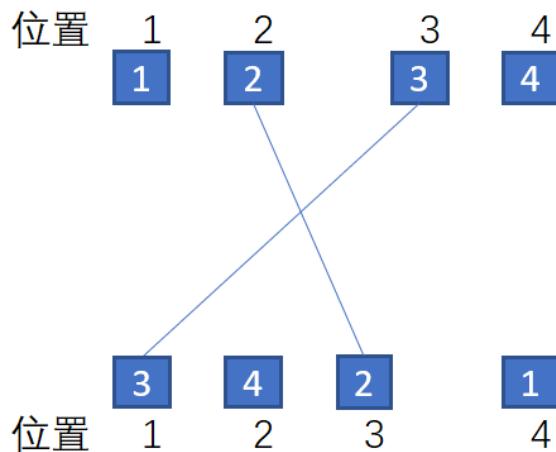
考虑上面一排中数字 $\{1,2,3,\dots,N\}$ 的位置是 (x_1,x_2,\dots,x_N) —当然由于上面这一排是初始排列所以有 $x_k=k$ ；下面一排数字的位置是 (y_1,y_2,\dots,y_N) ，很显然， (y_1,y_2,\dots,y_N) 也是 $\{1,2,3,\dots,N\}$ 的一个排列。在 python 中可以用 `itertools.permutations()` 来生成所有 N 个数的排列。

针对下面一排的每一种可能的排列，将对应数字相同的两点连接起来，得到 N 根连线，每根连线用 (x_i,y_i) 表示， x_i 表示线段一端在上面一排的位置， y_i 表示线段一端在下面一排的位置。

接下来就是判断两根线段是否相交的问题了。有一个很简单的判断方法，如果两根线段的一段在上面一排中的相对位置与它们的另一端在下面一排中的相对位置不同的话，即表示它们相交。以数学的方式表示就是：

$$(x_i - x_j)(y_i - y_j) < 0 \Leftrightarrow (x_i, y_i) \text{ intersects with } (x_j, y_j)$$

示意图如下，图中两根线段的位置坐标分别为 $(2,3)$ 和 $(3,1)$ ，它们在图中也的确是相交的，它们的坐标满足上式的关系。



据此可以得到本题的算法流程：

遍历(1,2,...,N)的所有排列，for each pItem:

列出对应当前 pItem 的上下两排中对应数字的连线 → lines

For i = 0,1,2, ... , len(lines)-2

For j = I+1, I+2, ... , len(lines)-1

判断 lines(i) 和 lines(j) 是否相交，如果相交则交叉点数统计 num_crosspoints 加一

如果 num_crosspoints 等于 10，则满足条件的排列的统计次数加一

56.4 代码实现

运行结果： N=7, cnt_geq_10=573, tCost = 0.051(sec)

56.5 后记

这道题目拖了很长的时间，而且问题并没有彻底解决。

做完了以上代码实现后，翻了翻原书，看看原书的解释。然而。。。很失望。

如下图所示为原书给出的解法 1，高兴的是我给出解决方案不谋而合。失望的是完全没有解决我的疑问。书中给出的解释类似于数学证明中臭名昭著的“显而易见。。。易证。。。”，只不过，这个真的有这么显而易见吗？看到这种解释总有一种智商被鄙视的挫败感^-^

(解法1)

一种常用的方法是，用直线连接上方和下方的相同数字，取这些直线的交点（图2）。

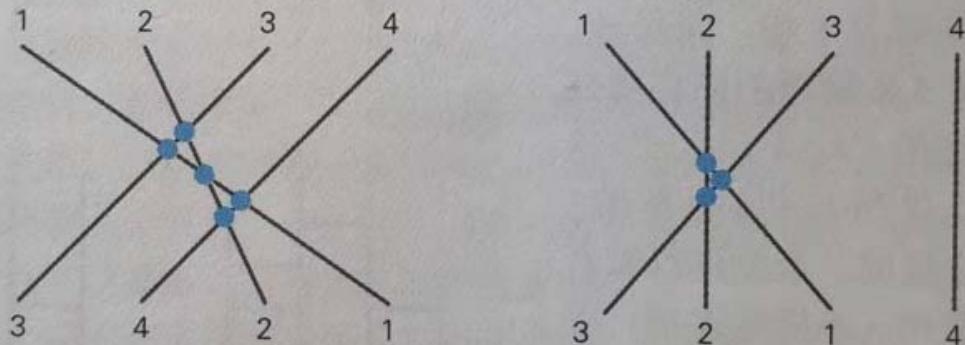


图2 求交点

每个交点都对应一条横线，所以只需要像冒泡排序一样求数字交换次数就可以了。就本题来说，只需要对所有下方数字的数列执行上述处理，找出其中横线条数为 10 的排列方式即可。用 Ruby 就可以实现，代码如代码清单 56.01 所示。

顺便把原书给出的另外两种解法也贴出来，供有兴趣的小伙伴钻研。当然，我是一如既往地没看懂（也失去了去搞明白的耐心了）

(解法2)

还有一种方法：对下方数字，从左往右按顺序连线，使下方数字最终与上方数字中的目标数字相连（图3）。此时，我们可以从下方数字所在的竖线开始只向右连线。

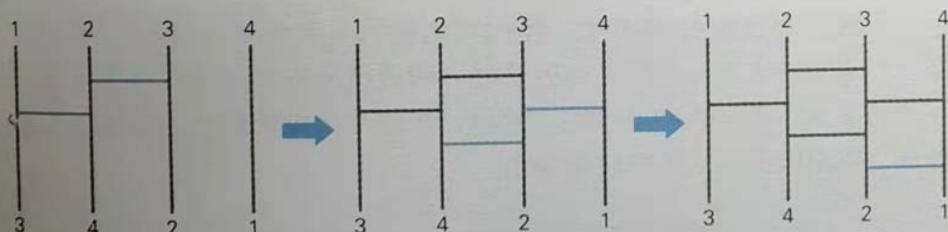


图3 从下方数字开始连线，直至与上方数字中的目标数字相连

这种方法的关键点在于所有的线都从下方数字连出，所以与前面一样，这里也要遍历下方数字的所有数列（代码清单 56.02）。

(解法3)

下面试着优化处理速度。我们可以在第2种解法的基础上优化一下，用递归的方式来实现。也就是说，假设已经以最少横线条数构成了一个有 $n - 1$ 条竖线的鬼脚图。然后，在这个鬼脚图的最右边增加1条竖线，以最少横线条数构成鬼脚图。

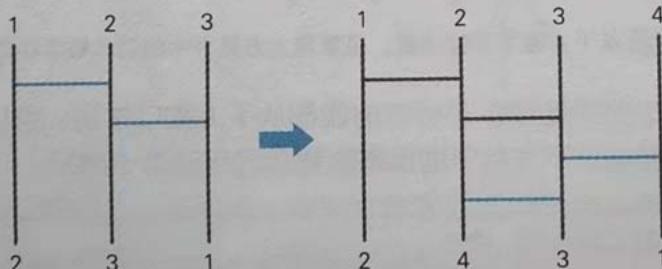


图4 向有 $n - 1$ 条竖线的鬼脚图中增加1条竖线

Point

关键在于把下方数字加到什么位置。如图4左侧的图所示，如果加在最右边，则横线条数不变；如果加在右数第1条和第2条竖线之间，则需要在最右边的2条竖线之间的最下方加1条横线。同样地，如果加在右数第2和第3条竖线之间，则要在当前所有横线的下方加2条横线（上图蓝色线条）；如果加在最左侧，则要加3条横线。

总结所有情况可知，如果已有2条竖线，要添加第3条竖线时，以横线条数为索引，排列方式数为元素，则可以得到以下表示排列方式的数组。

[1, 1]	处理前（要加0条横线的情况有1种，要加1条横线的情况有1种）
↓	
[1, 1]	往最右侧添加（横线条数不变）
[1, 1]	往右数第1条竖线前添加（要加1条横线的情况下有1种，要加2条横线的情况下有1种）
[1, 1]	往右数第2条竖线前添加（要加2条横线的情况下有1种，要加3条横线的情况下有1种）
[1, 2, 2, 1]	汇总（要加2条横线的情况下有1种，要加1条横线的情况下有2种，……，共6种）

57. Q57: 最快的联络网

57.1 问题描述

本题与学校常用的联络网有关。虽然最近很多人都用微信联系，但是要想确保联系到某个人还是得打电话。下面我们就组建一个联络网。

使用联络网时，是根据箭头方向，由前一个人联系后一个人的。为确保信息正确传达，最后一个人要联系第一个人。这里规定一个人不能同时和多人通话。

假设某个班级有老师1人，学生14人，两人打电话会花费1分钟。如果是如图5①所示的联络网，则需要9分钟老师才能确认所有人都联系过了（箭头下方的数字是打电话花费的时间）。如果是如②所示的联络网，那么虽然

应该是所占用的时间段(每段为1分钟)

联系到最后一个人的时间缩短了，但最后一个人和老师确认时常常会遇到老师正在与其他学生通话的情况，所以最终反而花费了10分钟（在②中，最下方的学生是直接和老师联系的，所以不需要再和老师确认）。

①中，老师拨打2次电话，接听2次电话，共通话4次；②中，老师拨打4次电话，接听6次电话，共通话10次。我们要尽量减少老师的通话次数，以减轻老师的负担。

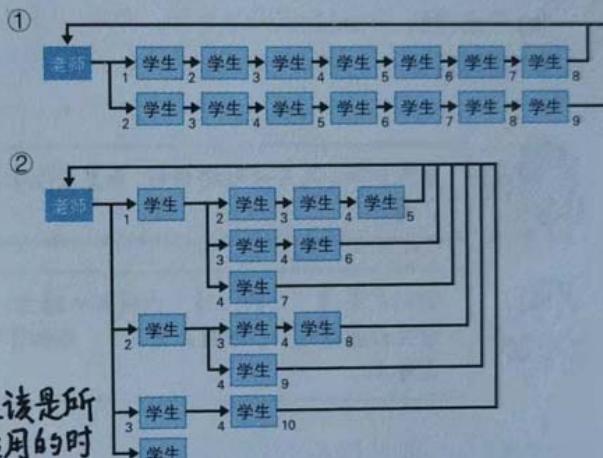


图5 联络网示例

问题

求直到老师最终确认所有人都已联络过时，联络所花费的最短时间。

并求在最短时间的联络网中，老师的最少通话次数。

57.2 解题分析

57.2.1 学生的状态

学生状态可以分为以下几种：

S0：尚未收到联络的学生，最终必须变为0

S1：直接收到老师联络的学生。这一类学生可以继续联络其他学生，也可以就此打住，他们即便处在

某一联络分支的最末端，也不需要给老师回电。他们不会变为其他类别

S2: 收到来自其他学生的联络，且尚未发出联络的学生。这一类的学生不能在他自己收到联络后就此打住，必须联络其他学生或者给老师回电。分为以下几种情况：

- (1) 如果他联络其他学生的话，就变为 S3 类型（之后他还可以继续联络其他学生）
- (2) 如果他联络过其他学生的话（变为 S3 类型），就不能（不应该）再给老师回电
- (3) 如果他在收到联络后给老师回电（变为 S4 类型）的话，后面就不应该再联络其他学生

S3: 收到来自其他学生的联络，且联络过其他学生（由 S2 类型变化而来）。这一类学生后续可以继续联络别的学生，也可以就此打住

S4: 给老师回过电话的学生。这类学生可能是来自于 S2. 他们不能再联络别人，即可以看作已经完成联络而退出联络网

进一步，由于学生之间仅按以上状态区分，而不区分个人身份。所以重要的只是处于各个状态的学生的人数，以及某个状态有几个学生给状态 S0 的学生或者老师打电话，而不关心具体是谁打电话谁接电话。

以小写的 s0~s4 分别对应于处于状态 S0~S4 的学生的个数，初始状态为 $s0=N, s1=s2=s3=s4=0$.

最终的状态应该为 $s0=0, s2=0$, 其它 $s1,s3,s4$ 只要满足 $s1+s3+s4=N$ 即可。

57.2.2 状态转移

状态转移取决于某个状态可以发生什么样的动作。注意，以下假定不会出现打电话冲突的情况，比如说同时两个人给老师打电话，或者同时两个人给某个 S0 状态的学生打电话。

老师(以下简记为 T)可能有三种动作：

Case-T1: Do nothing, just wait

Case-T2: 给处于 S0 状态的学生打电话

Case-T3: 接听某个处于 S2 状态的学生的电话

以下分这三种情况讨论。

Case-T1: Do nothing, just wait

在这种情况下，处于 S2,S3,S4 的学生都可能给处于 S0 状态下的学生打电话，只要满足总的电话数不超过 S0 人数个数。所以可以通过以下所示的三重嵌套循环来遍历所有各种情况：

```
# Teacher wait
for k in range(min(s0,s1)+1):
    for j in range(min(s0-k,s2)+1):
        for l in range(min(s0-k-j,s3)+1): # S3 call S0
```

在以上各种组合中，各状态的人数变化情况分析如下(以[S1 to S0]表示一个 S1 状态的人给一个 S0 状态的人打电话，余者类推):

每个[S1 to S0]会导致一个 S0 状态的学生变为一个状态 S2 的学生，S1 状态人数不变

每个[S2 to S0]会导致一个 S0 状态的学生变为一个状态 S2 的学生，同时打电话者本人变化为 S3 状态

每个[S3 to S0]会导致一个 S0 状态的学生变为一个状态 S2 的学生，S3 状态人数不变

由于老师没有接电话，所以 S4 人数不变

因此当 k,j,l 分别表示[S1 to S0]、[S2 to S0]、[S3 to S0]的人数时，各状态人数变化如下:

```
s0_nxt = s0 - (k+j+l) # S1 call S0: S0-->S3;
s1_nxt = s1
s2_nxt = s2 + k + l
s3_nxt = s3 + j
s4_nxt = s4
```

Case-T2：给处于 S0 状态的学生打电话

老师给 S0 状态的学生打电话，会导致一个学生从 S0 变到 S1。同时，供 S1,S2,S3 打电话的 S0 的人
数个数比 Case-T1 少了一个。

各状态人数变化状况可以参照 Case-T1 进行分析，此处不再赘述

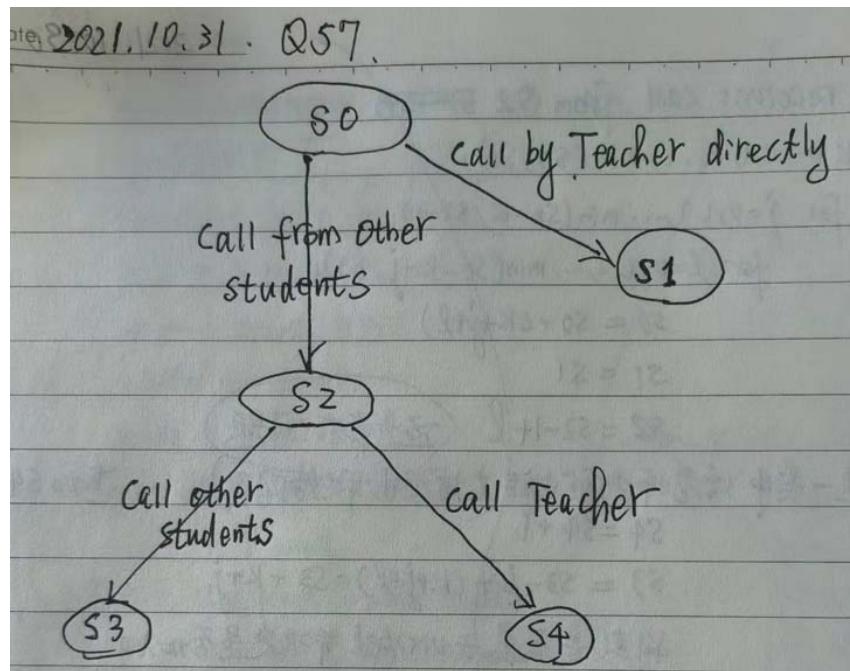
Case-T3：接听某个处于 S2 状态的学生的电话。注意只有当 s2 大于 0 时才有可能

老师接听某个 S2 状态的学生打电话，会导致某个 S2 状态学生变为 S4。

其它 S1,S2,S3 打电话的 S0 的人数分配与 Case-T1 相同。

各状态人数变化状况可以参照 Case-T1 进行分析，此处不再赘述

处于各个状态的人数的变化情况可以图示如下（注意，这个并不是有限状态机的状态转图，虽然看起来很像，我一时没有找到更好的表现方式）：



57.2.3 广度优先搜索

基于以上讨论，原问题可以重新表述为从状态（这个状态是指整个宏观状态，不要与以上学生状态 $S_0 \sim S_4$ 混淆！） $[s_0, s_1, s_2, s_3, s_4] = [N, 0, 0, 0, 0]$ 出发，经过以上允许的联络方式变化到 $[0, k_1, 0, k_2, k_3]$ ）所需要的最短的时间，显然这是一个图搜索之最短距离问题，因此可以用广度优先搜索算法来解决。

算法流程如下：

$Start = [N, 0, 0, 0, 0]$

$Step = 0$

初始化队列 q 和 $visited$ (implemented as set())

将 {Start, step} 加入队列 q

将 Start 加入已访问集合 $visited$

While q is not empty:

$cur, step = q.pop()$

如果 cur 等于目标状态 ($cur[0] == cur[2] == 0$)：结束搜索，返回 $step$ 结果；否则，继续搜索

#case-T1: Teacher wait

遍历各种可能的 S_1, S_2, S_3 给 S_0 打电话数的组合，for each:

更新 $s_0 \sim s_4$

如果 nxt 不在 $visited$ 中：

将{nxt, step+1}加入队列

将 nxt 加入 visited

#case-T2: Teacher call student in S0 state

...

#case-T3: Teacher receives a call from state S2

...

最后退出循环时最后从队列中弹出的 step 即为所求最短所需时间。

58. Q58: 丢手绢游戏中的总移动距离

58.1 问题描述

让我们一起追忆童年，来玩“丢手绢”的游戏吧。在丢手绢游戏中，一个人负责丢手绢，其他人则围成一圈坐着，然后丢手绢的人围着圈跑。一旦丢手绢的人把手绢丢到某个人身后，这个人就要在丢手绢的人跑了一圈重新回到自己身后之前，察觉到并追上他（丢手绢的人如果跑完一圈，就在被丢手绢的人原来的位置坐下）。

这里假设所有人跑动的速度一致，因而丢手绢的人一定不会被追上。另外假设被丢手绢的人也一定会在丢手绢的人跑完一圈之前察觉到。一直进行游戏，使围成一圈的人的排列顺序“变为最初顺序的倒序”。这里，由于大家是围成一圈，所以可以不考虑坐的位置，只考虑顺序，求最后所有人移动的总距离。计算移动距离的时候，假设围成一圈的人两两之间距离为1。

例) 6个人玩游戏时

编号为1~6的6个人围成一圈坐下。丢手绢的人编号为0，假设从0把手绢丢到1身后开始，且大家皆为顺时针跑动。使得最终顺序变为逆序的过程是：

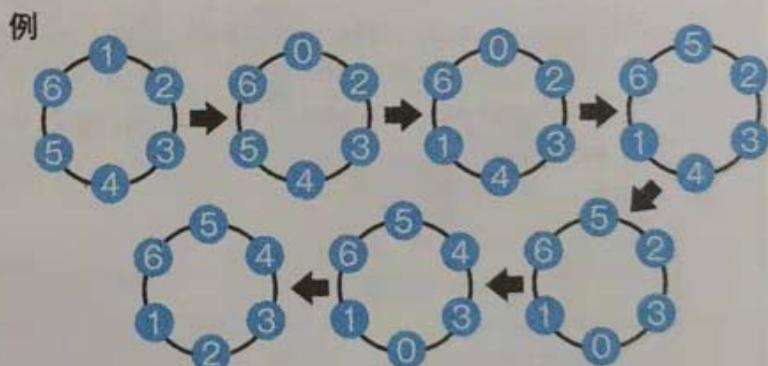


图7 6个人玩游戏时

变为逆序的过程是：0在1身后丢手绢并跑一圈，移动距离为6；1在5身后丢手绢并跑一圈，移动距离为10；5在0身后丢手绢并跑一圈，移动距离为8；0在4身后丢手绢并跑一圈，移动距离为9；4在2身后丢手绢并跑一圈，移动距离为10；2在0身后丢手绢后并跑一圈，移动距离为8。最后的总移动距离为51（= 6 + 10 + 8 + 9 + 10 + 8）（图7）。

问题

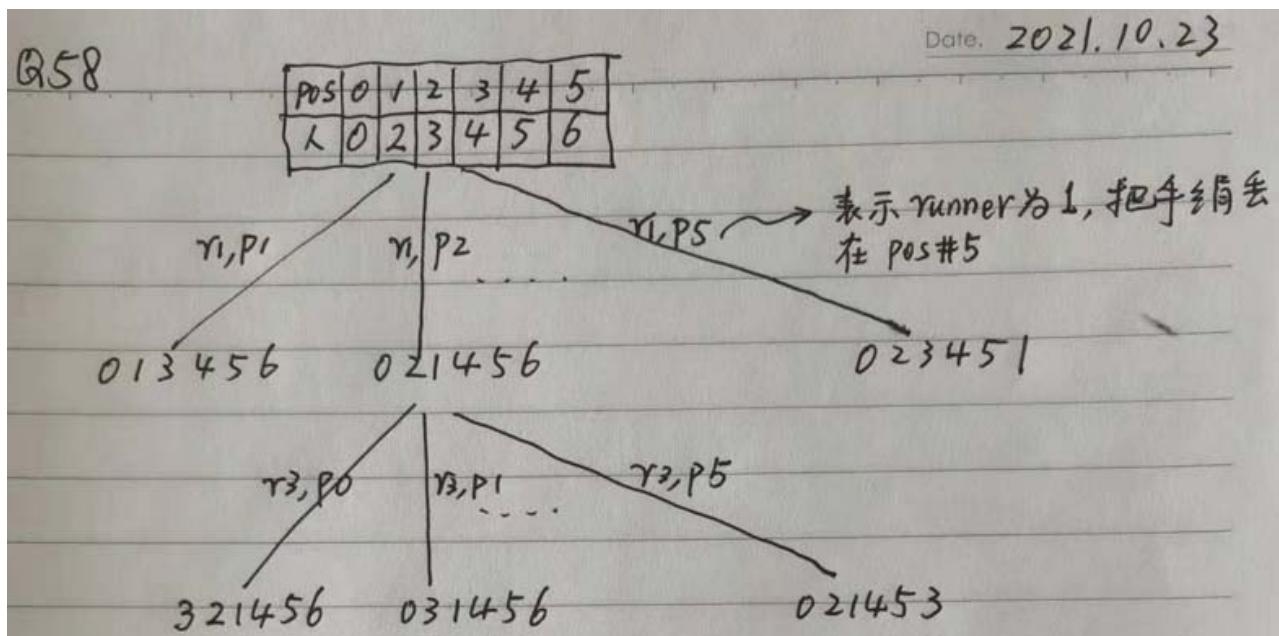
假设编号为1~8的8个人围成一圈坐下，另外有一个编号为0的人负责丢手绢，求最终顺序变为逆序时的最短的总移动距离。

58.2 解题分析

搜索最短距离，图搜索问题中的最短距离问题，可以用广度优先搜索策略来解决。

58.2.1 搜索树示意图

搜索树示意图如下：



用一维数组表示当前状态，但是要注意实际上表达的是围成一圈的状态。

算法流程如下：

$S_0 = [1, 2, 3, \dots, N]$

$S_1 = [0, 2, 3, \dots, N]$

Steps = N

初始化队列 q 和 visited(implemented as set())

将 $\{S_1, steps, runner, start\}$ 加入队列 q

将 S_0 和 S_1 加入已访问集合 visited

While q is not empty:

 curState, steps, runner, start = q.pop()

 如果 curState 等于目标状态：结束搜索，返回结果；否则，继续搜索

 For k = 0, 1, 2, ..., N-1: #尝试将手绢丢在 N 个不同的可能位置

交换当前 runner 与 curState 的位置 k 的人得到新的状态 nxt 和 new_runner

如果 nxt 不在 visited 中：

计算本次交换所需要的步数 curSteps

将{nxt, steps+curSteps, new_runner, k}加入队列

将 nxt 加入 visisted

最后从队列中弹出的 steps 即为所求结果

需要注意的要点：

- (1) 0 号玩家第一步固定地从把手绢丢在位置 0 (1 号玩家) 后面开始，因此 BFS 从 1 号玩家作为 runner 开始。0 号玩家需要的步数不要忘记
- (2) 搜索过程中不仅要记录当前状态，还需要记录到目前为止累积步数，当前 runner，已经当前 runner 从哪个位置出发
- (3) 计算当前 runner 丢手绢交换位置的步数时，需要注意 runner 需要先跑到预定位置，然后再跑一圈才能进入位置

考虑到围成一圈的对称性以及本题只要求相对位置变为逆序，因此在以上用一维数组来表示排列状态时，目标状态不是一个而是经过循环移位后等价的 N 个。参见代码中的 isTargets()

58.3 后记

N=8 时的答案与原书答案是一致的，但是 N=6 时与原书给的题解要小 (48 vs 51)，经过仔细查验，确信原书给的答案不正确。原书给的移动过程所需要的步骤数的确更短，但是就总的移动距离而言我的更短。。。N=6 时的我所得到的移动过程如下所示：

。。。待补充

以上 N=6 的移动过程有兴趣的小伙伴可以检验。

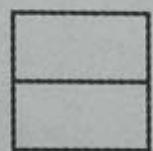
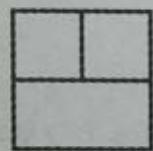
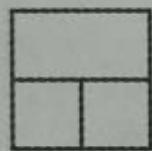
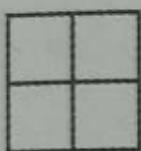
心中有点小小的激动，找出一个“错误”不是一件容易的事情^-^。

59. Q59: 合并单元格的方式

59.1 问题描述

日本人非常喜欢合并单元格，譬如“Excel 方格纸”^①这种用法。这次我们来探讨一下电子制表软件中“合并单元格”相关的问题。假设这里有2行2列的单元格，则合并后可得到的新的单元格如图9所示，共8种。图9中NG处的形状是合并不出来的，需要排除掉。

OK



NG

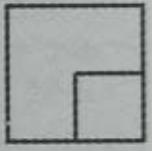
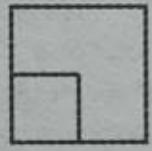
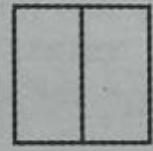
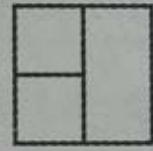
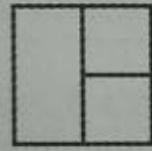
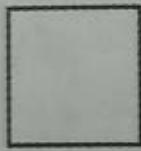
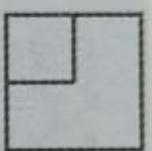
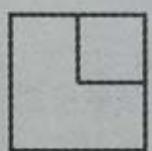


图9 2行2列的单元格的合并

问题

假设这里有4行4列的单元格，求共有多少种合并方式？此外，最终不存在 1×1 的单元格的合并方式有多少种（也就是说，有多少种合并方式能使得合并后得到新的单元格中不存在未合并的单元格）？

59.2 解题分析

似曾相似燕归来。。。

第一感就认出了这道题目跟之前“Q32：榻榻米的铺法”的相似之处，本题完全可以看作是Q32的增强版。在Q32中，榻榻米的尺寸是固定的 1×2 的大小。而本题可以看作是用任意尺寸长方形或正方形对房间铺设的问题！

基于这一认识，可以直接基于Q32的题解进行一些适当修改就可以的得到本题的题解。喜欢挑战的小伙伴如果没有做过前面的Q32（或者做过了但是记忆模糊了）的话，可以先独立做做这道题目，如果卡住了再回头去看一下Q32的题解看看能不能得到启发。最后还是卡住了的话，再看以下题解。

基于 Q32 的题解方案，有以下要点：

- (1) 外加围栏，以方便判断
- (2) 行扫描（或者列扫描也可以，道理相同），以很小的冗余扫描代价来换取巨大的判断简化的利益
- (3) 从某一个点出发的可能的合并（对应于榻榻米铺法）方法的遍历
 - a) 这是本题与 Q32 的本质的差一点。这里就不解释了，直接看代码很简单明了。单就代码而言甚至比 Q32 的还要简单。

第 3 点这是本题与 Q32 的本质的差异点。这里就不解释了，直接看代码很简单明了。单就代码而言甚至比 Q32 的还要简单。

其中利用到了 `numpy.any()` 函数用于判断某矩形区域内是否全 0。因为本题中并没有要求给出合并方案的图示化（事实上也不可能，因为种类数之多远远超出了直感），所以在递归调用时没有给合并块（“榻榻米”）编号并标记到格点内，而只是简单地用“0”表示空格，“1”表示已经被合并的格子。

60. Q60：分割为同样大小

60.1 问题描述

这里有横 m 格、纵 n 格的长方形。假设要把长方形分为面积相等的两个部分。要求两个部分（同色的部分）里的所有方格都要纵 / 横相连（相邻）。也就是说，不能把同色的部分分割到多处，而且即便对角连着也不看作是相连。

两个部分不要求形状一致，只要求面积相等。分割的最小单位是 1 个方格，不能斜着分割 1 个方格，也不能将 1 个方格分为多个。

当 $m = 4, n = 3$ 时，符合条件和不符合条件的分割方法如图 10 所示。

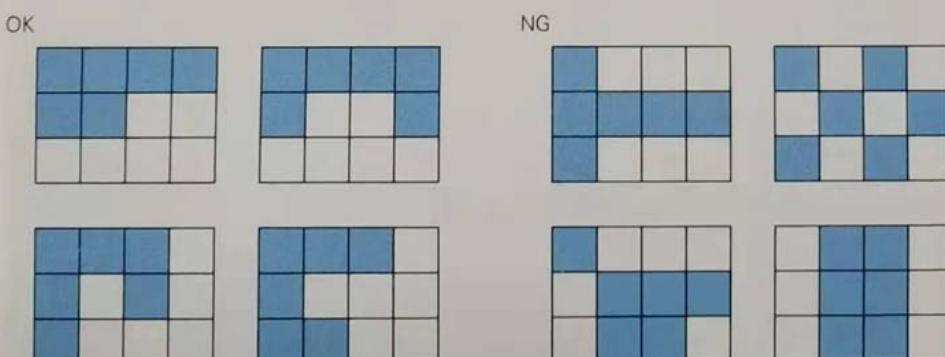


图 10 符合条件和不符合条件的分割方法示例

问题

求当 $m = 5, n = 4$ 时，有多少种分割方法（以分割线的位置为准。如果分割线一致，即使两个部分的颜色相反也要是看作是 1 种分割方法）？

60.2 解题分析

由于题目要求说“如果分割线已知，即使两个部分的颜色相反也看作是同一种分割方法”，这其实是说两种颜色是对称的，关键的是分割线的形状。“分割线这一侧涂成颜色 1 和另一侧涂成颜色 2”与“分割线这一侧涂成颜色 1 和另一侧涂成颜色 2”，是相同的。

60.2.1 思路 1

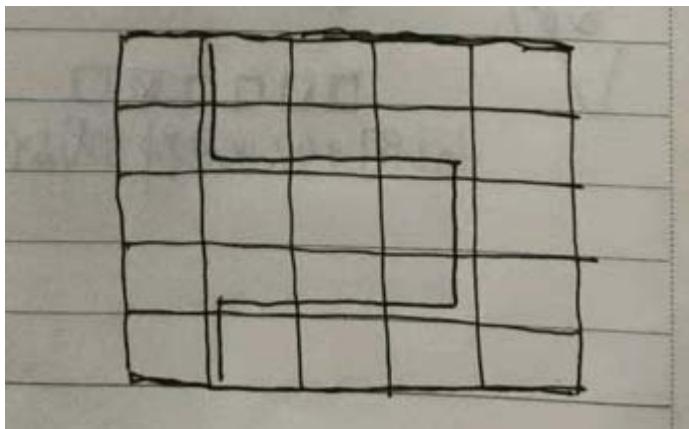
最直观的方法是，随机挑选一半的格子涂成一种颜色，另一半涂成另一种颜色。然后判断两个颜色的区域各自的连通性。

4*5 的区域有 20 个格子，所以分区的组合数有 $\binom{20}{10} = 184756$ 种，针对其中每种都要做两次连通性检测，两个颜色各一次，因为一个颜色连通不能保证另一个颜色的连通。连通性检测在 Q66 中“已经”出现了（因为我恰好先做了 Q66），在 Q66 中只需要做单方面的连通性检测。

60.2.2 思路 2

把问题看作是划分割线的问题。从边上某点开始沿内部格子的边画不相交无重复的线直到到达另外一个边上的点为止，这样划好后自然地保证边的“两侧”的区域格子连通。然后判断这个边界“两侧”的格子数是否相等。

分割线示意图如下所示：



60.2.3 思路 3

从区域生长的角度来考虑。

由于前面说了分割线形状相同，即便两个区域着色不同也算同一种情况。而左上角总是属于某一种颜色，所以可以考虑从左上角开始（当然考虑从任何一个格子开始都可以，都是等价的）。考虑从左上角开始始终以挨着（即保持连通性）的方式生长，每次增加一个格子着同样的颜色，直到生长区域的总面积等于一半。然后再检查未着色（或者说是着另一种颜色）的区域是否连通。

本题解采用思路 3 来实现。

问题可以看作是深度优先路径遍历问题。从每一个节点出发，按照题设要求的规则进行深度优先搜索，每遍历完一次计数一次。与 Q61 相类似（恰好先做了 Q61^-^），所以算法流程基于 Q61 的流程进行修改。

但是要注意的是，以本思路这种区域生长方式，有可能按不同顺序生长出相同的着色方式，因此需要进行重复性检验。以下用 `visited` 来存储已经访问过的着色方式，以避免重复。

基本算法流程如下：

`nodes` 初始化及加固栏处理(外加一圈格子以方便判断)

`Count = 0 # As global variable`

`Visited = set()`

`Def search(h, w, nodes):`

If 如果已着色格子数等于总数的一半： #与 Q61 不同之处

如果这种着色方式未被访问过：

将这种着色方式加入 `visited`

If 另一半未着色格子区域是连通的： #与 Q61 不同之处

`Count = count + 1`

`return`

如果上节点为 0，则(1)更新 `nodes`; (2)递归调用; (3)恢复 `nodes`

如果下节点为 0，则(1)更新 `nodes`; (2)递归调用; (3)恢复 `nodes`

如果左节点为 0，则(1)更新 `nodes`; (2)递归调用; (3)恢复 `nodes`

如果右节点为 0，则(1)更新 `nodes`; (2)递归调用; (3)恢复 `nodes`

与 Q61 的另一个差异是在 Q61 中，从不同点出发是代表不同的情况，而本题中从任何点出发是等价的，因为关心的是最终划分好的分割线形状，因此本题固定从左上角格子出发，而不需要对出发点进行扫描。

60.2.4 连通性检测

同 Q66 的做法。

本题着色时是将格子填为 1，未着色区域则保持为初始化的 0。这样“1”区域生长结束后，只需要对“0”区域判断连通性（生长过程已经保证了“1”区域是连通的）。在图论算法中连通性问题等价于 reachability 问题，即从某个格子出发只允许横向或纵向走一格的话能够到达的所有格子。解决 Reachability 问题的经典策略是深度优先搜索。

算法流程如下：

找到一个为所找颜色 color 的格子（肯定存在），其坐标标记为 start

初始化栈 s

将 start 加入 s

While s is not empty:

 Cur = s.pop()

 将 grids[cur] 置为 2 表示该格子已经被访问

 如果上边格子为 color，则将它加入栈

 如果下边格子为 color，则将它加入栈

 如果左边格子为 color，则将它加入栈

 如果右边格子为 color，则将它加入栈

最后判断 grids 还有没有为 color 的格子

补充说明：

- (1) 本题中可以直接将已经访问的格子置为别的值，因此不需要另设 visited 来记录已访问的节点。
这样做的好处是对于最后判断是否还有没有访问到的格子也很方便
- (2) 最后判断是否还有为 1 的格子。有的话表示还有 color 格子未被访问到，即 color 区域不是连通的。

60.3 代码及测试

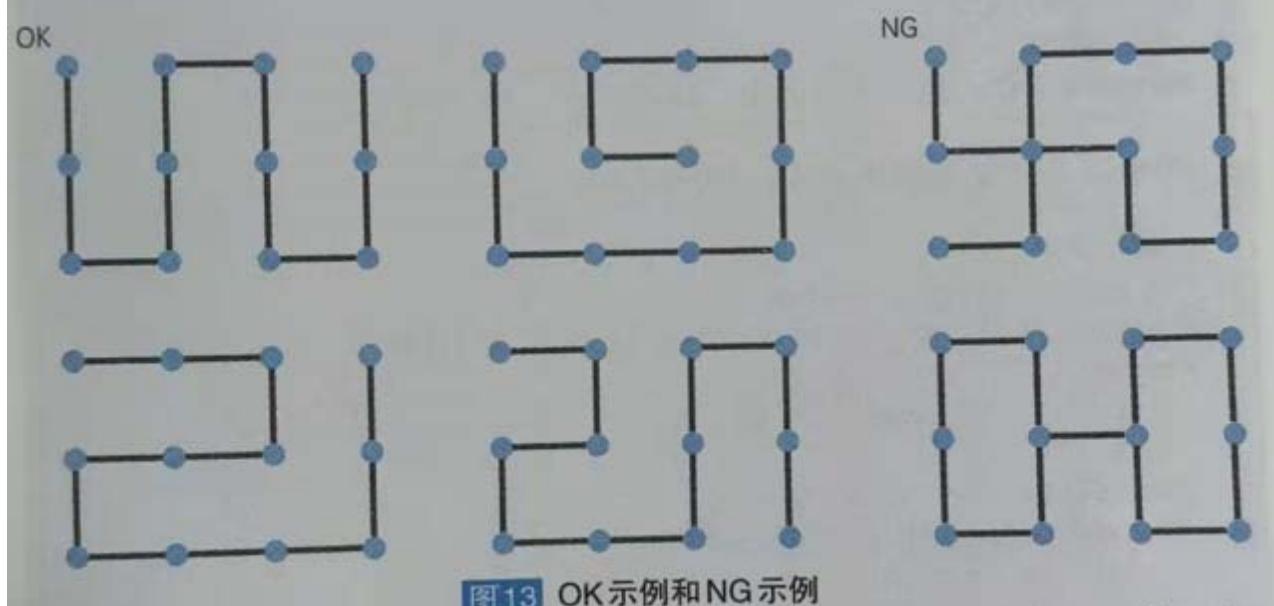
呃，熟悉的尴尬。。。正解是 245 种。老规矩，错误的题解自有其价值，贴出来看看有没有小伙伴能够看出其中的问题来。。。下面有关于错误分析的结论

61. Q61: 不交叉一笔画

61.1 问题描述

假设这里有横向的 m 个点和纵向的 n 个点，而我们要一笔连接所有点。要求只能用直线连接横向或者纵向相邻的点，并且不允许出现交叉。不能斜着连接，也不能用非直线连接（起点和终点重合也算是交叉）。

当 $m = 4, n = 3$ 时，图 13 中的 OK 示例是符合要求的连线方法，而 NG 示例里出现了交叉线条，所以不符合要求。



问题

求当 $m = 5, n = 4$ 时，共有多少种画法能一笔连接所有点（下称“一笔画”）（如果只是交换起点和终点，而连线的位置和形状都一致，那么只算作 1 种画法，但如果只是上下或者左右翻转后形状相同而位置不同，则算作不同的画法）？

根据题意要求：

- (1) 各节点不能重复访问。相应地，边也不会出现重复访问的情形
- (2) 由节点序列表示路径的话，正序列和逆序列表示相同的路径，不重复计算

61.2 解题分析

深度优先路径遍历问题。从每一个节点出发，按照题设要求的规则进行深度优先搜索，每遍历完一次计数一次。

基本算法流程如下：

```
nodes 初始化及加固栏处理(外加一圈格子以方便判断)  
Count = 0 # As global variable  
Def search(h, w, nodes):  
    If all nodes are visted:  
        Count = count + 1  
        return  
    如果上节点为 0, 则(1)更新 nodes; (2)递归调用; (3)恢复 nodes  
    如果下节点为 0, 则(1)更新 nodes; (2)递归调用; (3)恢复 nodes  
    如果左节点为 0, 则(1)更新 nodes; (2)递归调用; (3)恢复 nodes  
    如果右节点为 0, 则(1)更新 nodes; (2)递归调用; (3)恢复 nodes
```

就是这么简单粗暴（简洁明了），这就是套路的力量^-^

要点说明：

- (1) Nodes 为全 1 就表示找到了一条遍历路径（完成了一次无交叉一笔画）
- (2) 在每个节点处，探索上下左右是否可以画。不能出边界，外加的围栏节点初始化为“-1”，起到了保护和简化判断作用。
- (3) 由于要对 nodes 进行赋值以表示 visited 状态，所以在每次递归调用前要赋值，递归调用后要恢复。另外用 visited 来存储已访问信息会让问题变简单吗？对于此类（网格、棋盘等）问题似乎不会，甚至更麻烦。
- (4) 由于从任何一个节点出发都可以，因此外层需要对起始节点进行全扫描

以上流程没有考虑排除正序列和逆序列重复搜索的问题，因此浪费了一半的时间，所得答案要除以 2 才是正确答案。目前暂时没有想到如何排除正序列和逆序列重复搜索，暂时就此将就。

62. Q62: 日历中的最大矩形

62.1 问题描述

请试着找出单个月份的日历上，“只包含工作日”的最大的矩形。这里规定这个矩形不能包含周六日，不能包含假期，也不能跨月份。

举个例子，2014年4月~2014年6月的相应矩阵如图14所示，这些矩阵的总面积为51（4月=16天，5月=20天，6月=15天）。

2014年4月							2014年5月							2014年6月						
日	一	二	三	四	五	六	日	一	二	三	四	五	六	日	一	二	三	四	五	六
			1	2	3	4				1	2	3		1	2	3	4	5	6	7
6	7	8	9	10	11	12	4	5	6	7	8	9	10	8	9	10	11	12	13	14
13	14	15	16	17	18	19	11	12	13	14	15	16	17	15	16	17	18	19	20	21
20	21	22	23	24	25	26	18	19	20	21	22	23	24	22	23	24	25	26	27	28
27	28	29	30				25	26	27	28	29	30	31	29	30					

$4 \times 4 = 16$ 天

$4 \times 5 = 20$ 天

$5 \times 3 = 15$ 天

图14 2014年4月~2014年6月的情况

问题

求2006年~2015年这10年中，由每个月的“工作日”构成的最大矩形，并求所有120个月的矩形面积之和。

* 假日数据可以从以下网站中下载：

URL <https://github.com/leungwensen/70-math-quizzes-for-programmers>

URL <http://www.ituring.com.cn/book/1814> (点击“随书下载”)

假日数据是 /zh_CN 目录下的 q62-holiday.txt 文件，公休日因调休变为工作日的数据是同目录下的 q62-extra-workday.txt 文件。

62.2 解题分析

62.2.1 每个月的日历的矩阵表示

在python中可以用calendar模块的monthcalendar()来实现。参见另一篇博客。以下分别为两种方式打印的日历，第一条语句是将星期日设置为一周的第一天。

```
calendar.setfirstweekday(calendar.SUNDAY)
```

```
calendar.prmonth(2014,4)
```

```
print(np.array(calendar.monthcalendar(2014,4)))
```

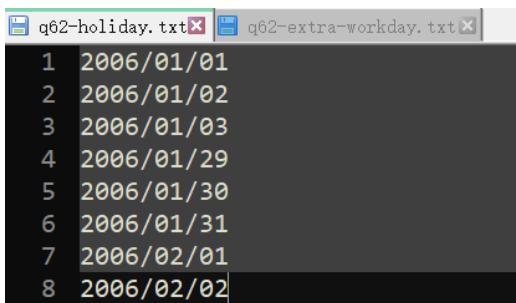
```
April 2014
Su Mo Tu We Th Fr Sa
      1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
[[ 0  0  1  2  3  4  5]
 [ 6  7  8  9 10 11 12]
[13 14 15 16 17 18 19]
[20 21 22 23 24 25 26]
[27 28 29 30  0  0  0]]
```

在由 `monthcalendar()` 生成了每个月的日历的矩阵表示形式后，为了方便后续搜索处理，将公休日（第 1 列和第 7 列）也都置为 0.

62.2.2 假日和调休公休日的处理

首先考虑假日的数据。

假日的数据在文件中是以以下格式存储的：



```
1 2006/01/01
2 2006/01/02
3 2006/01/03
4 2006/01/29
5 2006/01/30
6 2006/01/31
7 2006/02/01
8 2006/02/02
```

可以按行以字符串的形式读入，然后利用 `python datetime` 中模块将它们变换成 `datetime.datetime` object 并从中提取出年、月、日信息，然后再基于这个年月日信息将上一节得到的矩阵的对应元素置为 0（表示非工作日）。

接下来考虑因调休变为工作日的公休日的数据。存储格式同上，以同样方式处理即可，只不过这次是将对应元素置为非 0 值（比如说置为 1 即可）

在以下代码中用 `readfile()` 函数来从以上两个文件中读取数据，并提取日期信息，恢复出其中的年、月、日，然后存储在一个 `dict` 类型变量中，以 `(year, month)` 为 key，`value` 则是包含对应年月中的日期的列表。其中，设计到了利用 `datetime` 模块进行处理，关于 `datetime` 模块使用方法的简要介绍，参见博客：

62.2.3 求最大矩形

经过了以上处理后，日历矩阵变成了由 0 和非 0 元素构成的 5*7 的矩阵。问题变成了就这个矩阵中完全由非 0 元素构成的最大矩形面积。

62.2.4 暴力搜索

首先，求最大矩形问题当然可以用暴力搜索的方法来求解。

比如说，一个 2*2 的矩阵，其中以矩阵最左上角格子(0, 0)为最左上角的矩形有多少个呢？恰好是 4 个。对于一般的情况 $n*m$ 的矩阵，其中以矩阵最左上角格子(0, 0)为最左上角的矩形总共有 $n*m$ 个。扫描这 $n*m$ 个矩形排除掉中间含有 0 元素的矩形（或者将其面积置 0 更简单），求剩下矩形的最大面积即得“以矩阵最左上角格子(0, 0)为最左上角的矩形”的最大面积。接下来，同理可以求得以格子(0, 1)、(0, 2)、...、(1, 0)、(1, 1)、为最左上角的矩形的最大面积。再求这些最大值中的最大值即得当前矩阵中不含 0 元素的最大矩形面积。

这样暴力搜索的复杂度是多少呢？为简便起见，考虑原矩阵为正方形，大小为 $n*n$

首先，对矩形最左上角的格子的扫描， $n*n$

其次，针对每个矩形最左上角格子候选，其对应的可能的矩形数取决于它的坐标。假定它的坐标为(i, j)，则可能的矩形数为 $(n-i)*(n-j)$

这样，总的需要评估其面积的矩形个数为：

$$N_{rect} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} (n-i) * (n-j) \sim O(n^4)$$

这种方案只能想一想作为基准参考，不能去实现。

62.2.5 滑动窗搜索

暴力搜索是以格子（考虑某个格子为所求矩形的左上角格子）为基点进行搜索。也可以从另一个角度考虑滑动窗方案，考虑以不同大小和形状的矩形框在日历矩阵上滑动。因为是要求最大的矩形面积，所以扫描用的滑动矩形窗面积按从大到小的顺序安排，这样找到了第一个不包含 0 的滑动位置就找到了原题要求的最大矩形面积。

由于有可能多种形状的矩形框具有相同的面积，比如说 4*2 和 2*4 的矩形框的面积都为 8。所以先构建一个字典，以面积为 key，对应的可能的形状的列表作为 value，代码如下：

```
# 2. Construct the dictionary for rectangulars area-shape pair
area_shape = dict()
for i in range(1,6):
    for j in range(1,8):
        if i*j not in area_shape:
            area_shape[i*j] = []
        area_shape[i*j].append((i,j))
```

有了以上准备工作后，针对某个月的处理流程如下所示：

生成当月的日历的矩阵形式（以星期日为一周的第一天）

将第一列（星期日）和第七列（星期六）置 0

将 `holidays` 对应元素置 0 (注 1)

将 `extra-workdays` 对应元素置为 100(非 0 即可) (注 1)

`Found = False`

按面积从大(`max:35=5*7`)到小遍历：

获取面积为 a 的可能形状(`tuple(i,j)`)列表 `ij_list`

遍历列表 `ij_list`: `for (i,j) in ij_list:`

遍历形状为 `i*j` 的矩形窗在日历矩阵上滑动的可能的左上角位置：

如果当前滑动窗覆盖区域没有 0 元素，则返回

注 1：将 `holidays` 和 `extra-workdays` 对应元素的值重置时需要根据日期信息去确定它在矩阵中的对应位置。首先需要确定当月的第一天对应的 `weekday`(星期几)，这样就能确定当月 1 日在矩阵中的位置，进而可以推得指定日期在矩阵中的位置。这个处理对应于以下代码 (`extra-workday` 的处理与此相同)：

```
# Set holidays to 0
if (y,m) in h:
    holidays = h[(y,m)]
    for hday in holidays:
        # Find the position of the current holiday in month calendar matrix
        i = (hday + fst_wkday - 1)//7
        j = (hday + fst_wkday - 1)%7
        c[i,j] = 0
```

62.3 代码及测试

以上代码没有完全回答题目所问的问题。原因是原书所附带数据并不完整，只有到 2012/10/07 为止的数据。所以我只跟题图补充了 2014 年 4, 5, 6 月的假日数据（这三个月没有公休日调休的情况），并测试了这三个月的最大矩形面积与题目描述中提示的值是否一致。但是，关于 2014 年 6 月的题目描述中提示的值其实是错的，很显然存在一个 4*4 的完全工作日的矩形区域。

63. Q63: 迷宫会合

63.1 问题描述

涂抹横向和纵向排列的 $n \times n$ 个方格中的几个，制作成迷宫。这里规定被涂抹的方格是墙壁，没有被涂抹的就是道路。

两人分别从 A 点和 B 点同时出发，每次前进 1 个方格，且按照“右手法则”前进。所谓右手法则，就是靠着右边墙壁前进，不一定要按最短路径前进，但最终要回到入口或者到达出口（其中一人从 A 点出发向 B 点前进，另一人从 B 点出发向 A 点前进。A 点和 B 点的位置固定，假设就是左上角和右下角）。

那么，两人在途中相遇的情况一共有多少种呢？同时到达同一点算相遇，其中一人到达终点，另一人同时回到这个位置也算。

举个例子，当 $n = 4$ 时，图 17 中①的情况下两人可以相遇，②的情况下两人就不能相遇（①的情况下，如果 A 像 “↓ ↓ ↑ → → ↓ ↓ ← →” 这样移动，而 B 像 “← ↑ ↑ ← → ↑ ↓ ← ← ↑” 这样移动，在第 5 次移动时两人就会相遇）。

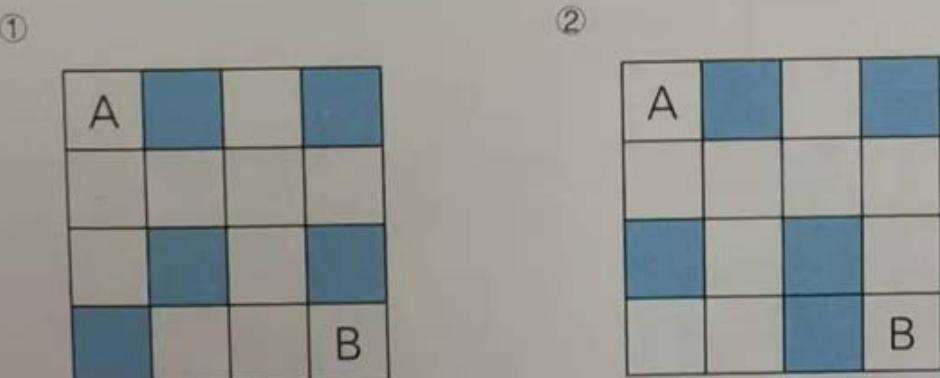


图 17 当 $n = 4$ 时的示例

问题

求当 $n = 5$ 时，两人能在途中相遇的迷宫模式有多少种？

63.2 解题分析

5×5 的网格共有 25 个格子，每个格子有两个可能（墙壁或通道），共有 $2^{25} = 33554432$ 种。但是，

并非每种都能构成有效的迷宫。相当惊人的一个基数，没有有效的缩小范围或者计算优化策略的话，会需要话很长的时间。

63.2.1 有效的迷宫

有效的迷宫是指入口（左上角）到出口（右下角）是连通的。因此判断某种通道/墙壁排列是否有效可以作为 Reachability 问题来处理，可以用广度优先搜索来进行判断。代码如下（注意 0 表示墙壁，1 表示通道，而且为了方便判断，外加一圈围栏，围栏全部初始化为墙壁）：

63.2.2 移动

根据题意，在迷宫中 A 和 B 都必须按照“右手法则前进”，这意味着，在任何一个地方，

先看向右能否前进，如果能则向右移动；如果不能则：

看向前能否前进，如果能则向前移动；如果不能则：

看向左能否前进，如果能则向左移动；如果不能则：

掉头向后移动

注意，前后左右是一个相对的概念，取决于人当时的朝向。因此：

如果上一次移动是向左的话，下一次移动就要按照“上>左>下>右”的优先度选择前进方向；

如果上一次移动是向下的话，下一次移动就要按照“左>下>右>上”的优先度选择前进方向；

如果上一次移动是向右的话，下一次移动就要按照“下>右>上>左”的优先度选择前进方向；

如果上一次移动是向上的话，下一次移动就要按照“右>上>左>下”的优先度选择前进方向；

由此可得代码如下：

63.2.3 能否碰上

在选定一种合法的迷宫模式后，两人是否能碰上按照以下算法进行判断：

A,B 位置分别初始化为[1,1]和[N,N] #考虑 0-indexed 以及外加了一圈围栏

A 和 B 的移动方向分别初始化“向左”和“向右” # why?请读者思考

While-loop:

按照上节所述规则移动 A

按照上节所述规则移动 B

如果 A 位置于 B 位置重合:

返回 True (表示可以碰头)

如果 A 和 B 是否至少有一人到达各自的终点, 返回 False (表示不能碰上)

需要注意的是,一开始受到题目描述中“所谓右手法则,就是靠着右边墙壁前进,不一定要按最短路径前进,但最终要回到入口或者到达出口”的误导,在失败判断中按以下条件处理:

```
if (A == [1,1]) or (A == [N,N]) or (B == [1,1]) or (B == [N,N]):
```

但是这个是不对的。按照右手法则,在某些特别的迷宫配置条件下,完全有可能 A 和/或 B 先回到起点然后再移动,然后再碰头。按照以上条件的话,再碰头之前回到自己出发点就判为失败了。

63.2.4 搜索

基于以上准备工作,搜索流程如下所示:

遍历所有可能的“墙壁”/“通道”配置, for each of one:

如果 A 或 B 起点位置为墙壁, go to try the next configuration

如果不是合法的迷宫, go to try the next configuration

如果是合法的迷宫:

判断是否能够碰头并相应计数

63.3 代码及测试

速度太慢了。需要进一步考虑优化方法。最近几道题目都这样,习惯了。。。^_^-。不过作为普通的渣渣,不指望一蹴而就地搞定,而是先做一个可行的方案然后来做优化比较可行吧^_^-。

64. Q64: 麻烦的投接球

64.1 问题描述

棒球运动有一项基本的投接球练习。在投接球中，重要的是投出让对方接起来比较容易的球。这里假设有 12 个学生，以 6 人为一组分为两组相对练习。

假设如图 19 所示，为每个学生分配 1~12 的编号，大家反复进行投接球练习，使得 1 号学生手上的球最终会传给 12 号。但同时，有以下几个条件。

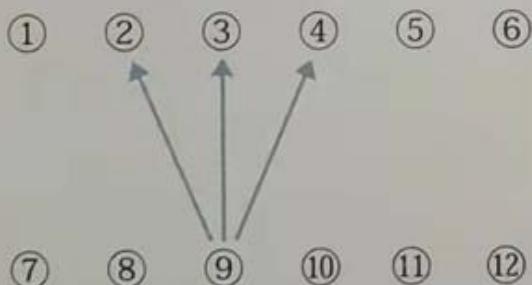


图 19 投接球示例

条件 1：1 个学生同时只能拿 1 个球

条件 2：每次只能由 1 个学生投球，并且一定会有接球人

条件 3：最开始只有 1~11 号学生手上有球

条件 4：每个学生投球的目标只能是三者（正对面及其左右的学生）之一

条件 5：投接球结束后，除了 1 号和 12 号，其他学生手上一定要拿着最开始的球

最开始只有 12 号手上没有球，所以第 1 次投球的学生只能是 5 号或者 6 号。

问题

求 1 号手上的球传到 12 号并且满足条件 5 时，最少投球次数是多少？假设每个人手上的球都能彼此区别开来。

64.2 解题分析

老规矩，先来一个最基本的方案。

由于是找最小需要的投接次数，所以可以看作是一个图搜索问题的最短路径问题。

广度优先搜索。

64.2.1 状态表示

12 个人对应 12 个位置，可以表示为一个数组。

11 个球可以标记为 0,1,2, … ,10。空的位置（即当前不持球的人）填入“-1”（可以认为是持有“空球”）（当然其实就用 11 表示“空球”也可以）。

因此初始状态就是：[0,1,2, … ,10, -1]，而目标状态则是[-1,1,2, … ,10, 0]。

对于每个人（位置），允许向他投掷球的人（位置）是固定的，可以预先建立一张查找表，比如说：

0: (0,1); 1: (0,1,2); …; 10: (3,4,5); 11: (4,5);

当然允许投掷的前提是当前这个人未持球（空的）。

这样，问题就转变成从状态出发[0,1,2, … ,10, -1]，在规定的投掷动作要求下到达目标状态[-1,1,2, … ,10, 0]所需要的最少投掷次数（投掷可以看作是空球”-1”与投掷位置处现有的球的位置交换）。

64.2.2 算法流程

算法流程如下：

Start = [0,1,2,3, … , -1], target = [-1,0,1,2,3, … , 10]

Step = 0

初始化队列 q 和 visited(implemented as set())

将{Start, step}加入队列 q

将 Start 加入已访问集合 visited

While q is not empty:

cur, step = q.pop()

如果 cur 等于目标状态：结束搜索，返回 step 结果；否则，继续搜索

搜索 cur 中持空球的位置 empty

For k = throw_dict[empty]: #遍历可以给位置 empty 投球的人（位置）

交换当前位置 empty 与位置 k 的球，得到 nxt

如果 nxt 不在 $visited$ 中：

将 $\{nxt, step+1\}$ 加入队列

将 nxt 加入 $visited$

最后从队列中弹出的 $step$ 即为所求结果

65. Q65: 图形的一笔画

65.1 问题描述

现有如图20左侧所示的 4 个图块。我们来思考一下对用这些图块横向和纵向拼合而得到的图形一笔画的情况。

举个例子，如果横向和纵向都有 2 个图块，则可以得到如图20右侧所示的图形（拼合之后，图块的边界重合）。上面的 2 个图形不能一笔画出来，下面的 2 个可以。

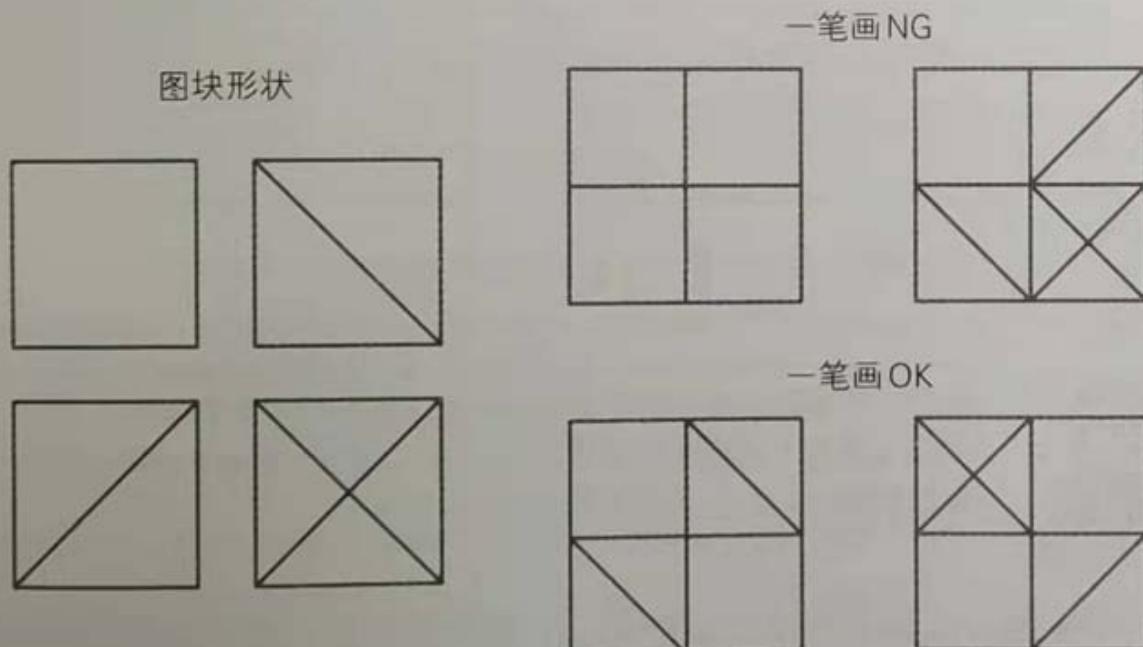


图20 图块形状和一笔画的示例

问题

求横向有 4 个图块、纵向有 3 个图块（即 3 行 4 列）时，能拼出多少个可以一笔画的图形（上下镜像和左右镜像的图形算作不同图形）？

65.2 解题分析

3×4 的网格共有 12 个格子，每个格子可以任选以上 4 种模块之一，共有 $4^{12}=16777216$ 种。嗯，相当惊人的一个基数，没有有效的缩小范围或者计算优化策略的话，会需要化很长的时间。不过，先来一个最基本的方法吧—先确保温饱，再追求小康。。。

65.2.1 一笔画的条件

这是一个图论中的一个经典问题。可以追溯到哥尼斯堡七桥问题，当年欧拉大神以解决这个问题为契机创立了图论这一数学分支。事实上，这个问题还有个炫酷的名字叫做欧拉路径问题：欧拉路径就是一条能够不重不漏地经过图上的每一条边的路径。进一步，而若这条路径的起点和终点相同，则将这条路径称为欧拉回路。

关于图论就不瞎 BB 了（说多了怕露馅^-^），简单地说结论：对于一个无向连通图，（边的）无重复遍历的充要条件是度数为奇数的顶点的个数为 0 或者 2。“连通图的边的无重复遍历”的通俗版本就是一笔画。其中度数就是每个顶点所连接的边的个数。“有向图”是什么情况不记得了，反正跟本题无关，先不管了。

这个知识点是解决本问题的先决条件。当然即便不知道这个知识点也是可以解的，就是针对每个图形去做遍历搜索也是可以判定是否满足一笔画的条件，但是那样的话你需要的计算时间就会再高几个数量级。

65.2.2 各模块的定点度数

上面所述 4 种图形的 4 个顶点的度数分别为 ([左上顶点, 右上顶点, 左下顶点, 右下顶点]):

左上模块: {2,2,2,2}

右上模块: {3,2,2,2}

左下模块: {2,3,3,2}

右下模块: {3,3,3,3}

注意，右下模块的中间还有一个顶点，但是该顶点的度数为 4，不会影响一笔画判断，所以可以忽略。

65.2.3 如何计算整个拼图的度数

可以分三大类顶点考虑：

第 1 类：四个角顶点(corner vertex)

其度数只跟一个块有关。比如说左上角顶点的度数就是左上角块的左上角顶点的度数；右上角顶点的度数就是右上角块的右上角顶点的度数；。。。余者类推

第 2 类：四条边上的顶点(edge vertex)

其度数跟两个块有关，是两个块的对应顶点的度数之和减 1（因为有一条边重叠了）。至于对应顶点是哪个，取决于是在那条边上。

第 3 类：中间的顶点(interior vertex)

其度数跟上下左右的四个块有关，是四个块的对应顶点的度数之和减 4（因为共 4 条边重叠了）。四个块的对应顶点分别是左上块的右下顶点，右上块的左下顶点，左下块的右上顶点，右下块的左上顶点。

65.2.4 算法流程

粗暴的遍历循环。。。

在统计某一种拼图模式的奇度数的顶点个数时，如果超过了 2 就表明肯定无法一笔画，可以提前退出。

66. Q66: 设计填字游戏

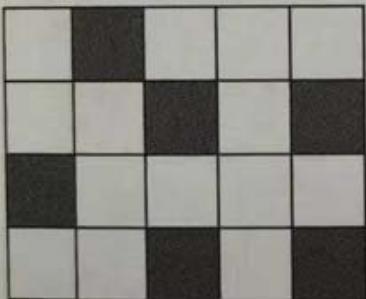
66.1 问题描述

填字游戏（Crossword Puzzle）就是在行与列交叉处的方格内填字的游戏。假设这里有填了字的方格（白色）和没有填字的方格（黑色），这些方格的设置有如下规则（图22）。

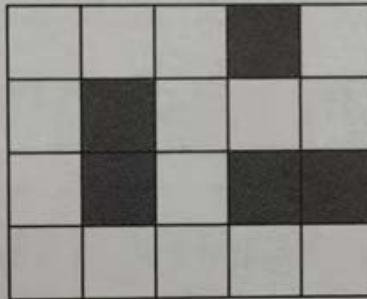
- 黑色方格不能纵向和横向相连
- 黑色方格不能分裂整个表格

（参考：[URL https://zh.wikipedia.org/wiki/填字游戏](https://zh.wikipedia.org/wiki/填字游戏)）

OK示例



NG示例1



NG示例2

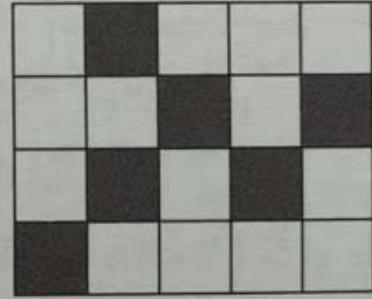


图22 填字游戏示例

问题

求在方格为5行6列的填字游戏中，满足上述条件的设计有多少种（只考虑黑色和白色的位置，不考虑其中填入的字）？

66.2 解题分析

与前面的Q32、Q59以及后面的Q68（碰巧先做了Q68）是类似的搜索问题。有兴趣的小伙伴如果在思考本问题时卡住了可以先看看本系列的以上三题的题解看看是否能找到头绪，如果还是卡住了的话再看本题的题解吧^-^。

66.2.1 基本算法流程

本题的基本搜索框架直接基于Q68（因为Q68是前几天刚做的记忆犹新改起来方便），

算法流程如下（代码中实际上用 grids 替代 seats）：

```

Seats 初始化及加固栏处理(外加一圈格子以方便判断)

Count = 0

Def search(h, w):
    If h == H+1: #说明已经到达最下面的围栏行，找到一个“不违规”的安排
        If 所有白色格子都连通:
            Count = count + 1

    Elseif w == W+1: #到达最右边的围栏列，换到下一行继续扫描
        Search(h+1,w)

    Else: #给当前格子填色，白色：1，黑色：2
        #填白色：不用进行违规检查。填完直接递归调用搜索右边一格
        Seats[h,w] = 1
        Search(h,w+1)
        Seats[h,w] = 0 # 返回上一级的状态

        #填黑色
        Seats[h,w] = 2
        判断填黑色后是否有违规情况出现，如果没有则继续探索右边一个座位：
        Search(h,w+1)
        Seats[h,w] = 0 # 返回上一级的状态
    
```

要点说明：

- (5) 与 Q32, Q59, Q68 一样，逐行扫描，每次向右移一格。右边到头即移到下一格。到达最下边的围栏行则表明已经完成一次搜索，找到一个合规的“预备”方案（因为还要做白色格子连通性检查）
- (6) “违规检查”只需要针对当前填黑色的格子进行。这是因为扫描是向右、下方向进行，而且只需要检查是否存在两个相邻格子都为黑色

与 Q68 不同的地方在于：

- (a) 违规检查不同
- (b) 不要求黑、白格子数相同，因此 search() 函数不需要传入各种颜色（对应 Q68 的 boy/girl）的数量

(c) 本题最后还要进行白色格子形成的区域的连通性，这个是最大的不同点。处理方式参见下一节

66.2.2 连通性检查

本题要求填完色后，黑色格子不能分裂整个表格，这是这道题目与 Q32,A59,Q68 最大的差异所在。换言之，白色格子构成的区域必须保持连通，这在图论算法中等价于 `reachability` 问题，即从某个白色格子出发只允许横向或纵向走一格的话能够到达任何其它的白色格子。解决 `Reachability` 问题的经典策略是深度优先搜索。

算法流程如下：

找到一个为白色的格子（根据上一节的搜索策略，一定存在至少一个白色格子），器坐标标记为 `start`

初始化栈 `s`

将 `start` 加入 `s`

While `s` is not empty:

`Cur = s.pop()`

将 `grids[cur]` 置为 0 表示该白色格子已经被访问

如果上边格子为白色，则将它加入栈

如果下边格子为白色，则将它加入栈

如果左边格子为白色，则将它加入栈

如果右边格子为白色，则将它加入栈

最后判断 `grids` 还有没有为 1 的格子

补充说明：

(3) 本题中可以直接将已经访问的格子清零，因此不需要另设 `visited` 来记录已访问的节点。这样做好处是对于最后判断是否还有没有访问到的格子也很方便

最后判断是否还有为 1 的格子。有的话表示还有白色格子未被访问到，即白色区域不是连通的。

67. Q67：不挨着坐是一种礼节吗

67.1 问题描述

注意，本问题不区分人，只考虑各个座位被占用的不同顺序的个数。

67.2 解题分析

67.2.1 基本思路

每个人选座位时，首先看有没有“超空位”（两边都没有挨着人的空位暂称为“超空位”），如果有则优先从“超空位”中选取。如果没有超空位，则从空位中任选一个坐下（空位是一定有的，因为人数和座位数是相等的）。

由于车厢中的作为安排方式（对面两排），两头两位的座位有一边是一定不挨着人的。如何表达才最方便于实现呢？

借用棋盘问题中的围栏思想，将两排座位排成一排，但是两头和中间各插入一个 dummy 座位（不能坐人），这样连同 dummy 座位总共有 15 个，可以表达为长度为 15 的数组。Dummy 座位置为“-1”表示不能坐人，正常座位初始化为“0”，被人占据了则复制为“1”。这样，判断一个正常座位是否为“超空位”，只要看它本身是否为 0 且两边都不是 1（即为“0”或者“-1”）。

67.2.2 动态规划

考虑已经有 k 个座位被占了，从当前状态出发往后的剩下的 $(12-k)$ 个座位被占的顺序数只与当前这个 k 个座位被占的状态有关。因此这个可以用动态规划的策略来解决。本题考虑用 top-down 的方式（递归+memoization）来实现动态规划策略。

算法流程如下：

```
Memo = dict()

Def search(pos):
    If pos in memo:
        Return memo[pos]
    If pos 全满:
        Return 1
    Cnt = 0
    hasSuperEmpty = False
    # 搜索是否有超空位
    For k in [1,2,3,4,5,6, 8,9,10,11,12,13]:
        如果是超空位更新 pos, 递归调用: Cnt = cnt + search(pos)
        hasSuperEmpty = True
```

If not hasSuperEmpty: #如果没有找到超空位

For k in [1,2,3,4,5,6, 8,9,10,11,12,13]:

 如果是空位，更新 pos，递归调用: Cnt = cnt + search(pos)

 Memo[pos] = cnt

Return cnt

初始化数组 pos[15]，其中：pos[0],pos[7],pos[14]初始化为"-1"，其余为'0'。

Count = search(pos)

68. Q68: 异性相邻的座次安排

68.1 异性相邻的座次安排

回想起学生时期调座位的时候，我们的心里总是会小鹿乱撞。想必很多人都对谁会坐自己旁边这件事莫名地激动吧？

这里我们考虑一种“前后左右的座位上一定都是异性”的座次安排。也就是说，像图26右侧那样，前后左右都是同性的座次安排是不符合要求的（男生用蓝色表示，女生用灰色表示）。

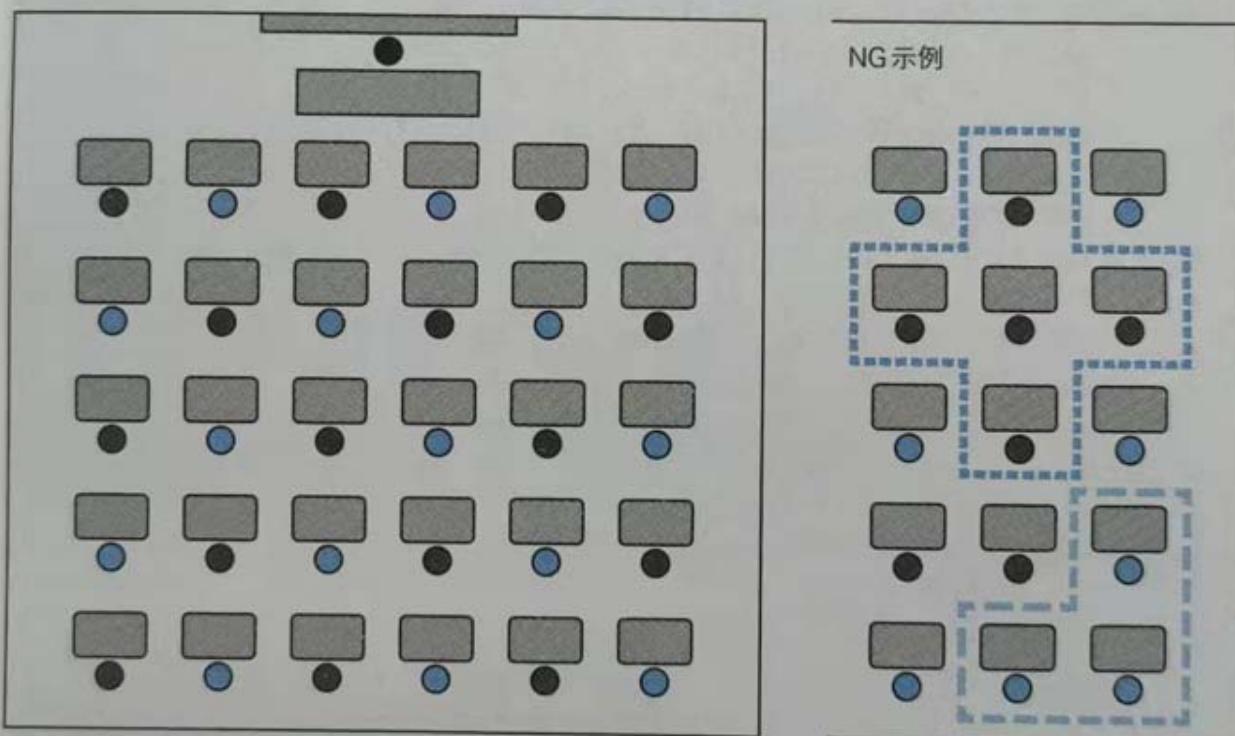


图26 座次安排示例

问题

假设有一个男生和女生分别有 15 人的班级，要像图26那样，排出一个 6×5 的座次。求满足上述条件的座次安排共多少种（前后或者左右镜像的座次也看作不同的安排。另外，这里不在意具体某个学生坐哪里，只看男生和女生的座次安排）？

这道题的描述应该是有问题的（不知道是原文的问题还是翻译的问题）。

前面的描述中提到“前后左右的座位一定都是异性”和 NG 示例中的“前后左右全都是同性”两种情况。问题中所要求满足的“上述条件”是什么呢？仅从上下文来看第一感当然是说“前后左右的座位全部都是异性”。但是仔细一想就知道这个不对，每个座位的“前后左右的座位都是异性”的安排情况只有两种。

“前后左右的座位全部都是异性”和“前后左右全都是同性”的两种极端情况之间还有巨大的灰色空间。问题的原意应该是指满足“任何一个座位的前后左右不全是同性（或至少有一个异性）”的情况吧？

68.2 解题分析

先考虑一个基本方案。

从搜索方式来说，这个问题与前面的 Q32 和 Q59 等题有类似之处。只需要基于 Q32 或 Q59 的基本框架略作修改即可。

算法流程如下：

Seats 初始化及加固栏处理(外加一圈格子以方便判断)

Count = 0

Def search(h,w, boy, girl):

If h == H+1: #说明已经到达最下面的围栏行，找到一个“不违规”的安排

If boy == girl: #男生数和女生数检查

Count = count + 1

Elseif w == W+1: #到达最右边的围栏列，换到下一行继续扫描

Search(h+1,w, boy, girl)

Else: #给当前座位安排人，只有两种情况：男生或女生

#试着安置男生到该座位

Seats[h,w] = 1

判断安置男生后是否有违规情况出现，如果没有则继续探索右边一个座位：

Search(h,w+1, boy+1, girl)

Seats[h,w] = 0 # 返回上一级的状态

#试着安置女生到该座位

Seats[h,w] = 2

判断安置女生后是否有违规情况出现，如果没有则继续探索右边一个座位：

```
Search(h,w+1, boy, girl+1)  
Seats[h,w] = 0 # 返回上一级的状态
```

要点说明：

- (7) 与 Q32, Q59 一样，逐行扫描，每次向右移一格。右边到头即移到下一格。到达最下边的围栏行则表明已经完成一次搜索，找到一个合规的安排方案
- (8) “违规检查”只需要针对当前座位的左边座位和上边座位，这是因为扫描是向右、下方向进行的，当前座位的右边和下边还没有安排人，所以当前座位、右边座位和下边座位肯定还没有违规
- (9) 最后还必须满足男生和女生人数都必须相等

68.3 代码及测试

一如既往，给出一个功能正确的方案并不难，但是运行太慢了！所以这些问题的难点在于如何提高运行效率，比如说书中所提示的剪枝？容我再想一想如何剪枝。。。

69. Q69: 蓝白歌会

69.1 问题描述

至此，本书终于迎来了最后一个问题。请大家回想一下每年年终的“红白歌会”^①。为分出红组和白组的胜负，野鸟会^②的工作人员会负责统计观众举起的红色和白色卡片的张数，这一幕大多数日本人都很熟悉（由于本书印刷颜色的关系，这里改为蓝白歌会）。

如果蓝色和白色的卡片分布过于分散，就不便于统计。因此为使结果一目了然，需要对人群进行移动调整（如果蓝色和白色人数相等，则最终两个人群之间会形成纵向或横向的分界线）。不过每次只能移动两人，并且只能是纵向或横向相邻的两人交换位置。

反复交换位置，使人群最终变为如图27①中“终止状态”的4种状态之一。求以最少步骤由起始状态变为4种终止状态之一时，移动次数最多的起始状态。

举个例子，假设有 4×4 ，即16个人，以8人为一组分为两组。他们的起始状态如②所示时，移动次数为8；起始状态如③所示时，移动次数为10。这种情况下，移动次数最多为10，并且像③这样的起始状态有64种（蓝色和白色对换，以及左右对换等情况也算不同的状态）。

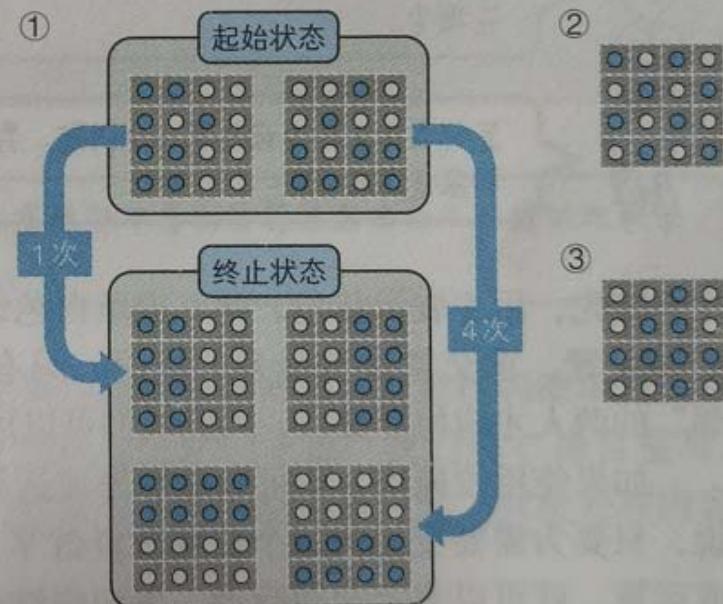


图27 蓝色和白色的移动示例

问题：当有 $4 \times 6 = 24$ 个人，以12人为一组分为两组时，求所需移动次数最多的起始状态有多少种？

69.2 基本思路

由于终止状态是确定的（4个），所以适合于逆向搜索（本系列前面出现过不少逆向思维解决的问题例）。

位置交换动作是可逆的，从 4 个确定性的终止状态开始，通过合理的位置交换操作（确保距离最小化）寻找距离 4 个确定性的终止状态最远的初始状态。这个显然是图搜索中最大距离问题，适合于用广度优先搜索算法。

另外，由于本题要求不仅找出最大距离，还要求找出满足这一条件的初始状态的个数，所以相当于要找出再广度优先搜索中出现在最远一层的所有状态的个数。

69.3 算法流程

基于广度优先搜索的算法流程如下所示：

初始化：

4 个出发状态(即原问题的终止状态)的初始化

`q, visited` 分别用作队列和存储已访问节点

`step = 0`

将 `start1, start2, start3, start4` 加入队列 `q`

将 `start1, start2, start3, start4` 加入 `visited`

`while (q 非空):`

`curState, curStep = q.pop()`

 遍历 `curState` 每个格子：

 确认它和右侧格子是否可以交换，如果可以：

 交换得到 `nxtState`，判断 `nxtState` 是否在 `visited` 中，如果不在：

 将 `nxtState` 及 `(curStep+1)` 加入队列 `q`

 将 `nxtState` 加入队列 `visited`

 确认它和下侧格子是否可以交换，如果可以：

 交换得到 `nxtState`，判断 `nxtState` 是否在 `visited` 中，如果不在：

 将 `nxtState` 及 `(curStep+1)` 加入队列 `q`

 将 `nxtState` 加入队列 `visited`

`ansSteps = curStep`

搜索 `visited` 中距离等于 `ansSteps` 的状态个数（这意味着 `visited` 也要记录距离信息）

69.4 实现要点

当前人员状态用二维数组（本题解用 numpy 数组）表示，为了方便判断，与棋盘问题类似采用了外加围栏的方式。

如以上算法流程图中所述，由于最后需要最终层的所有状态个数，因此 `visited` 以字典的方式记录状态以及其对应的距离。

由于 numpy 数组不能用作 dict 的 key，所以，表示状态的 numpy 二维数组先变换为一维数组，然后转变为 tuple，用作 `visited` 的 key，而距离（层数, curStep）。存入队列时，则是将前述 `visited` 的 key 与距离（层数, curStep）组成一个嵌套式的 tuple 再存入。注意队列 `q` 与 `visited` 的这两种存储方式的区别。

在每个格子上查询是否可以交换时，只考虑它与右边一格以及下边一格交换的可能性。这样按行扫描下去，不会遗漏也不会产生不必要的重复。这个技巧也是在前面的题目中已经出现过。

[Reference]

[Revision history]