**从VB到Python之石子划分问题**

余姚二中 梁见斌

VB代码：

17.【加试题】给出n堆石子，以及每堆石子数。请将它们分为两堆，使得这两堆的总石子数差最小。输入n，以及每堆石子数，输出分为两堆后的最小差值。比如，n＝4，四堆石子分别有13，6，8，14颗，则可以分为13＋8和14＋6的两堆，它们的最小差为1。

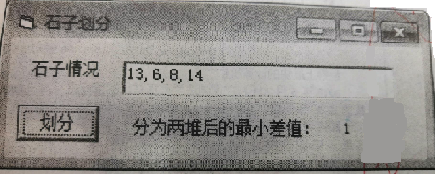
分堆算法为

（1）求得所有石子数total，以及它的一半half；

（2）在所有石子堆中作适当选择，对每种选择方案，求不超过half的己选中堆中的石子总数的最大值mx。所求即为（total－max）－max。

（3）以a(j)表示第j堆石子数；以b(j)表示第j堆石子是否被选中，如果b(j)＝1，表示第j堆被选中，如果b(j)＝0表示第j堆没有被选中。

（4）各种方案的表达及次序如下：以00…00（均不选中），00…01（只选中第n堆石子），00…10（只选中第n－1堆石子），00…11（选中第n－1堆和第n堆石子），00…100（选中第n－2堆石子），00…101（选中第n－2堆和第n堆石子），11…11（选中所有n堆石子）。

小李依据上述描述设计如下VB程序（程序运行界面如图所示）。请回答下列问题：

（1）四堆石子分别有22,36,15,3颗，则分为两堆后它们的最小差值为 。

（2）请在划线处填入合适的代吗  
Const maxn = 20

Dim n As Integer

Dim a(1 To maxn) As Integer

Dim b(0 To maxn) As Integer

Private Sub Form\_Load()

'读取石子数据，存储在数组a中，石子堆数存储在变量n中，代码略

End Sub

Private Sub Command1\_Click()

Dim i As Integer, j As Integer

Dim total As Integer, half As Integer, sum As Integer, max As Integer

total = 0

For i = 1 To n

total = total + a(i)

Next i

half = total / 2

max = 0

For i = 1 To n

b(i) = 0

Next i

i = n

Do While i > 0

sum = 0

For j = 1 To n

sum = ①

Next j

If ② Then max = sum

i = n

Do While i > 0 And b(i) = 1

i = i - 1

Loop

If i > 0 Then

b(i) = ③

For j = i + 1 To n

b(j) = 0

Next j

End If

Loop

Label2.Caption = Str(total - max - max)

End Sub

答案：（1）sum + b(j) \* a(j) (2分)

（2）sum > max And sum <= half (2分)

（3）1 (2分)

算法分析：从题目的描述来看，本题采用了穷举法，用包含n个元素的列表b分别表示堆石子的选择状态，b[i]=1表示选择第i堆石子，b[i]=0表示不选择。可以穷举从b=[0,0,...0,0]到b=[1,1,...1,1]，即从都不选择到都选择。

若把列表b的值对应整数t的n位二进制数，则相当于穷举从t=0到t=2^n-1。循环遍历每一个t，将t的n位二进制数存储到列表b；然后遍历b，若b[i]=1则选择第i堆石子；

完成本选择模式后，判断是否有解，若本选择模式能获得更好的解，则更新最优解。

题目所给的算法直接修改列表b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，数据结构很清晰。

对应Python代码：

#算法1：穷举法解石子划分问题，直接修改列表b的值来模拟整数t递增的过程

def divide\_stones\_1(a):

n, max\_s = len(a), 0

b =[0] \* n #初始化所有的位都是0

i = n - 1 #先选择第n堆

while i >= 0: #遍历从[0,1<<len(a)]的所有选择模式

s, b[i] = 0, 1 #将b[i]右侧的1都改成0，b[i]改成1，相当于二进制数递增1

for i in range(len(b)):

s += a[i] \* b[i] #累计被选中的石子堆的数量

if s > max\_s and s <= half: #更新最优解

max\_s = s

i = n - 1

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

b[i] = 0

i -= 1 #当b[i]==0时跳出循环，并将b[i]改成1，相当于二进制数递增1

return total - max\_s \* 2 #返回最小差值

#主函数部分：从文件中读取多行数据，并逐行处理

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

for line in fin.readlines():

print(line.strip())#依次读取每行

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

total = sum(a)

half = total / 2

print(divide\_stones\_1(a))

算法1直接修改列表b的值来模拟整数t递增的过程，这样虽然不需要引入整数t，累计被选中的石子堆的数量时也很方便，但是模拟整数t递增的过程算法较为抽象，如果不明白二进制加法原理，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b的过程。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

def divide\_stones\_2(a):

n, max\_s = len(a), 0

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

b = binary\_number(t, n) #将整数t的n位二进制数存储到列表b

s = 0

for i in range(n):

s += a[i] \* b[i] #累计被选中的石子堆的数量

if s > max\_s and s <= half: #更新最优解

max\_s = s

return total - max\_s \* 2 #返回最小差值

#将整数t的n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b的操作比较耗费时间。

我们可以使用位运算来处理整数t的二进制数，这样就无需引入列表b，可以直接操作整数t了，效率有所提升。代码如下：

#算法3：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\_3(a):

n, max\_s = len(a), 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<<len(a)]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 s > max\_s and s <= half: #更新最优解

max\_s = s

return total - max\_s \* 2 #返回最小差值

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，每层递归函数的选择模式都继承自上一层函数，每层只选择1堆石子，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，利用已经超出半数就无需再选择新的石子堆来进行剪枝，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：

#深搜法解石子划分问题：使用列表b存储当前翻转模式，每层只选择1堆石子

#参数介绍：a——列表，存储各堆石子数量；b——列表，存储当前选择模式；s——正整数，表示已经选择的石子数量。

def dfs\_1(a, b, s):

global max\_s

if s > max\_s: #更新最优解

max\_s = s

for i in range(len(b)):

if b[i] == 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

b[i] = 1 #将整数t的第i个二进制位设置成1

dfs\_1(a, b, s+a[i])

b[i] = 0 #将整数t的第i个二进制位恢复成0

else:

break

#深搜法解石子划分问题：直接用位运算操作整数t的各个二进制数位

#参数介绍：a——列表，存储各堆石子数量；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。

def dfs\_2(a, t, s):

global max\_s

if s > max\_s: #更新最优解

max\_s = s

for i in range(len(lib)):

if t & lib[i] == 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 |= lib[i] #将整数t的第i个二进制位设置成1

dfs\_2(a, t, s+a[i])

t &= ~lib[i] #将整数t的第i个二进制位恢复成0

else:

break

穷举法是最基本的算法思想，利用二进制数表示大整数和位运算规律，是本题闪光点。绝大多数穷举算法都可以转化成回溯算法，利用回溯和剪枝可以简化代码，提高效率

进一步思考：当石子堆数量n太大时，对应的n位二进制数太大了，用穷举法和回溯法都不适合；因为我们对每个石子堆的操作都是选择或者不选择，所以本题可以当做一个0-1背包问题来解决（当然石子的总数不能太大，否则需要太大的空间），我们可以把背包的容量用石子总量的一半half来代替，物品的质量和价值都用每堆石子的数量来代替，最大价值就是不超过half的石子数量，这样就可以套用0-1背包问题的基本模型来解决本题。

我们先使用与回溯算法关系较为紧密的记忆化搜索算法来实现。代码如下：

#记忆化搜索（备忘录算法）求0-1背包问题，b[n][c]初始化为-1

def dfs\_3(n, c):

global b

if b[n][c] != -1:

return b[n][c]

max\_s = 0

if n == 1: #处理只给定了1堆石子的情形

if c >= a[n-1]:

max\_s = a[n-1]

else:

if c < a[n-1]: #若装不下，则不装第n堆石子

max\_s = dfs\_3(n-1, c)

else: #如果装得下，从装和不装两者中取最大值

max\_s = max(dfs\_3(n-1, c), dfs\_3(n-1, c-a[n-1]) + a[n-1])

b[n][c] = max\_s

return b[n][c]

记忆化搜索算法使用了递归，属于自顶向下的思考方法，我们可以反其道而行之，使用自底向上的思考方法，即动态规划算法来实现。我们先使用二维列表来实现它：

#动态规划：二维列表存储记录，b[i][j]初始化为0

def dp\_1(n, c):

b = [[0 for i in range(total+1)] for i in range(n+1)]

for i in range(1, n+1): #记录前i(1<=i<=n)堆石子装入容量为1-c的背包的最大价值

for j in range(1, a[i-1]): #背包容量不够，不能装下第i堆石子

b[i][j] = b[i-1][j]

for j in range(a[i-1], c+1): #背包容量足够，可以选择装或不装第i堆石子

b[i][j] = max(b[i-1][j], b[i-1][j-a[i-1]] + a[i-1])

return total - b[n][c] \* 2 #返回最小差值

二维列表的动态规划算法是直接从记忆化搜索算法中逆推过来的，它的优点是代码结构和状态转移方程一致对应，便于理解，缺点是耗费空间过大，我们可以对其进行降维优化。

常见的优化方法是用2个一维列表代替二维列表，代码如下：

#动态规划：使用2个一维列表代替二维列表，pre[j]和cur[j]均初始化为0

def dp\_2(n, c):

#pre[j]相当于dp\_1()中的b[i-1][j]，cur[j]相当于b[i][j]

pre = [0] \* (total + 1)

cur = [0] \* (total + 1)

for i in range(1, n+1): #记录前i(1<=i<=n)堆石子装入容量为1-c的背包的最大价值

for j in range(1, c+1): #背包容量不够或不装更好，不选择第i堆石子

if j < a[i-1] or pre[j] > pre[j-a[i-1]] + a[i-1]:

cur[j] = pre[j]

else: #背包容量足够且选择第i堆石子有更优解

cur[j] = pre[j-a[i-1]] + a[i-1]

for j in range(1, c+1): #复制上一行的数据到当前行

pre[j] = cur[j]

return total - cur[c] \* 2 #返回最小差值

更进一步的优化方法用一维列表存储记录，这也是经典的0-1背包算法框架。代码如下：

#动态规划：优化的动态规划算法，一维列表存储记录，f[j]初始化为0

def dp\_3(n, c):

f = [0] \* (total + 1)

for i in range(1, n+1): #记录前i(1<=i<=n)堆石子装入容量为1-c的背包的最大价值

for j in range(c, a[i-1]-1, -1): #须先求出列坐标j较大的F[j]，故让循环变量j的值从大到小递减

if f[j] < f[j-a[i-1]] + a[i-1]: #当(j < a[i-1] or f[j] >= f[j-a[i-1]] + a[i-1])时，f[j]的值不变

f[j] = f[j-a[i-1]] + a[i-1]

return total - f[c] \* 2 #返回最小差值

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，回溯算法和动态规划算法。在石子堆数较少，而每堆石子数量较多时，适合用前两种算法，反之则用动态规划算法。

Python语言的自动回收内存机制和丰富的内置函数，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。