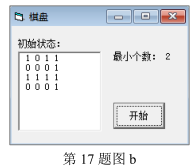
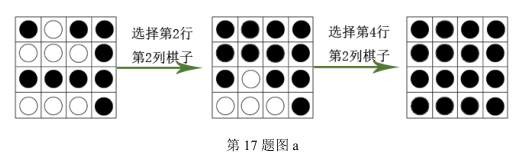
**从VB到Python之翻转棋子游戏**

余姚二中 梁见斌

VB代码：

17．【加试题】（2018学年第一学期9+1高中联盟期中考）小明在玩翻转棋盘的游戏，游戏规则是在一个 4╳4 的棋盘中挑选棋子（棋子只有黑白两面），对于每颗被挑选的棋子，要将该棋子和它在棋盘上的上下左右 4 个相邻位置上的棋子一起翻转，若某个方向上无棋子，则不作处理，最后使得棋盘纯色（纯白或纯黑）。

如第 17 题图 a 所示的棋盘，需要选择第 2 行第 2 列、第 4 行第 2 列两颗棋子，按照规则进行翻转便可使得棋盘变为纯黑。现编写程序找出实现棋盘纯色所需翻转棋子次数最少的方案并输出所挑选棋子的个数，若无答案则输出“无法翻转为纯色”。

每颗棋子的颜色状态可以用 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，第 17 题图 a 中的棋子初始状态可以表示为 1011000111110001。

每颗棋子的被选状态也可以用 0、1 进行标记，0 表示不选中，1 表示被选中。例如，一个16位的二进制串 0000010000000100 就表示该棋盘中的第 2 行第 2 列、第 4 行第 2 列这两个棋子被选中，如果将该二进制串转化为十进制数，即 2^10 +2^2 =1028。整张棋盘的所有选棋子方案为 0000000000000000~1111111111111111，也就是十进制数 0~65535，利用枚举算法在0~65535 之间枚举，即可找到最优方案。

程序运行界面如第 17 题图 b 所示，请回答下列问题。

（1）若某方案用十进制表示为 137，则该方案总共选择了 个棋子。

（2）实现上述功能的程序代码如下，请将划线处的代码补充完整。

'数组 a 储存棋盘原状态，数组 b 储存翻转后的棋盘状态

Dim a(1 To 16) As Integer, b(1 To 16) As Integer, minc As Integer

Function check() As Boolean '判断棋盘是否纯色

Dim flag As Boolean, i As Integer

flag = True

For i = 1 To 15

If b(i) <> b(i + 1) Then flag = False

Next i

check = flag

End Function

Private Sub Command1\_Click()

Dim k As Integer, c As Integer, i As Long, j As Long

minc = 17

For i = 0 To 2 ^ 16 - 1

For j = 1 To 16

b(j) = a(j)

Next j

k = 16

c = 0

j = i

Do While j > 0

If ① Then

b(k) = 1 - b(k)

If k > 4 Then b(k - 4) = 1 - b(k - 4)

If k < 13 Then b(k + 4) = 1 - b(k + 4)

If k Mod 4 <> 0 Then b(k + 1) = 1 - b(k + 1)

If ② Then b(k - 1) = 1 - b(k - 1)

c = c + 1

End If

j = j \ 2

k = k - 1

Loop

If ③ Then minc = c

Next i

If minc = 17 Then Label2.Caption = "无法翻转为纯色" Else Label2.Caption = Label2.Caption + Str(minc)

End Sub

Private Sub Form\_Load()

'生成棋盘状态，用数组 a(1)—a(16)表示，代码略

For i = 1 To 16

s = s + Str(a(i))

If i Mod 4 = 0 Then List1.AddItem s: s = ""

Next i

End Sub

答案：（1）j Mod 2 = 1 (2分)

（2）(k - 1) Mod 4 <> 0 (2分)

（3）check() = True And c < minc (2分)

算法分析：从题目的描述来看，本题采用了枚举法，用一个16位的二进制串表示棋盘中某些位置的棋子被选中，例如，二进制串 0000010000000100 就表示该棋盘中的第 2 行第 2 列、第 4 行第 2 列这两个棋子被选中，如果将该二进制串转化为十进制数，即 2^10 +2^2 =1028。整张棋盘的所有翻转棋子方案为 0000000000000000~1111111111111111，也就是十进制数 0~65535，利用枚举算法在0~65535 之间枚举，即可找到最优方案。

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。

对应Python代码：

#算法1：穷举法解翻转棋子游戏，直接用位运算操作整数t的各个二进制数位，效率较高

def exhaustion\_1(a):

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

return min\_c

#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

#主函数部分：从文件中读取多行数据，并逐行处理

size = 4 #4\*4的方阵

lib = tuple(map(lambda x: 1 << x, range(size\*\*2-1,-1,-1)))#从高到低标记每个二进制位的1

min\_cc = len(lib)+1

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

for line in fin.readlines():

print(line.strip())#依次读取每行

a = list(map(int, list(line.strip())))

min\_c = exhaustion\_0(a) #穷举法解翻转棋子游戏

if min\_c <= len(a):

print(min\_c)

else:

print("impossible")

算法1直接用位运算操作整数t的各个二进制数位，效率较高，如果不明白位运算规则，是很难理解的。

我们可以采用一个折中的方法来帮助大家理解该算法，即在代码中展示把整数t转换成n位二进制数，并存储到列表d的过程。d[i]=1表示翻转第i个棋子，d[i]=0表示不翻转。代码如下：

#算法2：穷举从t=0到t=2^n-1的过程，把整数t的n位二进制数存储到列表b ，并根据列表d的值来确定翻转模式

def exhaustion\_2(a):

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

d = binary\_number(t, len(a)) #将整数t的16位二进制数存储到列表d

for i in range(len(d)):

if d[i] == 1: #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

return min\_c

#将整数t的16位二进制数存储到列表d

def binary\_number(t, n):

d =[0] \* n #高位补零，凑足n位

i = n -1

while t > 0:

d[i] = t & 1 #相当于t % 2

i, t = i - 1, t >> 1 #相当于t // 2

return d

算法2虽然便于理解，但把整数t转化成二进制数并存储到列表d的操作比较耗费时间。我们可以穷举从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。根据二进制加法的原理，我们可以直接修改列表d的值来模拟整数t递增的过程，这样就不需要引入整数t了，数据结构也很清晰。代码如下：

##算法3：穷举法，根据二进制加法的原理，直接修改列表d的值来模拟整数t递增的过程

def exhaustion\_3(a):

min\_c = len(a) + 1 #初始化最小步数为最大值

d =[0] \* len(a) #初始化所有的位都是0

while True: #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(d)):

if d[i] == 1: #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

i = len(d) - 1

while i >= 0 and d[i] == 1: #修改列表d的值来模拟整数t递增的过程，每次递增1

d[i] = 0

i -= 1

if i >= 0: #将d[i]右侧的1都改成0，d[i]改成1，相当于二进制数递增1

d[i] = 1

else: #已经递增到最大值，即所有的二进制位均为1

break

return min\_c

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。

类比算法1和算法3，我们分别可以采用直接操作列表b和整数t两种不同数据结构来实现算法。代码如下：

#深搜法解翻转棋子游戏：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

#参数介绍：a——列表，存储当前棋盘状态；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。

def dfs\_1(a, t, c):

global min\_cc

if check(a): #已经有解就不必再翻转更多棋子了

if c < min\_cc:#本翻转模式能获得更好的解，则更新最优解

min\_cc = c

else:

for i in range(len(lib)):

if c+1 < min\_cc and t & lib[i] == 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 |= lib[i] #将整数t的第i个二进制位设置成1

flip(a, i, size)

dfs\_1(a, t, c+1)

flip(a, i, size)#回溯

t &= ~lib[i] #将整数t的第i个二进制位恢复成0

else:

break

#深搜法解翻转棋子游戏：使用列表d存储当前翻转模式，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

#参数介绍：a——列表，存储当前棋盘状态；d——列表，存储当前翻转模式；c——正整数，表示已经翻转的棋子数量。

def dfs\_2(a, d, c):

global min\_cc

if check(a): #已经有解就不必再翻转更多棋子了

if c < min\_cc:#本翻转模式能获得更好的解，则更新最优解

min\_cc = c

else:

for i in range(len(d)):

if c+1 < min\_cc and d[i] == 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

d[i] = 1

flip(a, i, size)

dfs\_2(a, d, c+1)

flip(a, i, size) #回溯

d[i] = 0 #回溯

else:

break

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。

当然，除了使用位运算直接操作整数t，也可以采用直接操作列表b的方式，这两种方式的代码实现分别如下：

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

#广搜法解翻转棋子游戏：直接用位运算操作整数t的各个二进制数位，效率更高

def bfs\_1(a):

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 len(a)+1 #无解则返回len(a)+1

#广搜法解翻转棋子游戏：使用列表d存储当前翻转模式，较为直观

def bfs\_2(a):

q = Queue() #创建队列对象

d =[0] \* len(a) #初始化所有的位都是0

q.put(d) #翻转棋子数为0

while not q.empty():

d = 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(d)):

if d[i] == 1: #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(d)):

if d[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

t = d.copy()

t[i] = 1

q.put(t) #生成列表d的拷贝，并将其第i个二进制位设置成1后入列

else:

break

return len(a)+1 #无解则返回len(a)+1

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。