**华中科技大学计算机科学与技术学院**

**算法分析与设计报告**



专 业： 计算机科学与技术

班 级： 计算机ACM1701班

学 号： U201714780

姓 名： 刘晨彦

成 绩：

指导教师： 何琨

**完成日期： 2019年11月20日**

# 实验一

## 一、实验题目：矩阵链乘法算法的实现

## 二、实验目的与内容

1、实验目的：

本次实验的目的是：了解矩阵链乘法问题的内容，掌握动态规划算法的设计方法，利用动态规划算法解决矩阵链乘法问题，分析算法复杂度。

2、实验内容：

两人一组，不限编程语言，各自利用动态规划算法解决矩阵链乘法问题，分析复杂度、证明算法的正确性，共同完成PowerPoint进行展示交流。

## 三、算法设计

1、矩阵链乘法算法描述

矩阵链乘法程序描述：

STEP 1: 输入矩阵链

STEP 2：For from 0 to 矩阵数：

STEP 3：For from 0 to 矩阵数：

STEP 4：若为0：记录矩阵链中第到第个矩阵（时仅有一个矩阵）相乘时的最小次数为0，分割方法为第到第个矩阵相乘。

STEP 5：若为1：记录矩阵连中第到第个矩阵（时仅有两个矩阵）相乘时的最小次数为：第个矩阵的行数第个矩阵的列数第个矩阵的列数，分割方法为第到第个矩阵相乘。

STEP 6：若不等于0或1，For from to :

STEP 7：记录第到第个矩阵相乘、第到第个矩阵相乘和两者结果再次相乘所需的次数，分割方法为第到第个矩阵相乘，第到第个矩阵相乘。若该次得到的乘法次数比之前记录的乘法次数更少，则此次的乘法次数和分割方法覆盖上一次的记录。若循环结束进入STEP 8，否则返回STEP 6。

STEP 8：若循环结束进入STEP 9，否则返回STEP 3。

STEP 9：若循环结束，打印最小乘法次数，调用output()函数打印矩阵相乘的次序。否则返回STEP 2。

结果输出函数output(Start, End, record)描述：

STEP 1：若Start == End， 以字符串形式返回 'M'+str(Start)

STEP 2：若Start == End – 1，以字符串的形式返回 'M'+str(Start) + '\*' + 'M'+str(End)。

STEP 3：否则查询记录中从Start到End矩阵的分割k，以递归的方式返回：'(' + output(Start, k, record) + ')\*('+ output(k+1, End, record) + ')'。

2、矩阵链乘法算法流程图

算法描述的流程图如图1.1：

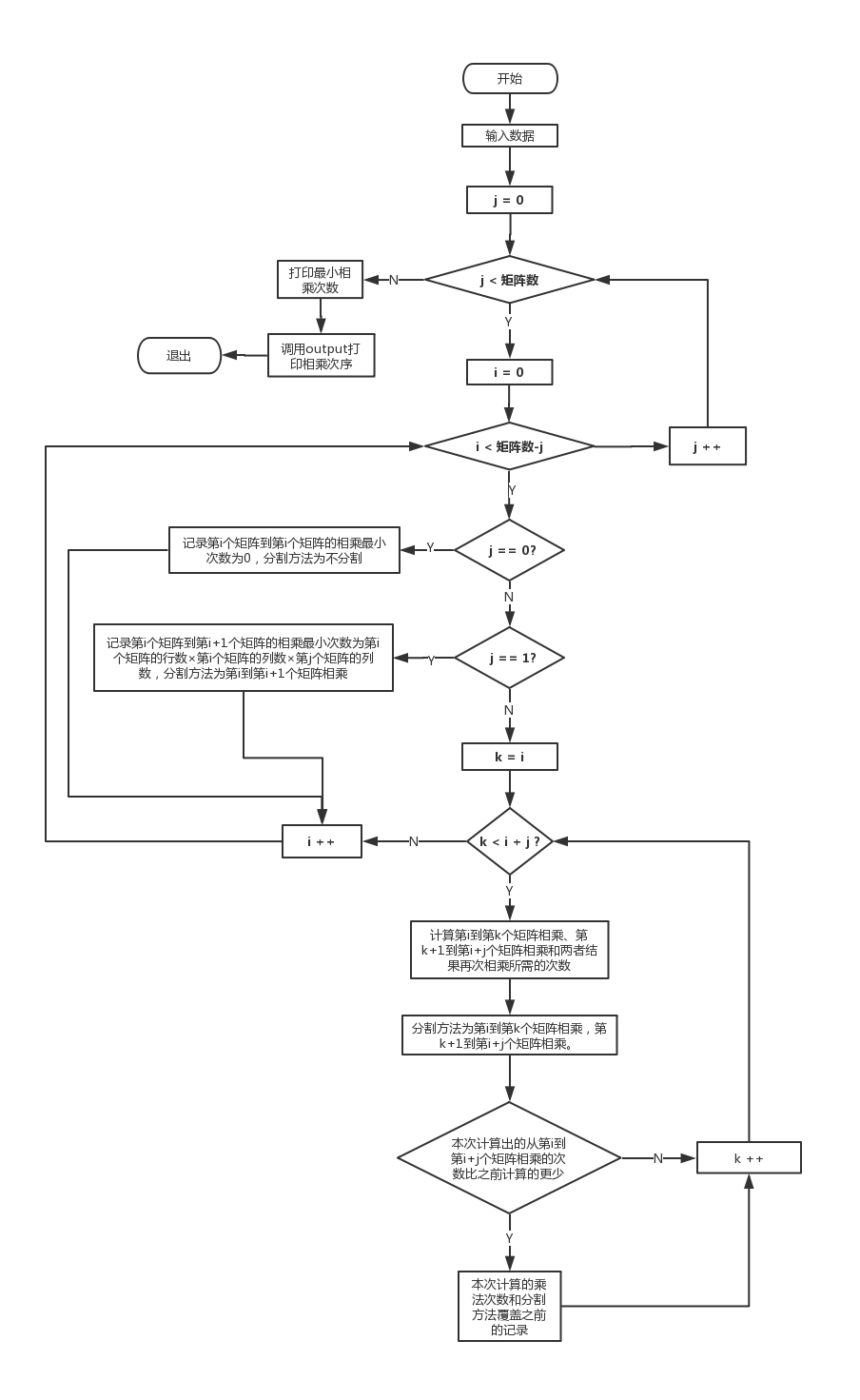


图1.1 矩阵链乘法算法流程图

## 四、实验环境

操作系统：Windows 10

编译环境：Sublime Text3

语言版本：python3.7.4

## 五、实验过程

矩阵链乘法算法的MST程序代码如下：

|  |
| --- |
| 矩阵链乘法算法： |
| def MCM(Start, End, MatrixChain, record):  for j in range(0, End - Start + 1):  for i in range(0, End - j + 1):  if j == 0:  record[(i, i + j)] = [i + j, 0]  elif j == 1:  record[(i, i + j)] = [i + j, MatrixChain[i][0] \* MatrixChain[i][1] \* MatrixChain[i + j][1]]  else:  interium = [0,0]  for k in range(i, i + j):  interium[0] = k  interium[1] = record[(i,k)][1] + record[(k + 1, i + j)][1] + MatrixChain[i][0] \* MatrixChain[k][1] \* MatrixChain[i + j][1]  if (i, i + j) not in record.keys():  record[(i, i + j)] = interium.copy()  else:  if record[(i, i + j)][1] > interium[1]:  record[(i, i + j)] = interium.copy()  print('At least have to multiply',record[(Start, End)][1], 'times')  print(output(Start, End, record)) |
| output函数 |
| def output(Start, End, record):  if Start == End:  return 'M'+str(Start)  elif Start == End - 1:  return 'M'+str(Start) + '\*' + 'M'+str(End)  else:  return '(' + output(Start, record[Start, End][0], record) + ')\*('+ output(record[Start, End][0]+1, End, record) + ')' |

## 六、算法测试

测试样例1

|  |  |  |  |
| --- | --- | --- | --- |
| 样例输入 | 设计理由 | 理论输出 | 样例输出 |
| [(10, 100), (100,5), (5,50)] | 检查算法是否正确 | At least have to multiply 7500 times  (M0\*M1)\*(M2) |  |

由于理论输出与样例输出相符，所以测试样例1验证成功。

测试样例2

|  |  |  |  |
| --- | --- | --- | --- |
| 样例输入 | 设计理由 | 理论输出 | 样例输出 |
| （随机生成的长度为500的矩阵链） | 检查算法性能 | 较快的时间内能够完成 |  |

由于理论输出与期望相符，所以测试样例2验证成功。

综上，算法通过所有样例的测试。

## 七、结果分析

1、矩阵链乘法算法正确性证明：

1）当只有1个矩阵时，显而易见算法正确

2）当只有2个矩阵时，根据算法可得最小乘法次数。

3）当有3个矩阵时，可以算法将矩阵链分成和两种情况并选择乘法次数最少的情况。根据1）和2）可知两个矩阵和一个矩阵相乘时均能获得正确结果，算法中同时计算了分割矩阵链后获得的两个矩阵再次相乘的次数，并于左右两个矩阵链乘法次数相加，符合要求，能够获得正确结果。

4）依次类推，可知有个矩阵时，将矩阵链分为长度为和的两个矩阵链，其中。由于任意的和都已在先前证明正确，故有个矩阵时正确。b t.2

2、矩阵链算法的时间复杂度证明

设矩阵链长度为，则根据算法可知复杂度：。

3、矩阵链空间复杂度证明

由算法可知，从任意第个()矩阵到第个矩阵()相乘的最小乘法次数都需要记录。故矩阵链算法的空间复杂度为。

## 八、总结

本次实验首先尝试了递归写法，但是由于不便于返回矩阵链的分割方式，故后来选择循环重写了代码。递归方法写起来过程清晰，但需要考虑上一次递归和下一次递归之间的关系，以及要理清传递的内容；循环写起来则要严格考虑每次循环的次数，思路没有递归那么清楚，但是最后输出结果时直接查找之前计算的结果即可。两种方法各有优缺点。

本次实验加深了对动态规划算法的了解，动态规划算法巧妙地将问题化简成每一步的小问题，通过每一步求解、记录最优解来获得整体的最优解。在“大化小”的方法上和分治有异曲同工之处。

# 选做题

## 一、实验题目：0/1背包算法的实现

## 二、实验目的与内容

1、实验目的：

本次实验的目的是：了解01背包问题的内容，掌握动态规划算法的设计方法，利用动态规划算法解决01背包问题，分析算法复杂度。

2、实验内容：

两人一组，不限编程语言，各自利用动态规划算法解决01背包问题，分析复杂度、证明算法的正确性，共同完成PowerPoint进行展示交流。

## 三、算法设计

1、01背包算法描述

Knap(Start, End, Weight)函数描述：

STEP 1: 若Start和End（物品个数）相同，且weight大于0，返回价值0和空的物品选择记录。

STEP 2: 若weight小于等于0，返回价值-9999和空的物品选择记录。

STEP 3: 利用递归调用自身，获取选择第Start个物品时背包的价值和此时第Start个物品之后的选择记录，不选择第Start个物品时背包的价值和此时第Start个物品之后的选择记录。

STEP 4：比较两个价值。若选择第Start个物品时价值更大，则返回选择第Start个物品时的背包价值，并向第Start个物品之后的选择记录中加入第Start个物品的记录。否则则返回不选择第Start个物品的背包价值，和此时的商品选择记录

2、算法流程图

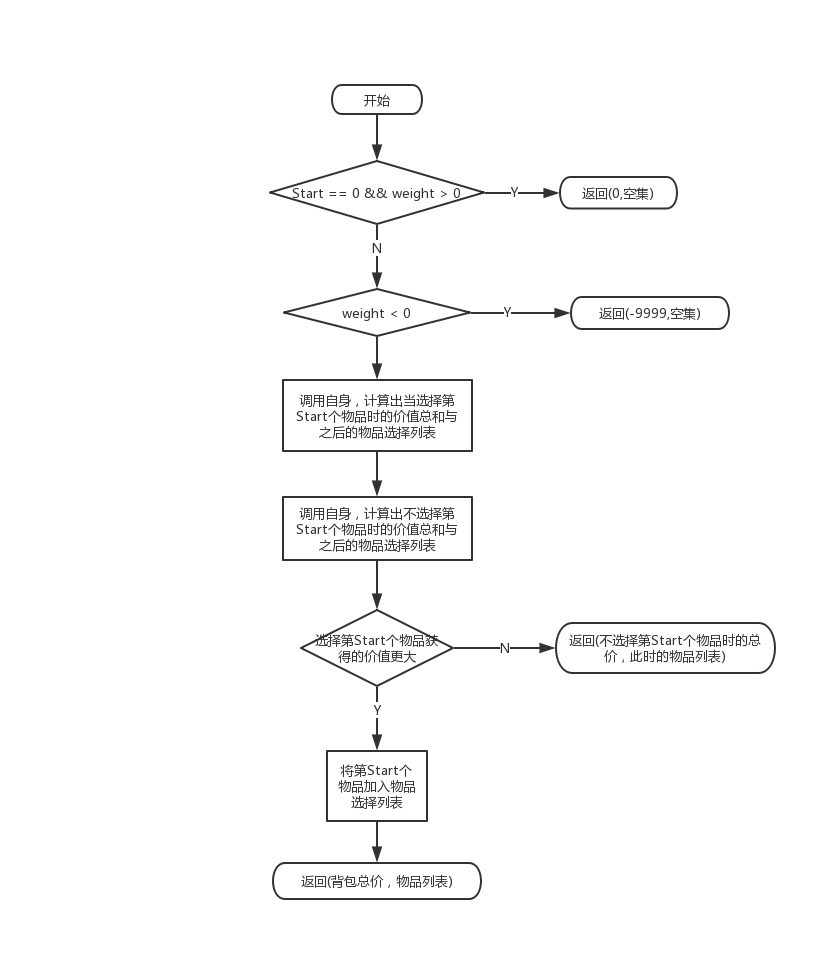


图2.1 01背包算法流程图

## 四、实验环境

操作系统：Windows 10

编译环境：Sublime Text3

语言版本：python3.7.4

## 五、实验过程

程序代码如下：

|  |
| --- |
| 01背包算法： |
| def Knap(Start, End, Weight):  if Start == End and Weight >= 0:  return (0,[])  elif Weight < 0:  return (-9999, [])  else:  ChooseThisOne = Knap(Start + 1, End, Weight - Items[Start][0])  NotChooseThisOne = Knap(Start + 1, End, Weight)  if ChooseThisOne[0] + Items[Start][1] > NotChooseThisOne[0]:  #if choose this one is better  ChoosedItems = [Start]  ChoosedItems.extend(ChooseThisOne[1])  return (ChooseThisOne[0] + Items[Start][1], ChoosedItems)  else:  return NotChooseThisOne |

## 六、算法测试

测试样例1

|  |  |  |  |
| --- | --- | --- | --- |
| 样例输入 | 测试理由 | 理论输出 | 样例输出 |
| Items = [(2,3),(3,4),(4, 5),(5,6)]，  weight = 8 | 验证算法正确性 | Maximun value: 10  Choices are: [1, 3] |  |

由于理论输出与样例输出相符，所以测试样例1验证成功。

测试样例2

|  |  |  |  |
| --- | --- | --- | --- |
| 样例输入 | 测试理由 | 理论输出 | 样例输出 |
| （随机生成的20个物品和背包容量） | 测试算法性能 | 最大背包价值和选择的物品编号 |  |

由于理论输出与样例输出相符，所以测试样例2验证成功。

综上，算法通过所有样例的测试。

## 七、结果分析

1、算法正确性证明：

当背包中只有一个物品时，显而易见可有算法得出最优解。

当背包中有个物品时，假设算法可得到正确结果。

当背包中有个物品时，可选择第1个物品，也可以选择放弃第1个物品，由前面假设可知，算法能正确得到两种情况下的最优解，比较两者最优解，可得到整体的最优解。

综上所述，算法正确。

2、算法时间复杂度分析：

算法使用递归算法，每次递归计算当前第一个物品加入背包与否的价值，当背包中有个物品时，算法复杂度为：。

3、算法空间复杂度分析：

由于每次递归时，需要记录当前添加到背包内的物品，故空间复杂度为。

## 八、总结

01背包的动态规划算法的时间和空间复杂度都较高，尤其是空间复杂度，由于要记录每种情况下物品的选择情况，供未来查找，所需要的空间较大。实现的算法在背包长度超过50时所需的时间超30s。和矩阵链算法相比，由于01背包问题是NP完全问题，故虽然都用了动态规划算法，但实际上01背包的动态规划只是一种伪多项式时间算法，两个时间复杂度相差很多。