Python算法之旅（第2期）

上期回顾：

上一期课后思考的题目是“使用位运算来处理整数t的二进制数，穷举从t=0开始，每次递增1，直到t=2^n-1的过程中，t对应的种选择模式，从而找出最优解”。

我分别用穷举和回溯两种算法来实现这个功能（算法分析详见上一期内容），代码如下：

#穷举法：直接用位运算操作整数t的各个二进制数位，效率更高

def divide\_stones\_4(a, n):

total = sum(a)

half = total // 2

max\_s = 0

lib = tuple(map(lambda x: 1 << x, range(n-1,-1,-1)))#从高到低标记每个二进制位的1

for t in range(1<<n): #遍历从[0,1<<n]的所有选择模式

s = 0

for i in range(n):

if t & lib[i] > 0: #t的二进制数第i位是1，则选择该堆石子

s += a[i] #累计被选中的石子堆的数量

if max\_s < s <= half: #更新最优解

max\_s = s

return total - max\_s \* 2 #返回最小差值

#深搜算法：直接用位运算操作整数t的各个二进制数位，每层递归函数的选择模式都继承自上一层函数，每层只选择1堆石子

def divide\_stones\_5(a, n):

total = sum(a)

half = total // 2

max\_s = 0

lib = tuple(map(lambda x: 1 << x, range(n-1,-1,-1)))#从高到低标记每个二进制位的1

def dfs(t, s): #参数：t,正整数，其二进制数代表当前选择模式；s,正整数，表示已经选择的石子数量。

nonlocal max\_s

if s > max\_s: #更新最优解

max\_s = s

for i, bit in enumerate(lib):

if t & bit == 0: #只在第一个值为1的二进制位左侧设置1，以避免重复

if s+a[i] <= half: #已经超出半数就无需再选择新的石子堆了

t |= bit #将整数t的第i个二进制位设置成1

dfs(t, s+a[i])

t &= ~bit #将整数t的第i个二进制位恢复成0

else:

break

dfs(0, 0)

return total - max\_s \* 2 #返回最小差值

在穷举算法中，无论是用列表b来模拟一个整数t的n位二进制数，还是用位运算直接操作整数t的各个二进制位，都是非常巧妙的方法，每当遇到这种题目，都有令人眼前一亮的感觉。无独有偶，我有幸遇到了与“石子划分问题”很相似的一道题目，与大家共同思考。

题目：翻转棋子游戏

难度：3星 有趣：3星 有用：3星

分类：穷举，深搜，广搜

描述：翻转棋子游戏是这样玩的：

有一张4\*4的棋盘，在16个位置上每个位置放着一个棋子，棋子一面是黑色，另一面是白色，棋子或者白色面朝上，或者黑色面朝上。

游戏的走法如下：每一步先选择一个位置，然后把该位置和上，下，左，右（不越界）相邻位置上的棋子翻转（白->黑，黑->白）。

我们用一个长度为16的字符串board存储棋盘上每颗棋子的颜色状态，每颗棋子的颜色可以用 0、1 来表示，0 表示白，1 表示黑，顺序为从左至右，从上至下。

例如，board="1011000111110001"，表示的棋盘状态如下：

1010

0000

1101

1001

当我们选择第三行，第一列的位置翻转时，棋盘变化为:

1010

1000

0001

0001

游戏的目的是用最少的步数把全部棋子变为白色向上或黑色向上。

函数：flip\_chess(board)

参数说明：board -- 长度为16的字符串，存储了棋盘上每颗棋子的初始颜色状态。

返回值：返回最少的翻转次数，如果无法翻转成目标状态，则输出"impossible"。

样例1: board="1010000011011001"，返回"impossible"；

样例2: board="1001110110011000"，返回4。

算法分析：

初看本题，最容易想到的是穷举法，用包含16个元素的列表d分别表示每个棋子的翻转状态，d[i]=1表示翻转第i个棋子，d[i]=0表示不翻转。

可以穷举从d=[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]到d=[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]，即从都不翻转到都翻转。若把列表d对应整数t的16位二进制数，则相当于穷举从t=0到t=2^16-1。

我们可以直接用位运算操作整数t的各个二进制数位，从而生成整数t对应的棋盘翻转模式，判断该模式是否有解。若该翻转模式能获得更好的解，则更新最优解。

根据上述分析，我们可以写出代码：

#判断是否已经翻转成功:全部棋子变为白色向上或黑色向上

def check(a):

return all(a) or not any(a)

#翻转棋盘a中位置i及其周围的棋子

def flip(a, i, size):

a[i] = not a[i] #翻转位置i的棋子

if i >= size:

a[i-size] = not a[i-size] #不是第一行则翻转上方棋子

if i < size\*(size-1):

a[i+size] = not a[i+size] #不是第size行则翻转下方棋子

if i % size > 0:

a[i-1] = not a[i-1] #不是第一列则翻转左方棋子

if (i+1) % size > 0:

a[i+1] = not a[i+1] #不是第size列则翻转右方棋子

#穷举法：直接用位运算操作整数t的各个二进制数位

def flip\_chess\_1(board):

a = list(map(int, list(board))) #将字符串转换成整数类型列表，以便于处理

lib = tuple(map(lambda x: 1 << x, range(len(a)-1,-1,-1)))#从高到低标记每个二进制位的1

min\_c = len(a) + 1 #初始化最小步数为最大值

for t in range(1<<len(a)): #遍历从[0,1<<len(a)]的所有翻转模式

b, c = a.copy(), 0 #复制棋盘a到b，c用来累计翻转棋子的数量

for i in range(len(lib)):

if t & lib[i] > 0: #t的二进制数第i位是1，则翻转位置i及其周围的棋子，并计数

flip(b, i, size)

c += 1

if check(b) and c < min\_c: #本翻转模式能获得更好的解，则更新最优解

min\_c = c

if min\_c <= len(a):

return min\_c

else:

return "impossible"

穷举法是容易想到的算法，但是效率实在太低，我们可以使用深度优先搜索（回溯加剪枝）来实现同样的功能，每层递归函数的棋盘都继承自上一层函数，每层只翻转1个棋子，为避免重复计算，我们只在第一个值为1的二进制位左侧设置1，如果没有更优解就剪枝，已经有解就不必再翻转更多棋子了，这样可以大幅度提高效率。代码如下：

#深搜法：直接用位运算操作整数t的各个二进制数位，每层递归函数的棋盘都继承自上一层函数，每层只翻转1个棋子

def flip\_chess\_2(board):

a = list(map(int, list(board))) #将字符串转换成整数类型列表，以便于处理

lib = tuple(map(lambda x: 1 << x, range(len(a)-1,-1,-1)))#从高到低标记每个二进制位的1

min\_c = len(a) + 1 #初始化最小步数为最大值

def dfs(t, c): #参数：t,正整数，其二进制数代表当前翻转模式；c，正整数，表示已经翻转的棋子数量

nonlocal min\_c

if check(a): #已经有解就不必再翻转更多棋子了

if c < min\_c:#本翻转模式能获得更好的解，则更新最优解

min\_c = c

else:

for i, bit in enumerate(lib):

if c+1 < min\_c and t & bit == 0: #如果没有更优解就剪枝，只在第一个值为1的二进制位左侧设置1，以避免重复

t |= bit #将整数t的第i个二进制位设置成1

flip(a, i, size)

dfs(t, c+1)

flip(a, i, size)#回溯

t &= ~bit #将整数t的第i个二进制位恢复成0

else:

break

dfs(0, 0)

if min\_c <= len(a):

return min\_c

else:

return "impossible"

更进一步思考，既然题目要求的是最优解，我们应该使用广度优先搜索才是效率更高的做法。把各种翻转模式存储到队列中，翻转棋子的数量从小到大依次加入队列。一开始设置翻转棋子数为0，然后逐渐增加翻转棋子数量（通过按位或运算实现）。

为避免翻转模式重复入队列，常见的算法是设置一个长度为2^16的列表f，用来标记整数t是否出现过，先初始化f[i]=False，一旦整数i入队列，则设置f[i]=True；但是我没有使用这种方法，而是规定每次只在当前翻转模式的左侧增加翻转的棋子，可以遍历lib，将其元素与代表当前翻转模式的整数t依次进行按位或运算，直到结果等于t，相当于从左向右依次修改0为1，直到遇到1，这样可以确保只在原二进制数的左侧增加1个1。每次按位或运算的结果就是获得新的翻转模式，将其加入队列即可。因为翻转棋子的数量越来越多，故最早获得的解就是最优解。代码如下：

from queue import Queue #导入模块Queue，以便操作队列

#广搜法：直接用位运算操作整数t的各个二进制数位

def flip\_chess\_3(board):

a = list(map(int, list(board))) #将字符串转换成整数类型列表，以便于处理

lib = tuple(map(lambda x: 1 << x, range(len(a)-1,-1,-1)))#从高到低标记每个二进制位的1

q = Queue() #创建队列对象

q.put(0) #翻转棋子数为0

while not q.empty():

t = q.get()

b, c = a.copy(), 0 #复制棋盘a到b，c用来累计翻转棋子的数量

for i in range(len(lib)):

if t & lib[i] > 0: #t的二进制数第i位是1，则翻转位置i及其周围的棋子，并计数

flip(b, i, size)

c += 1

if check(b): #因为翻转棋子的数量越来越多，故最早获得的解就是最优解

return c

else:

for i in range(len(lib)):

if t & lib[i] == 0: #从左向右依次修改0为1，直到遇到1，这样可以确保只在原二进制数的左侧增加1个1

q.put(t | lib[i])#将整数t的第i个二进制位设置成1后入列

else:

break

return "impossible" #无解则返回"impossible"

最后是主函数部分，我们从文件fzqz.txt中读取数据，每行数据表示棋盘上每颗棋子的初始颜色状态，一个示例如下：

1010000011011001

1001110110011000

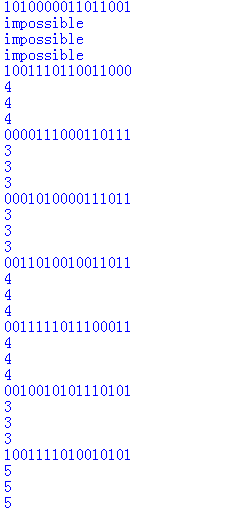
0000111000110111

1001111010010101

0111000001110010

总共有5行（组）数据，存储了5个不同棋盘上每颗棋子的初始颜色状态。

对于每组数据，预想的答案分别是：impossible，4，3，5，2。



size = 4 #4\*4的方阵

with open('fzqz.txt', 'r') as fin:

for line in fin.readlines():

a = line.strip()

print(a)#依次读取每行

print(flip\_chess\_1(a))

print(flip\_chess\_2(a))

print(flip\_chess\_3(a))

至此，我们就介绍了解决石子划分问题的三类算法：穷举算法，深搜算法和广搜算法。这三类算法由浅入深，效率逐步提升，同时思维也越发抽象，需要越来越多的学习和思考才能理解和掌握。

本题的各种算法也可以用c++语言来实现，两种语言相比较，我发现由于Python语言的自动回收内存机制和函数式编程特征，以及丰富的内置函数，使得编程者无需考虑过多的实现细节，能更关注算法思想本身，代码也更简洁，更优雅。

课后思考：

在上述算法中，我们直接用位运算操作整数t的各个二进制数位，位运算的效率是比较高的，但如果不明白位运算原理，是很难理解相关代码的。回想上一期的内容，我们用列表b来模拟一个整数t的n位二进制数，然后直接操作列表b，也可以实现相同的功能。

聪明的你请开动脑筋，写出用列表b来模拟整数t的代码吧！

另外，如果你有更 Pythonic（优雅的、地道的、整洁的）代码，或者与本文不同的算法思路和代码实现，请你一定留言或联系我，让我们一起讨论，共同进步。