**《算法设计与分析》实验报告**

**实验三 动态规划策略应用基础**

**学号：20211104040**

**姓名：曹文星**

**班级：软二**

**日期：4/11/2023**

**一、实验目的**

1、理解动态规划策略的基本思想。

2、了解适用动态规划策略的问题类型，并能利用动态规划策略设计相应的算法，解决具体问题。

3、掌握动态规划算法时间空间复杂度分析，以及问题复杂性分析方法。

**二、实验内容**

任务：按要求完成以下任务：

1、求解矩阵连乘问题。

要求:分别用自底向上的动态规划方法和自顶向下的备忘录方法计算最优值并构造最优解，通过实例比较两种方法的结果和效率。

2、求解最大子段和问题。

要求：分别用教材所给的三种方法求解（简单方法、分治法、动态规划），通过实例比较结果和时间效率。

提交内容：算法的思想、程序源代码及其说明、实例运行结果及分析。

**三、设计分析**

1、求解矩阵连乘问题。

分别用自底向上的动态规划方法和自顶向下的备忘录方法计算最优值并构造最优解，通过实例比较两种方法的结果和效率,需要熟练使用以上方法解决问题,同时利用计数工具计算耗时.

2、求解最大子段和问题。

分别用教材所给的三种方法求解（简单方法、分治法、动态规划），通过实例比较结果和时间效率,需要熟练使用以上方法解决问题,同时利用计数工具计算耗时.

**四、算法描述及程序**

**一:矩阵连乘问题:**

Python代码解析

**FH.py**

# !自底向上的动态规划方法

# 从最小的子问题开始，逐步扩大子问题的规模，直到求解原问题

# 输入的数据流构建:

# 假设p是一个列表，存储了n个矩阵的维度信息

# p[0]是第一个矩阵的行数，p[1]是第一个矩阵的列数（也是第二个矩阵的行数,依此类推

# n是矩阵的个数

# 定义一个二维数组m，用来存储 最优值 ，m[i][j]表示计算矩阵Ai…Aj的最少数乘次数。

# 定义一个二维数组s，用来存储 最优划分点 ，s[i][j]表示计算矩阵Ai…Aj时，最优的划分位置k。

def FH(p, n): # 配置主函数,包含输出功能

print(matrix\_chain(p, n).index(1)) # 返回最优值

s = matrix\_chain(p, n).index(0)

print\_optimal(s, 0, len(s) - 1) # 放输出里面去

def matrix\_chain(p, n):

m = [[0 for i in range(n)] for j in range(n)]

s = [[0 for i in range(n)] for j in range(n)] # 创建两个n\*n的二维数组m和s，初始化为0

# 动态规划算法

for k in range(2, n + 1): # k表示子问题的规模

for i in range(1, n - k + 2): # i表示子问题的起始位置

j = i + k - 1 # j表示子问题的终止位置

m[i][j] = float('inf') # 初始化最小代价为无穷大

for k in range(i, j): # k表示断开位置

q = m[i][k] + m[k + 1][j] + p[i - 1] \* p[k] \* p[j] # 计算当前断开位置的代价

if q < m[i][j]: # 如果当前代价小于之前的最小代价，更新最小代价和断开位置

m[i][j] = q

s[i][j] = k

return m[1][n], s

def print\_optimal(s, i, j):

if i == j:

print("A" + str(i), end="") # 如果只有一个矩阵，则直接输出

else:

# 输出左括号

print("(", end="")

# 输出左边的子问题的最优解

print\_optimal(s, i, s[i][j])

# 输出右边的子问题的最优解

print\_optimal(s, s[i][j] + 1, j)

# 输出右括号

print(")", end="")

**HF.py**

# !自顶向下的备忘录方法

# 从原问题开始，递归地求解子问题，并将已经求解过的子问题的结果存储起来，避免重复计算

# 输入的数据流构建:

# 假设p是一个列表，存储了n个矩阵的维度信息

# p[0]是第一个矩阵的行数，p[1]是第一个矩阵的列数（也是第二个矩阵的行数,依此类推

# n是矩阵的个数

# 定义一个二维数组m，用来存储最优值，m[i][j]表示计算矩阵Ai…Aj的最少数乘次数。

# 定义一个二维数组s，用来存储最优划分点，s[i][j]表示计算矩阵Ai…Aj时，最优的划分位置k。

# ----思路----

# 初始化m[i][j]为一个特殊值（如-1），表示该子问题还未求解过。

# 定义一个递归函数matrix\_chain(i,j)，用来求解子问题m[i][j]和s[i][j]。

# 在函数中，如果m[i][j]不等于特殊值，说明该子问题已经求解过，直接返回m[i][j]。

# 否则，如果i等于j，说明单个矩阵不需要计算，将m[i][j]设为0，并返回0。

# 否则，遍历所有可能的划分位置k，比较不同划分方式的数乘次数，并记录最小值和对应的k。

# 将m[i][j]设为最小值，并返回该值。

# 调用matrix\_chain(1,n)作为最优值，并根据s数组构造最优解。

def HF(p, n): # 配置主函数,包含输出功能

print(matrix\_chain(p, n).index(1)) # 返回最优值

s = matrix\_chain(p, n).index(0)

print\_optimal(s, 0, len(s) - 1)

def matrix\_chain(p, n):

m = [[0 for i in range(n)] for j in range(n)]

s = [[0 for i in range(n)] for j in range(n)] # 创建两个n\*n的二维数组m和s，初始化为0

matrix\_chain\_aux(p, m, s, 1, n) # 调用递归函数来计算m[1][n]和s[1][n]

return m[1][n], s # 返回最优值和最优解

def matrix\_chain\_aux(p, m, s, i, j): # 递归计算m[i][j]和s[i][j]

# 两个边界条件(递归返回条件)

if i == j: # 如果i=j，那么m[i][j]=0，单个矩阵不需要计算

m[i][j] = 0

return m[i][j]

if m[i][j] > 0: # 另外如果m[i][j]不为0，说明已经计算过了，直接返回

return m[i][j]

m[i][j] = float('inf') # 将m[i][j]设为infinity(无穷大)

for k in range(i, j): # 遍历所有可能的划分位置,k

q1 = matrix\_chain\_aux(p, m, s, i, k) # 计算左边的子问题的最优值

q2 = matrix\_chain\_aux(p, m, s, k + 1, j) # 计算右边的子问题的最优值

q = q1 + q2 + p[i - 1] \* p[k] \* p[j] # 计算当前划分的代价

if q < m[i][j]: # update m[i][j]和s[i][j]

m[i][j] = q

s[i][j] = k

return m[i][j]

def print\_optimal(s, i, j): # 解释同前

if i == j:

print("A" + str(i), end="")

else:

print("(", end="")

print\_optimal(s, i, s[i][j])

print\_optimal(s, s[i][j] + 1, j)

print(")", end="")

**Test.py**

# -\*- coding:gbk -\*-

from FoottoHead import matrix\_chain as FH

from HeadtoFoot import matrix\_chain as HF

# 矩阵连乘问题

# test 3

p = [5, 10, 3, 12, 5, 50, 6]

n = len(p) - 1

FH(p, n)

HF(p, n)

p = [10, 20, 30, 40, 30]

n = len(p) - 1

FH(p, n)

HF(p, n)

p = [2, 3, 4, 5, 6]

n = len(p) - 1

FH(p, n)

HF(p, n)

# 一般来说，自底向上的方法更加简洁和高效，因为它避免了递归调用带来的开销。

# 自顶向下的方法则更加灵活和自然，因为它可以根据需要求解子问题。

**二:** 求解最大子段和问题

Python源代码及解析:

**Test.py**

# -\*- coding:gbk -\*-

# 最大子段和问题

from MaxStrSum.Simple import maxSubSum as SM1

from MaxStrSum.Divide import maxSubSum as SM2

from MaxStrSum.Dynamic import maxSubSum as SM3

from MaxStrSum.DynamicBetter import BettermaxSubSum as SM

a = [1, -2, 3, 5, -4, 6, 7, -3, 9] # 结果为3加到9,23

b = [1, -7, 3, -3, 8, -4, 6, 7, -8, 9] # 结果为8加到9,18

c = [6, -5, 3, -6, 5, -4, 7, 6, -7, -3, 5] # 结果为5加到6,14

print("蛮力", SM1(a))

print("分治", SM2(a))

print("动态", SM3(a))

print('------------')

print("蛮力", SM1(b))

print("分治", SM2(b))

print("动态", SM3(b))

print('------------')

print("蛮力", SM1(c))

print("分治", SM2(c))

print("动态", SM3(c))

print('------打印序列位子(直观位子)------')

print("Better", SM(a)) # 3,9

print("Better", SM(b)) # 5,10

print("Better", SM(c)) # 5,8

**Simple.py**

def maxSubSum(a): # 暴力法.

maxSum = 0

for i in range(len(a)): # i: 0->len

for j in range(i, len(a)): # j: i->len 选取的对象i之后的区域

thisSum = 0 # 记录当前的Sum

for k in range(i, j + 1): # k: i->j 切片i到j之间的字段

thisSum += a[k] # 不断地累加对应的值

if thisSum > maxSum: # 发现更大,则update

maxSum = thisSum

return maxSum

**Divide.py**

def maxCrossSum(a, left, mid, right):

sumNUm2 = sumNUm1 = leftSum = rightSum = 0 # 初始化计数器

for i in range(mid, left - 1, -1): # 中->左回退寻找拼接

sumNUm1 += a[i]

if sumNUm1 > leftSum:

leftSum = sumNUm1

for i in range(mid + 1, right + 1): # 中->右寻找寻找拼接

sumNUm2 += a[i]

if sumNUm2 > rightSum:

rightSum = sumNUm2

return leftSum + rightSum # 结合为左右最大之和

def maxS(a, left, right):

if left == right: # 左右碰在一起了,弹出左右对应元素更大的

return a[left] if a[left] > 0 else 0

mid = (left + right) // 2 # 划分子问题

leftSum = maxS(a, left, mid) # dive into them

rightSum = maxS(a, mid + 1, right)

crossSum = maxCrossSum(a, left, mid, right) # CrossPart

return max(leftSum, rightSum, crossSum) # 左中右三者取最大

def maxSubSum(a): # 分治法.

return maxS(a, 0, len(a) - 1) # 选取位置:头-尾.

**Dynamic.py**

def maxSubSum(a): # 动态规划

dp = [0] \* len(a) # 构建一维DP数组[0\*length]

maxSum = dp[0] = a[0] # 初始化dp首项&计数器

for i in range(1, len(a)): # i遍历项数,O(n)复杂度.

dp[i] = max(dp[i - 1] + a[i], a[i]) # 核心: 当前序列要么取之前的缺一个当前项的序列加上当前指向元素, 要么重新开始取,看哪个更大.

if dp[i] > maxSum: # 如果发现取到了更大的,记得更新max

maxSum = dp[i]

return maxSum

**DynamicBetter.py**

def BettermaxSubSum(a): # 加上记录器.

dp = [0] \* len(a) # 构建一维DP数组[0\*length]

maxSum = dp[0] = a[0] # 初始化dp首项&计数器

startsit = endsit = 0

for i in range(1, len(a)): # i遍历项数

dp[i] = max(dp[i - 1] + a[i], a[i]) # 核心: 当前序列要么取之前的缺一个当前项的序列加上当前指向元素, 要么重新开始取,看哪个更大.

if dp[i] == a[i]: # 发现重开了,则需要重新记录开始位置

startsit = int(i)+1

if dp[i] > maxSum: # 如果发现取到了更大的,记得更新max

maxSum = dp[i]

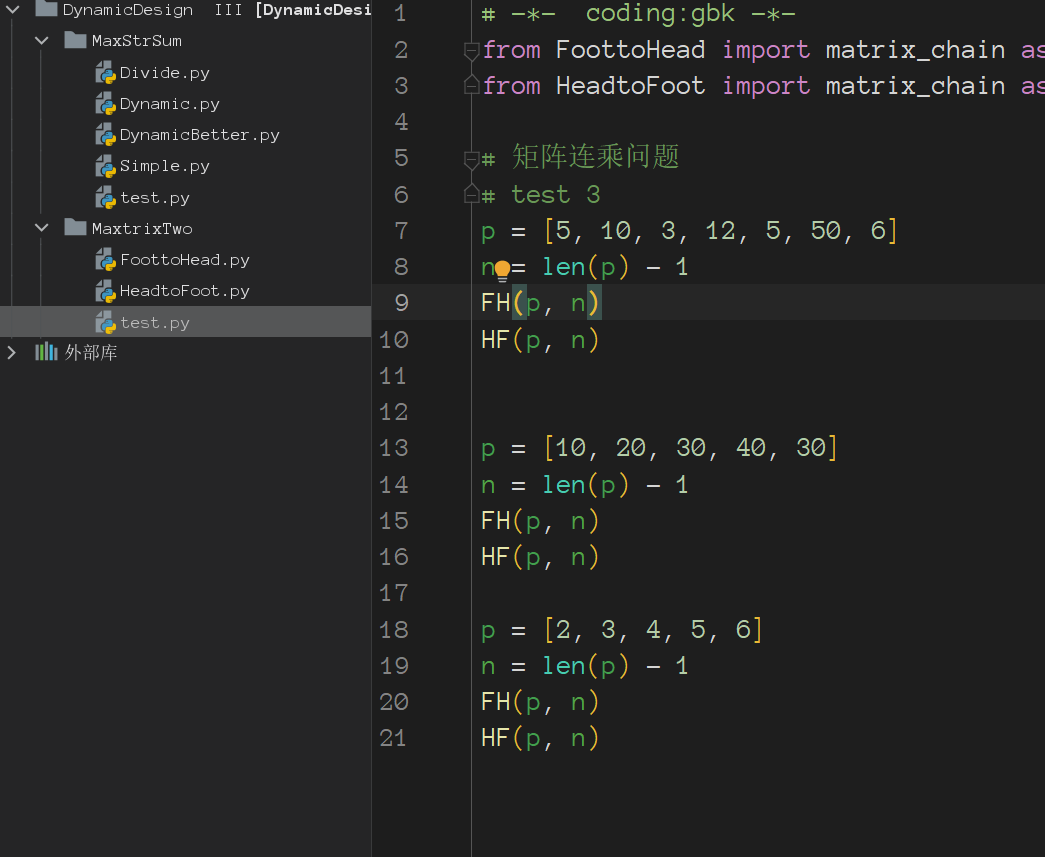
endsit = int(i)+1

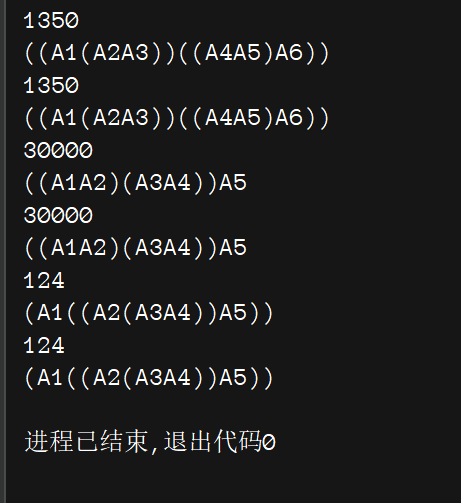
return maxSum, startsit, endsit

**五、测试与分析**

**问题一:**

工作面板:

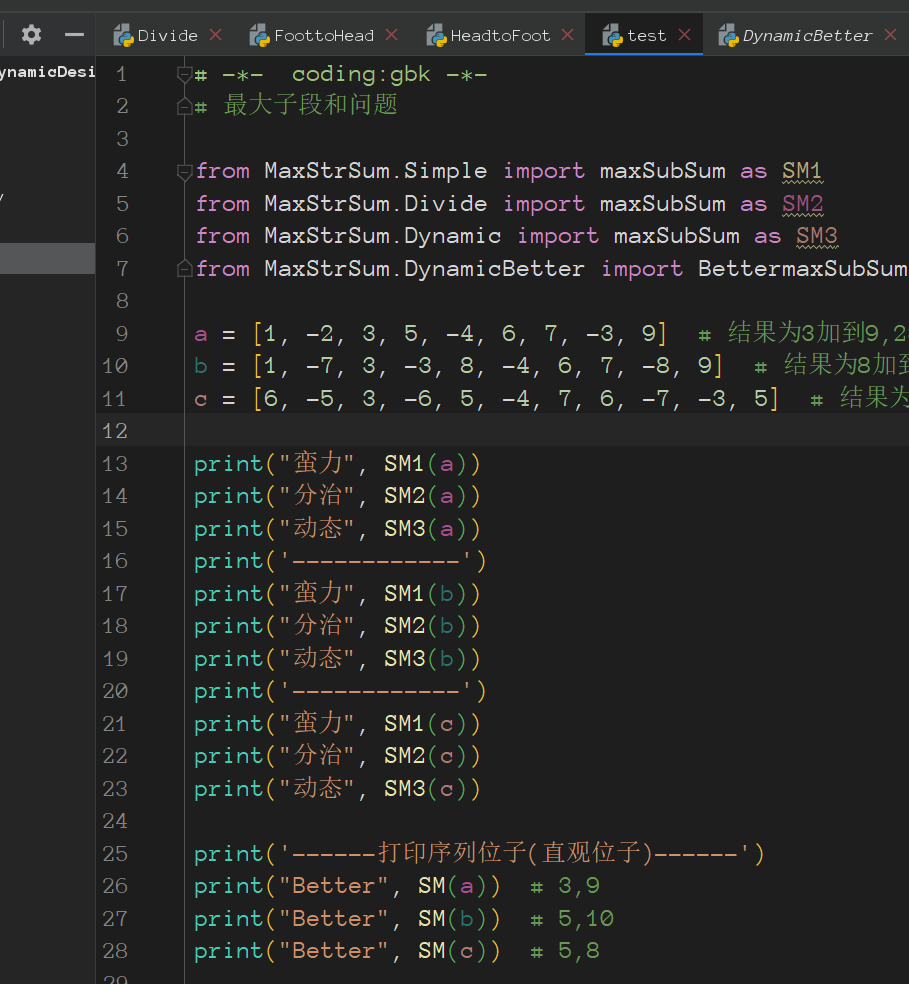




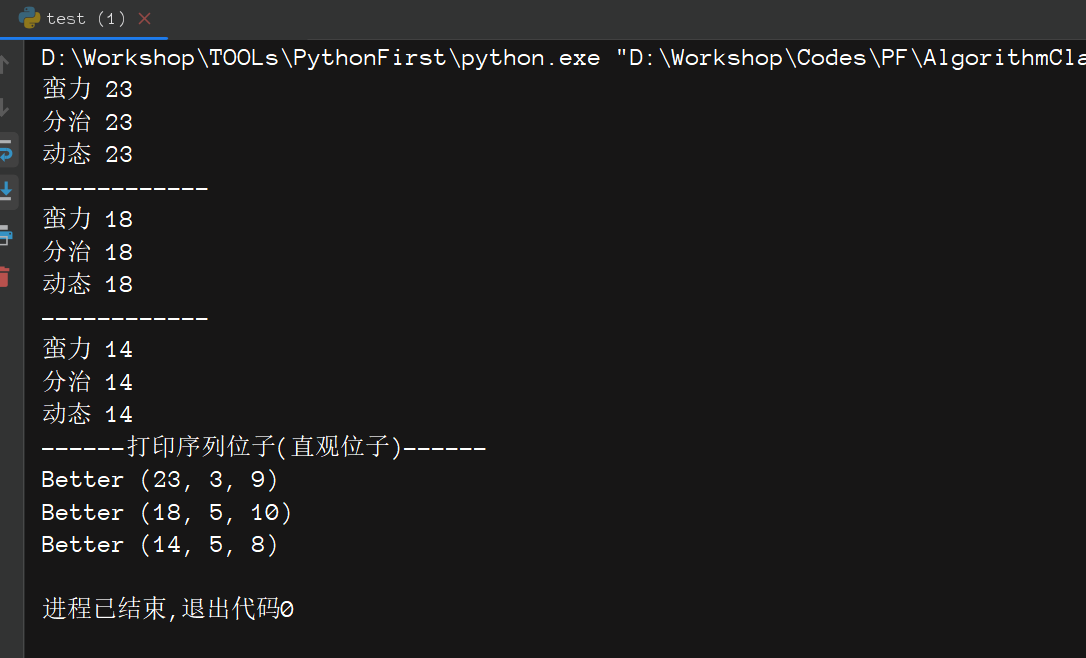
测试的输出结果

**问题二:**

工作面板:



非常抱歉,由于不可控因素,我的电脑没法给python正确的时间,时间模块失效.



运行结果:基本完成了要求

**六、实验总结与体会**

首先, 时间模块无法使用的问题还在询问, 解决有望.

体会到了动态规划的厚重, 另外, 我的python运行一些代码总是出问题, 看来要之后解决了