**《算法设计与分析》实验报告**

**实验二 递归与分治策略应用基础**

**学号： 20211104040**

**姓名： 曹文星**

**班级： 软工二班**

**日期： 3月23日,2023**

**一、实验目的**

1、理解递归的概念和分治法的基本思想

2、了解适用递归与分治策略的问题类型，并能设计相应的分治策略算法

3、掌握递归与分治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：以下题目要求应用递归与分治策略设计解决方案，要求算法描述准确且程序运行正确。

1、求n个元素的全排列，n个元素中允许出现重复元素，通过实例验证算法。

2、实现整数划分算法，通过实例验证，

3、实现Strassen矩阵乘法，并通过实例与传统乘法进行时间效率比较，分别对不同规模的矩阵进行实验，记录运行时间。

4、解决一个2k\*2k的特殊棋牌上的L型骨牌覆盖问题。

提交结果：算法设计分析思路、源代码及其时间复杂度分析说明和测试运行报告。

1. **设计分析**
2. **算法描述及程序**
3. 求n个元素的全排列，n个元素中允许出现重复元素，通过实例验证算法。

以下是Pycharm中的python实现代码: 并辅以解说.

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

# Permutation - All in Order by Python - 曹文星

# Question : 求n个元素的全排列，n个元素中允许出现重复元素，通过实例验证算法。

def permutation(something): # 主方法.

res = []

used = [False] \* len(something) # 构建标记列表:元素长度个false

something.sort() # 先将传入的排序，方便去重

backtrack(something, [], used, res) # 开始调用回溯算法, 结果保存在res里, 新传入一个未命名path列表

return res

# 递归-分治法设计策略: 1 结束器, 2 分治过程:把大问题化为子问题 3 进入子问题, 标记状态 4. 恢复现场, 标记还原

def backtrack(num, path, used, res):

# ? 1 Terminator

if len(path) == len(num): # 如果当前path中元素的个数已经与总的元素个数相等,即都找到了, 就停止继续搜索.

res.append(path[:])

# ? 2 process result

return # 由于python的参数是引用类型, 因此不用管传值,只要直接返回最后取res即可.

# 核心for循 : 在字符个数间do some iteration

for i in range(len(num)):

if not used[i]: # 进入前提: 未取过该元素

# ? 3 process current logic

if i > 0 and num[i] == num[i - 1] and not used[i - 1]: # 去重器:发现当前位置上的数和前一个位置上的数相同，并且前一个位置上的数还没被使用过，那么就可以跳过这个位置，避免重复。

continue # 这个去重器的设置是最大的难点, I think

used[i] = True # 来都来了, 先标记我来过了.

path.append(num[i]) # 来都来了, 路径标记

# ? 4 drill down

backtrack(num, path, used, res) # 进入子问题: 找下一个路径中的元素

# ? 5 restore current status

# 退出子问题就会来到这里, Bite the Dust !我回来了,恢复现场

used[i] = False # 继续尝试没取这个元素的情景

path.pop() # 同时记得把已经取这个元素的记录删除.

# 思路总结:

# 使用回溯算法五步法，从头到尾依次枚举每个位置上应该放置的数。

# 如果当前路径的长度已经等于数组的长度，就把这个路径加入到结果中,之后程序会自动尝试其他路径。

# 特别的,由于在本题中允许元素重复.因此我需要先将数组排序，方便给排在相邻元素之间进行去重。

# 测试

nums = [1, 2, 2]

killer = ['Killer Queen', 'King Crimson', 'White Snake']

helper = [114, 514, 1919, 810]

print(permutation(nums)) # 输出 [[1, 2, 2], [2, 1, 2], [2, 2, 1]]

print(permutation(killer))

print(permutation(helper))

2、实现整数划分算法，通过实例验证，

以下是python代码, 写了三种方法.

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

"""最终版.

Integer Division - divide by Python - 曹文星

Question : 实现整数划分算法，通过实例验证

算法思路：

整数划分问题就是将一个正整数拆分成若干个正整数的和，使得这些正整数的和等于原来的正整数。

例如，整数4可以拆分成1+1+1+1、1+1+2、1+3、2+2、4等若干种不同的拆分方法。

2种思路: DP数组 / 递归(层次过多,容易炸,不推荐

使用动态规划算法来解决这个问题。首先定义一个二维数组dp，其中dp[i][j]表示将正整数i拆分成若干个正整数的和，其中最大的数为j的方案数。

需要特别注意的是，本题的算法时间复杂度为O(n^2)，在n较大时，仍然可能会出现运算时间过长的情况。

递归算法的时间复杂度较高，极有可能会导致在计算较大的整数划分问题时出现栈溢出等问题，不适合计算较大的整数划分问题。"""

def integer\_partition(n): # 划分算法: 一维变二维

dp = [[0] \* (n + 1) for \_ in range(n + 1)] # 这里创建了DP数组来存储计算的整数划分算法方案数.

# DP[i][j]表示将i划分为若干个整数之和(正整数,下不赘述),并且其中最大的正整数不超过j, 这个二维数组中对象的值为对应的方案数.

for i in range(1, n + 1): # 初始化DP数组作用: 这实际上是通过常识设定程序运行的界限,减轻计算压力

dp[i][i] = 1 # 很简单: 把4划分, 如果里面有4这个元素它本身, 那么它绝对只有一种划分方法: 4自己.

dp[i][1] = 1 # 很容易: 把4划分, 里面最大数是1的只可能都是1,即只有一种方案.

# 核心操作:对DP数组

for i in range(3, n + 1): # i从3循环到n,判断其余元素.

for j in range(2, i): # j从2循环到i-1, 用来枚举最大的正整数j, 肯定从2开始,一直到i以下

if i - j >= j: # 比较零散的分法:划分为由j组成的若干个正整数的和.

dp[i][j] = dp[i - j][j] + dp[i - j][j - 1]

else: # 比较紧凑的分法:划分为由i-j和最大正整数小于j的若干整数之和

dp[i][j] = dp[i - j][i - j] + dp[i - j][j - 1] if (i - j != j - 1) | (j % 2 != 0) else dp[i - j][i - j]

return sum(dp[n][:n + 1]) # dp[n][j]表示将n划分,最大正整数不超过j的方案, 因为超不过n就这样可以了.

'''经过反复调试,这里检测到了一个问题: Dp[5][2]元素的值计算为3,实际上只有2( 221和21111两种划分 ). 获得3结果的路径是DP[3][1]+DP[3][2]=1+2=3.

问题对应的2个DP元素相加的含义是: {111}+({21}+{12})=3个, 很明显是DP[3][2]重复了.

那么去检查DP[3][2]的生命周期, 发现: dp[i - j][i - j] 和 dp[i - j][j - 1]是重复的, 故猜测问题出在这里.添加去重器.

接下来问题出在DP[5][3] = 311 & 32 本该是2,这里计算为1 来自: [2][2] + [2][2]本该是2 ,触发了查重,结果只有1. 这下又是查重器的问题了

啊这,手算后好像是要加上一个j%2来控制的(胡言乱语)

最后就成了这样....Help!

本来是准备问学长前辈了, 经过反复碰壁一整夜后第二晨起来决定从头搞起: 查重一直出问题就很可能是算法写的有问题

换了一个递推公式如下:

当n=1或m=1时，f(n,m)=1； 废话

当n<m时，f(n,m)=f(n,n)； 废话

当n=m时，f(n,m)=f(n,m-1)+1； 这个是修改的内容

当n>m时，f(n,m)=f(n,m-1)+f(n-m,m)。 这个是修改的

发现只有一个参数好像是真的不行啊,如果有一个参数的请告诉我吧.'''

def Great\_partition(n, m):

dp = [[0] \* (m + 1) for \_ in range(n + 1)]

for i in range(1, n + 1):

for j in range(1, m + 1):

if i == 1 or j == 1:

dp[i][j] = 1

elif i < j:

dp[i][j] = dp[i][i]

elif i == j:

dp[i][j] = dp[i][j - 1] + 1

else:

dp[i][j] = dp[i][j - 1] + dp[i - j][j]

return dp[n][m]

# 次数累加全局变量

times = 0

# 暴力递归算法1

def integer\_par\_recursion(n):

global times

times = 0

int\_divide(n, 1, {0: 0}) # 用python字典这个数据结构存储划分因子，从1开始，用0占位

return times

def int\_divide(number, index, dividing\_number):

# 从1开始遍历该整数所有划分因子

global times

for i in range(1, number + 1):

if i >= dividing\_number[index - 1]: # 与前一位划分因子比较，去重，如先有24,42则不行

dividing\_number[index] = i # 当前数-划分因子后还剩数，如6-1剩5

number\_rest = number - i # 整数被划分完毕

if number\_rest == 0:

# !这里可以选择输出划分因子

# for j in range(1, index):

# print(str(dividing\_number[j]) + '+', end='')

# print(str(dividing\_number[index]))

times = times + 1

else: # 未被划分完毕，继续，dividing\_Number划分位数+1

int\_divide(number\_rest, index + 1, dividing\_number)

else:

pass

# 暴力递归算法2 :更有感觉了

def int\_divide\_recursion2(n, m):

if n < 1 or m < 1:

return 0

if n == 1 or m == 1:

return 1

if n < m:

return int\_divide\_recursion2(n, n)

if n == m:

return int\_divide\_recursion2(n, m - 1) + 1

return int\_divide\_recursion2(n, m - 1) + int\_divide\_recursion2(n - m, m)

# 测试

# 递归1

print(integer\_par\_recursion(6))

print(integer\_par\_recursion(5))

print(integer\_par\_recursion(4))

print(integer\_par\_recursion(3), "from递归选手1")

# 递归2

print(int\_divide\_recursion2(6, 6))

print(int\_divide\_recursion2(5, 5))

print(int\_divide\_recursion2(4, 4))

print(int\_divide\_recursion2(3, 3), "from递归选手2")

# 分治Dp

print(Great\_partition(6, 6))

print(Great\_partition(5, 5))

print(Great\_partition(4, 4))

print(Great\_partition(3, 3), "from分治选手")

3、实现Strassen矩阵乘法，并通过实例与传统乘法进行时间效率比较，分别对不同规模的矩阵进行实验，记录运行时间。

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

# Strassen\_matrix\_multiply - SMM by Python - 曹文星

import datetime # 计算时间

"""Strassen矩阵乘法是一种快速矩阵乘法的算法，相比于传统的矩阵乘法算法，它的时间复杂度更低，但是它的常数因子比较大，所以在小规模矩阵的计算上不如传统的算法。

Strassen矩阵乘法的核心思想是将两个矩阵各自拆分成四个子矩阵，并通过一些特殊的运算来计算它们的乘积。

M1 = (A11 + A22) × (B11 + B22)

M2 = (A21 + A22) × B11

M3 = A11 × (B12 - B22)

M4 = A22 × (B21 - B11)

M5 = (A11 + A12) × B22

M6 = (A21 - A11) × (B11 + B12)

M7 = (A12 - A22) × (B21 + B22)

C11 = M1 + M4 - M5 + M7

C12 = M3 + M5

C21 = M2 + M4

C22 = M1 - M2 + M3 + M6

最后将这四个子矩阵C11、C12、C21和C22类似田字格合并成一个n×n的矩阵C即可。"""

# Strassen矩阵乘法python示例:

def strassen\_matrix\_multiply(A, B):

u = len(A)

if u == 1:

return [[A[0][0] \* B[0][0]]]

# 将矩阵A和B分成四个子矩阵

A11, A12, A21, A22 = split\_matrix(A)

B11, B12, B21, B22 = split\_matrix(B)

# 递归计算七个乘积

M1 = strassen\_matrix\_multiply(add\_matrices(A11, A22), add\_matrices(B11, B22))

M2 = strassen\_matrix\_multiply(add\_matrices(A21, A22), B11)

M3 = strassen\_matrix\_multiply(A11, subtract\_matrices(B12, B22))

M4 = strassen\_matrix\_multiply(A22, subtract\_matrices(B21, B11))

M5 = strassen\_matrix\_multiply(add\_matrices(A11, A12), B22)

M6 = strassen\_matrix\_multiply(subtract\_matrices(A21, A11), add\_matrices(B11, B12))

M7 = strassen\_matrix\_multiply(subtract\_matrices(A12, A22), add\_matrices(B21, B22))

# 计算四个子矩阵C11、C12、C21和C22

C11 = subtract\_matrices(add\_matrices(add\_matrices(M1, M4), M7), M5)

C12 = add\_matrices(M3, M5)

C21 = add\_matrices(M2, M4)

C22 = subtract\_matrices(add\_matrices(add\_matrices(M1, M3), M6), M2)

# 合并四个子矩阵

C = [[0 for j in range(u)] for i in range(u)]

for i in range(u // 2):

for j in range(u // 2):

C[i][j] = C11[i][j]

C[i][j + u // 2] = C12[i][j]

C[i + u // 2][j] = C21[i][j]

C[i + u // 2][j + u // 2] = C22[i][j]

return C

def add\_matrices(A, B): # 矩阵相加

l = len(A)

C = [[0 for \_ in range(l)] for \_ in range(l)]

for i in range(l):

for j in range(l):

C[i][j] = A[i][j] + B[i][j]

return C

def subtract\_matrices(A, B): # 矩阵相减

m = len(A)

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

for i in range(m):

for j in range(m):

C[i][j] = A[i][j] - B[i][j]

return C

def split\_matrix(A): # 矩阵切分为4份

k = len(A)

m = k // 2

A11 = [[0 for \_ in range(m)] for \_ in range(m)]

A12 = [[0 for \_ in range(m)] for \_ in range(m)]

A21 = [[0 for \_ in range(m)] for \_ in range(m)]

A22 = [[0 for \_ in range(m)] for \_ in range(m)]

for i in range(m):

for j in range(m):

A11[i][j] = A[i][j]

A12[i][j] = A[i][j + m]

A21[i][j] = A[i + m][j]

A22[i][j] = A[i + m][j + m]

return A11, A12, A21, A22

# 传统矩阵乘法可以按照以下的公式来计算：

# C[i][j] = sum(A[i][k] \* B[k][j] for k in range(n))

def matrix\_multiply(A, B):

o = len(A)

C = [[0 for \_ in range(o)] for \_ in range(o)]

for i in range(o):

for j in range(o):

C[i][j] = sum(A[i][k] \* B[k][j] for k in range(o))

return C

# 打印矩阵

def matrix\_print(A):

print('\n打印矩阵\n----------')

for a in A:

for row in a:

print(row, end=' ')

print()

print('----------\n')

# 测试

n = 4

matrix3 = [[5 for \_ in range(n)] for \_ in range(n)]

matrix32 = [[6 for \_ in range(n)] for \_ in range(n)]

n = 4

matrix4 = [[6 for \_ in range(n)] for \_ in range(n)]

matrix42 = [[7 for \_ in range(n)] for \_ in range(n)]

n = 4

matrix5 = [[7 for \_ in range(n)] for \_ in range(n)]

matrix52 = [[8 for \_ in range(n)] for \_ in range(n)]

#####

start1 = datetime.datetime.now()

matrix\_out1 = strassen\_matrix\_multiply(matrix3, matrix32)

matrix\_out2 = strassen\_matrix\_multiply(matrix4, matrix42)

matrix\_out3 = strassen\_matrix\_multiply(matrix5, matrix52)

# 打印结果

matrix\_print(matrix\_out1)

matrix\_print(matrix\_out2)

matrix\_print(matrix\_out3)

end1 = datetime.datetime.now()

print('Strassen time is ', end1 - start1)

# Strassen time is 0.000999s

#####

start2 = datetime.datetime.now()

matrix\_out1 = matrix\_multiply(matrix3, matrix32)

matrix\_out2 = matrix\_multiply(matrix4, matrix42)

matrix\_out3 = matrix\_multiply(matrix5, matrix52)

# 打印结果

matrix\_print(matrix\_out1)

matrix\_print(matrix\_out2)

matrix\_print(matrix\_out3)

end2 = datetime.datetime.now()

print('Normal time is ', end2 - start2)

# Normal time is 0.001143s

4、解决一个2k\*2k的特殊棋牌上的L型骨牌覆盖问题。

Python代码如下:

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

# LBoard\_cover - LBC by Python - 曹文星

#

# "我已经拿起,最初的那块骨牌了!"

# 问题描述：

# 在一个 2^k \* 2^k 方格组成的棋盘中，若恰有一个方格与其他方格不同，称该方格为特殊方格，且称该棋盘为特殊棋盘(Defective Chessboard)。

# 特殊棋盘块是当k=2 时16个特殊棋盘中的一个，在棋盘覆盖问题中，要求用图示所示的4种不同形状的L型骨牌覆盖给定棋盘上除特殊方格以外的所有方格，且任何俩个L型骨牌方格不得重叠。在任何一个2k\*2k 的覆盖棋盘中，用到的L型骨牌个数为(4^k-1)/3。

# 解题思路：使用分治法的思路，棋盘覆盖问题，通过在棋盘的中心位置放置一个L形骨牌，将原问题拆分成四个相同规模的子问题，再对各个子问题递归求解。

# 递归函数的出口为在2\*2的棋盘中放置一个L形骨牌。

board\_size = 8 # 大小为标准国际象棋棋盘大小

tile = 0

label = 0

"""

board: 棋盘本身或是指更小的单元,用二维数组表示,其中有一个(右下角)定义的死点

tr, tc: 左上角的行列坐标

dr, dc: 特殊方块的行列坐标

size: 棋盘大小

label: 骨牌标记

"""

def cover(board, tr, tc, dr, dc, size): # 递归调用方法,参数如所示

global tile

global label

if size == 1:

return

tile += 1

sub\_size = size // 2

label += 1

# 特殊方块在左上角小棋盘内

if dr < tr + sub\_size and dc < tc + sub\_size:

cover(board, tr, tc, dr, dc, sub\_size)

else:

# 不在左上角小棋盘内，覆盖左上角小棋盘右下角

board[tr + sub\_size - 1][tc + sub\_size - 1] = label

cover(board, tr, tc, tr + sub\_size - 1, tc + sub\_size - 1, sub\_size)

# 特殊方块在右上角小棋盘内

if dr < tr + sub\_size and dc >= tc + sub\_size:

cover(board, tr, tc + sub\_size, dr, dc, sub\_size)

else:

# 不在右上角小棋盘内，覆盖右上角小棋盘左下角

board[tr + sub\_size - 1][tc + sub\_size] = label

cover(board, tr, tc + sub\_size, tr + sub\_size - 1, tc + sub\_size, sub\_size)

# 特殊方块在左下角小棋盘内

if dr >= tr + sub\_size and dc < tc + sub\_size:

cover(board, tr + sub\_size, tc, dr, dc, sub\_size)

else:

# 不在左下角小棋盘内，覆盖左下角小棋盘右上角

board[tr + sub\_size][tc + sub\_size - 1] = label

cover(board, tr + sub\_size, tc, tr + sub\_size, tc + sub\_size - 1, sub\_size)

# 特殊方块在右下角小棋盘内

if dr >= tr + sub\_size and dc >= tc + sub\_size:

cover(board, tr + sub\_size, tc + sub\_size, dr, dc, sub\_size)

else:

# 不在右下角小棋盘内，覆盖右下角小棋盘左上角

board[tr + sub\_size][tc + sub\_size] = label

cover(board, tr + sub\_size, tc + sub\_size, tr + sub\_size, tc + sub\_size, sub\_size)

# 测试代码

# 初始化

board1 = [['.' for \_ in range(board\_size)] for \_ in range(board\_size)]

board1[board\_size - 1][board\_size - 1] = '\*'

# 打印开始状态

for i in range(board\_size):

print(' '.join(board1[i]))

cover(board1, 0, 0, board\_size - 1, board\_size - 1, board\_size)

print("\n处理完毕\n")

for i in range(board\_size):

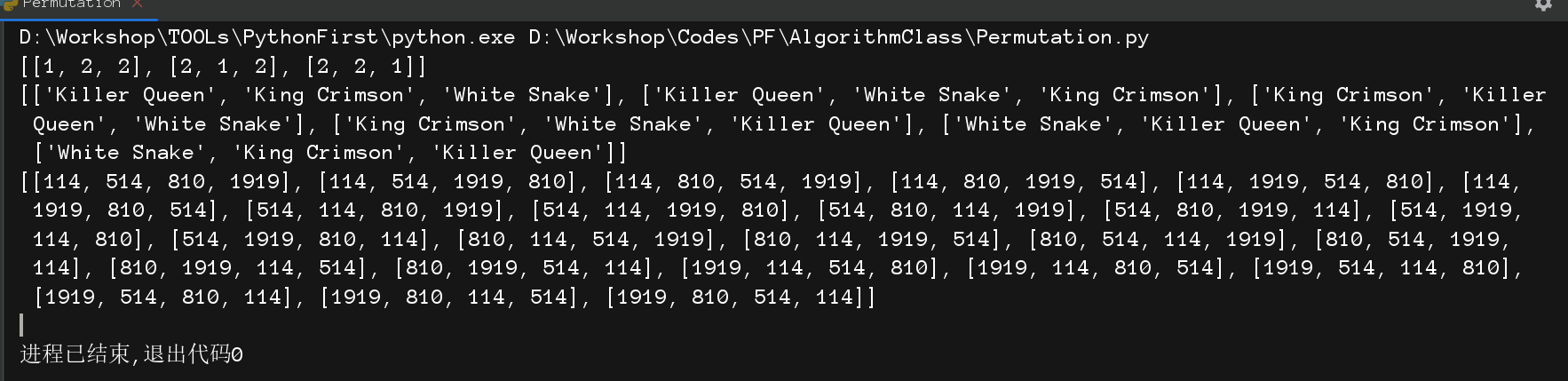
print(' '.join(str(board1[i])))

# 输出可能有点问题...

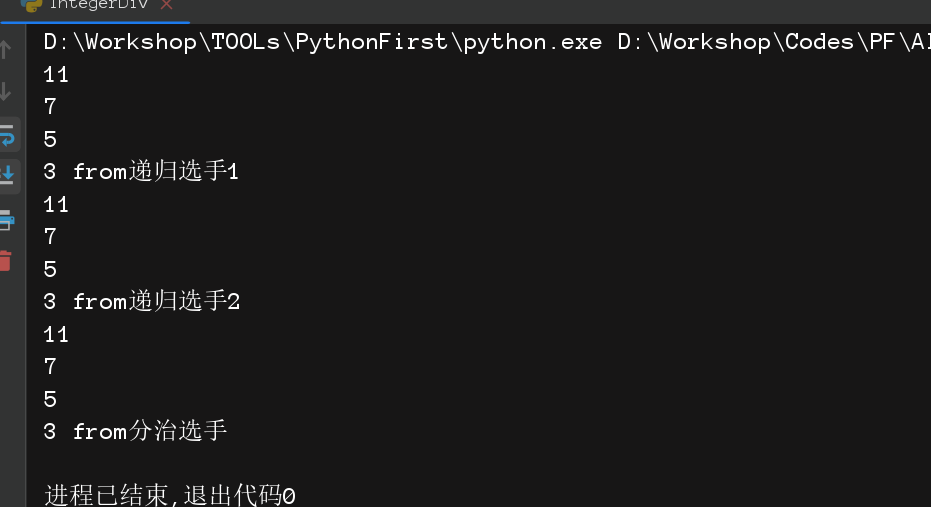
print("\nL型骨牌数量:", tile) # ==21

最后输出解决方案,用数字表示不同的骨牌块

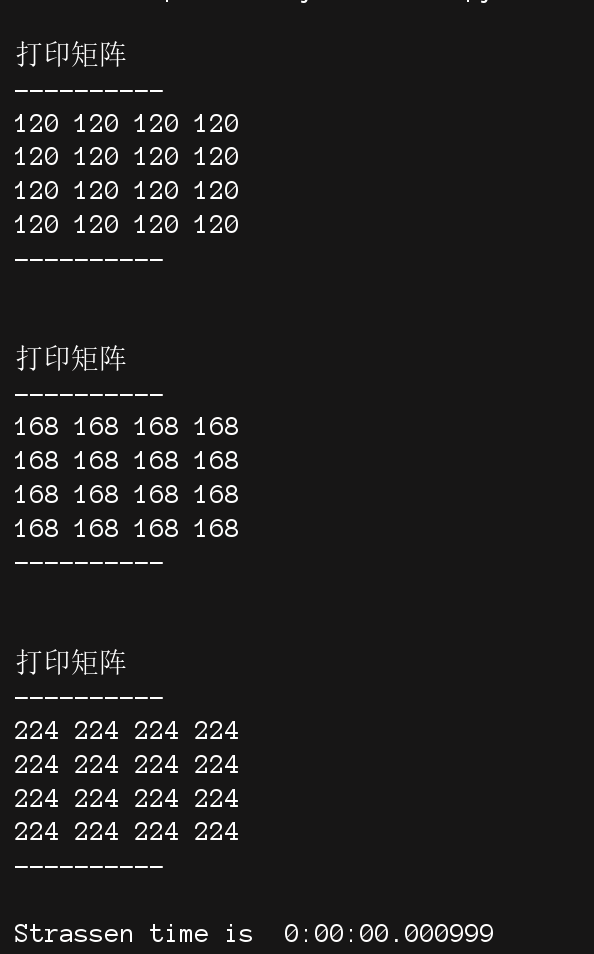
1. **测试与分析**



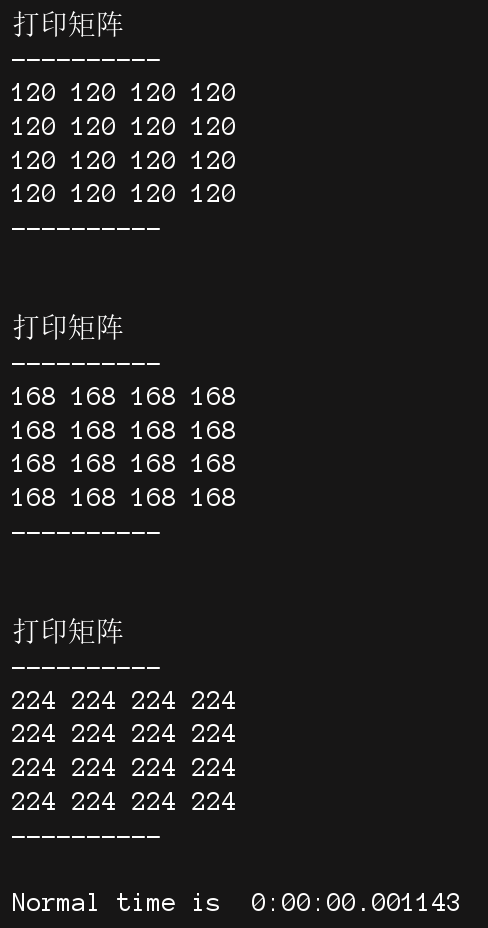
实验1输出样例:完成了全排列



实验2输出样例:通过三种不同的算法完成了整数划分算法

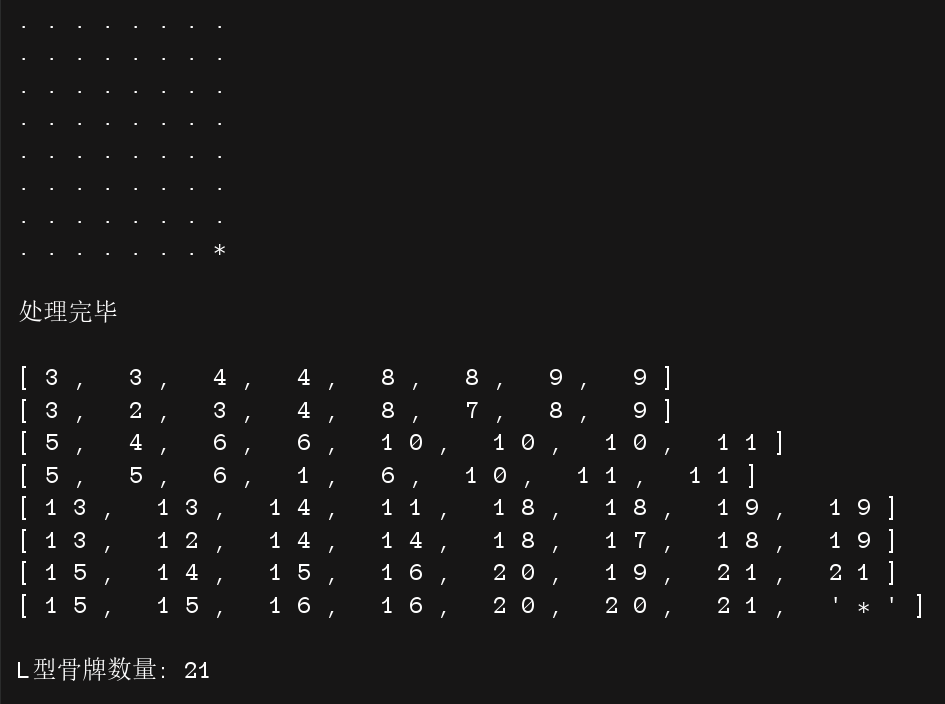


实验3测试1: StrassenMatrix算法时间:0.001秒



实验3测试2: 普通Matrix算法时间:0.00114秒

可以看出在4阶时两者时间相差不大



实验四, 骨牌解决图示

1. **实验总结与体会**

基本完成了任务目标, 有些甚至是超额完成.

递归和动态规划和分治法这些概念, 理解起来简单, 做起来难!