成绩：



**算法分析与设计课程设计**

**报 告**

二级学院 计算机科学与工程学院

专 业 计算机科学与技术

班 级 117030703

学生姓名 陈松 学号 11703990535

指导教师 刘万平

时 间 2019年11月28日

目录

[1、引言 2](#_Toc26184477)

[1.1 课程设计目的 2](#_Toc26184478)

[1.2课程设计要求 2](#_Toc26184479)

[1.3 常用算法设计技术介绍 2](#_Toc26184480)

[1.3.1 递归回溯 2](#_Toc26184481)

[1.3.2 广度优先搜索 3](#_Toc26184482)

[1.3.3 动态规划 3](#_Toc26184483)

[1.3.4 贪心算法 3](#_Toc26184484)

[2、 题目解题思路 4](#_Toc26184485)

[2.1 题目A2:绘制分形树问题 4](#_Toc26184486)

[2.1.1 题目描述 4](#_Toc26184487)

[2.1.2 问题分析与解决思路 4](#_Toc26184488)

[2.1.3模型建立与算法描述 5](#_Toc26184489)

[2.1.4 算法实现与复杂度分析 6](#_Toc26184490)

[2.1.5 程序实现及运行结果分析 7](#_Toc26184491)

[2.2 题目A4: 泊松分酒问题 7](#_Toc26184492)

[2.2.1 问题描述 7](#_Toc26184493)

[2.2.2 问题分析与解决思路 7](#_Toc26184494)

[2.2.3 模型建立与算法描述 8](#_Toc26184495)

[2.2.4 算法实现与复杂度分析 9](#_Toc26184496)

[2.2.5 程序实现及运行结果分析 10](#_Toc26184497)

[2.3 题目B2：树塔问题 11](#_Toc26184498)

[2.3.1问题描述 11](#_Toc26184499)

[2.3.2问题分析与解决思路 11](#_Toc26184500)

[2.3.3模型建立与算法描述 12](#_Toc26184501)

[2.3.4 算法实现与复杂度分析 12](#_Toc26184502)

[2.3.5程序实现及运行结果分析 13](#_Toc26184503)

[2.4 题目C2地图着色问题 14](#_Toc26184504)

[2.4.1 问题描述 14](#_Toc26184505)

[2.4.2问题分析与解决思路 14](#_Toc26184506)

[2.4.3 模型建立与解决思路 15](#_Toc26184507)

[2.4.4 算法实现与复杂度分析 16](#_Toc26184508)

[2.4.5 程序实现与运行结果分析 16](#_Toc26184509)

[2.5 题目B1：基因序列比较 18](#_Toc26184510)

[2.5.1问题描述 18](#_Toc26184511)

[2.5.2 问题分析与解决思路 19](#_Toc26184512)

[2.5.3模型建立与算法描述 19](#_Toc26184513)

[2.5.4 算法实现与复杂度分析 20](#_Toc26184514)

[2.5.5 程序实现与运行结果分析： 21](#_Toc26184515)

[3、总结 23](#_Toc26184516)

[3.1所遇问题及解决方法 23](#_Toc26184517)

[3.1.1地图着色问题 23](#_Toc26184518)

[3.1.2 广度算法的时间复杂度问题 23](#_Toc26184519)

[3.2 心得体会 23](#_Toc26184520)

# 1、引言

## 1.1 课程设计目的

（1）进一步加强学生的算法思维训练，培养学生的分析动手能力。

（2）巩固理论课程学习的常用算法设计方法及分析方法。

（3）学会为一些简单的综合实际问题设计算法。

（4）加强算法时间空间复杂性分析能力。

（5）进一步锻炼和加强学生的程序设计能力。

## 1.2课程设计要求

（1）利用所学的算法分析与设计理论知识，独立完成所选的题目；

（2）建立模型：为所选题目建立数学模型；

（3）数据结构设计：用数据结构表达模型；

（4）算法设计：在数结构的基础上设计解决问题的算法；

（5）算法分析：分析所设计算法的时间空间复杂度，说明算法的可行性。

（6）算法实现：用程序设计语言实现算法，并进行测试。

撰写的课程设计报告中应记录模型建立、数据结构、算法描述，算法时间、空间复杂度分析、算法正确性、程序执行的结果等。

## 1.3 常用算法设计技术介绍

### 1.3.1 递归回溯

递归，是一种算法结构，一个递归就是在函数中调用函数本身来解决问题。在描述问题的某一状态时，需要用到该状态的上一状态，而描述上一状态，又需要用到上一状态的上一状态……这种用自已来定义自己的方法，称为递归定义。当然递归也不是一直无休止的自身调用，他总需要有一个递归出口条件。比如最简单的递归算法，求阶乘，n的阶乘f(n) = n \* f(n-1),这就是一个自身调用自身的例子，他的出口条件当然就是1! = 1和0! = 1。而回溯，是一种算法思想，他是通过不同的尝试来生成问题的解，以深度优先方式系统搜索问题的解，有点类似于穷举，但是和穷举不同的是，回溯会“剪枝”，也就是对已经知道错误的结果没必要再枚举接下来的答案了。他适用于解决组合数较大的问题

### 1.3.2 广度优先搜索

广度优先搜索算法（Breadth First Search，缩写为BFS），又译作宽度优先搜索，或横向优先搜索，是一种图形搜索算法。简单的说，广度优先搜索算法是从根节点开始，沿着树的宽度遍历树的节点。如果所有节点均被访问，则算法中止。借助广度优先搜索算法，可以让你找出两样东西之间的最短距离。

### 1.3.3 动态规划

动态规划（dynamic programming）与分治方法类似，都是通过组合子问题的解来求解原问题。但是动态规划算法对每个子子问题只求解一次，将其保存在一个表格中，从而无需每次求解一个子子问题时都重复计算，避免来这种不必要对对计算工作。

       通常按照一下四个步骤来设计一个动态规划算法：

**刻画一个最优对结构特征。**

**递归地定义最优解对值。**

**计算最优解的值，通常采用自底向上的方法。**

**利用计算出的信息构造一个最优解。**

      动态规划算法通常基于一个递推公式及一个或多个初始状态。当前子问题的解将由上一次子问题的解推出。首先，需要先找到某个状态的最优解，然后在它的帮助下，找到下一状态的最优解，“状态”用来描述该问题的子问题的解。“状态”并不是随便给的，在大多数情况下，某个状态只与它前面出现的状态有关，而独立于后面的状态。然后分析状态转移方程以及初始状态。

     动态规划的本质就是递归，这样比较容易判断一个问题是否可以用动态规划算法去求解，因为只要思考问题的规模缩小/扩大之后是不是同样的方法求解即可。

     适用应用动态规划算法求解的最优问题应该具备两个要素：

**最优子结构**

**子问题重叠**

### 1.3.4 贪心算法

贪心算法（又称贪婪算法）是指，在对问题求解时，总是做出在当前看来是最好的选择。也就是说，不从整体最优上加以考虑，他所做出的是在某种意义上的局部最优解。

贪心算法不是对所有问题都能得到整体最优解，关键是贪心策略的选择，选择的贪心策略必须具备无后效性，即某个状态以前的过程不会影响以后的状态，只与当前状态有关

**贪心算法的要素**

**贪心选择**

**贪心选择是指所求问题的整体最优解可以通过一系列局部最优的选择，即贪心选择来达到。这是贪心算法可行的第一个基本要素，也是贪心算法与动态规划算法的主要区别。贪心选择是采用从顶向下、以迭代的方法做出相继选择，每做一次贪心选择就将所求问题简化为一个规模更小的子问题。对于一个具体问题，要确定它是否具有贪心选择的性质，我们必须证明每一步所作的贪心选择最终能得到问题的最优解。通常可以首先证明问题的一个整体最优解，是从贪心选择开始的，而且作了贪心选择后，原问题简化为一个规模更小的类似子问题。然后，用数学归纳法证明，通过每一步贪心选择，最终可得到问题的一个整体最优解。**

**最优子结构**

**当一个问题的最优解包含其子问题的最优解时，称此问题具有最优子结构性质。运用贪心策略在每一次转化时都取得了最优解。问题的最优子结构性质是该问题可用贪心算法或动态规划算法求解的关键特征。贪心算法的每一次操作都对结果产生直接影响，而动态规划则不是。贪心算法对每个子问题的解决方案都做出选择，不能回退；动态规划则会根据以前的选择结果对当前进行选择，有回退功能。动态规划主要运用于二维或三维问题，而贪心一般是一维问题 。**

# 题目解题思路

## 2.1 题目A2:绘制分形树问题

### 题目描述

如下图所示，先垂直绘制一根线段，然后在线段长度的三分之一处和三分之二处分别以固定夹角绘制另外两根线段，长度分别为原线段的2/3和1/3. 如此反复，直至线段长度小于某个较小的值。其中，线条颜色以及长度，夹角都可以自行进行微调。

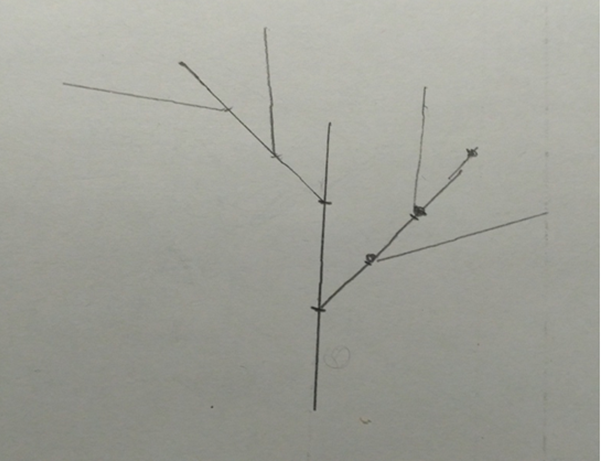


图1 分形树描述图

### 2.1.2 问题分析与解决思路

由题目分析，该题宜采用递归回溯策略解决，对于递归式，必须先设置一个结束条件，即当线段长度小于某值(sh)时，递归结束。

对于递归的一次过程，可以分成3个部分：

向前走1/3的距离，右转一定角度，分出2/3线段长度，角度回溯到主干，即向左转相同角度。

向前走1/3的距离，左转一定角度，分出1/3线段长度，角度回溯到主干，即向右转相同角度

向前走1/3的距离，保持方向不变，回溯倒退到起点位置。

如此递归回溯下去，分形树便绘制完成。

### 2.1.3模型建立与算法描述

**我们将上述分析过程和解决的思路进一步归纳为以下步骤：**

1. **设定递归结束条件，即可绘制的最小线段长度sh**
2. **输入起始长度n**
3. **判断 n 是否大于最小线段长度sh**
4. **绘制n/3长度后,右转一定角度，分出n/3\*2的长度进行递归，即进入步骤(3),**

**左转相同角度，进入步骤(5)**

1. **继续绘制n/3长度后, 左转一定角度，分出n/3的长度进行递归,即进入步骤(3)**

**右转相同角度，进入步骤(6)**

**（6） 最后绘制n/3长度后，保持方向不变回溯，即画笔落点回退长度n。**

**算法伪代码描述：**

**def draw\_branch(branch\_length, sh, angle, pensize):**

**# 递归结束条件**

**if branch\_length > sh:**

**# 向前走1/3的距离**

**selectColor(branch\_length / 3 \* 2, sh, pensize) # 1/3处无分支，则设为红色**

**turtle.forward(branch\_length / 3)**

**# 若右侧有分支，则右转画出分支**

**turtle.right(angle)**

**draw\_branch(branch\_length / 3 \* 2, sh, angle, pensize - 1)**

**turtle.left(angle) # 画完后，回到主支方向**

**# 向前走1/3的距离**

**selectColor(branch\_length / 3, sh, pensize) # 2/3处无分支，则设为绿色**

**turtle.forward(branch\_length / 3)**

**# 若侧左侧有分支，则右转画出分支**

**turtle.left(angle)**

**draw\_branch(branch\_length / 3, sh, angle, pensize - 1)**

**turtle.right(angle) # 画完后，回到主支方向**

**# 向前走1/3的距离**

**selectColor(branch\_length / 9 \* 2, sh, pensize) # 主支的最后1/3小于限定值，则化成绿色**

**turtle.forward(branch\_length / 3)**

**# 退回到原点处**

**turtle.penup()**

**turtle.backward(branch\_length)**

**turtle.pendown()**

### 2.1.4 算法实现与复杂度分析

算法很好的利用了递归函数的优势，位置长度信息仅依靠函数间的参数长度实现

时间复杂度：

算法递归过程基本操作为绘制线段。

建立递推关系式: f(n) = n + f(n/3) +f(n/3\*2) n >sh

所以时间复杂度：

**可见每层的值都为n，从根到叶节点的最长路径是：**

****

**因为最后递归的停止是在(2/3)kn == 1.则**

****

**于是**

****

**即T(n) = O(nlogn)**

空间复杂度：

算法实现了递归函数间的参数传递，并利用了参数进行回溯，所以没有除递归函数外的空间消耗。

### 2.1.5 程序实现及运行结果分析

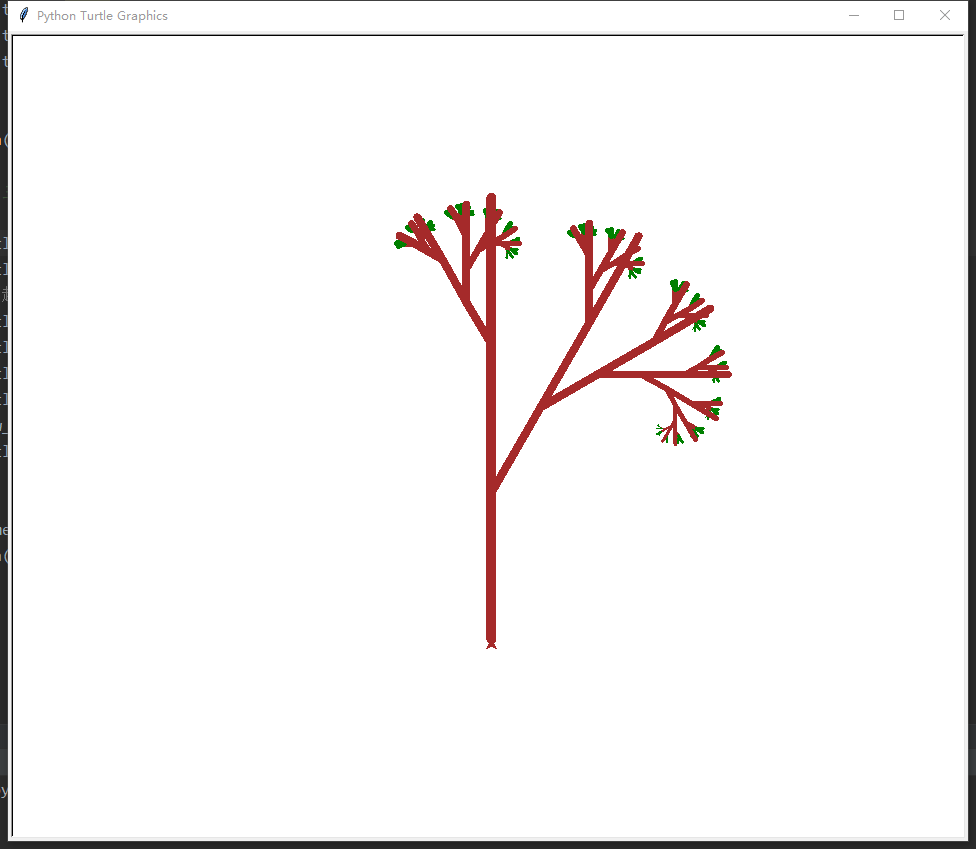


图 2 绘制分形树

程序运行结果分析：

由于采用的递归的策略，刚开始一直向右转相应角度并且递归绘制传入参数的n/3长度。直到传入长度小于定义的最小值。

## 题目A4: 泊松分酒问题

### 问题描述

西蒙.丹尼斯.泊松是著名的法国数学家和物理学家。据说在他遇到某个古老的谜题之后，就开始对数学感兴趣了，这个谜题是这样的：给定一个装满水的8品脱壶以及两个容量分别为5品脱和3品脱的空壶，如何通过完全灌满或者到空这些壶从而使得某个壶精确地装有4品脱的水？

### 问题分析与解决思路

由题目分析，该问题适合采用广度优先策略，其中广度算法的宽度记为所有的到水可能，如有容量不同的三个水壶a,b,c. 即’宽度’:a - > b、a->c、b ->c、b->a、c-a、c->b.

### 模型建立与算法描述

**我们将上述分析过程和解决的思路进一步归纳为以下步骤：**

(1) 初始化队列，将水壶情况(8,0,0)入队列

(2) 判断队列是否为空

(3) 取出队首元素，即获得水壶水量情况，

(4) 通过六种可能的倒水行为，产生新的六种水壶水量情况

(5) 判断其中是否存在有水壶水量为4的情况，若有则结束程序，否则进入步骤2

上述使用队列的算法明显可以用广度遍历完成，但对于可以进一步优化算法

修剪分支：

设定条件，去除灌水的壶的水量为0和被灌水的壶水量满的情况的情况

避免重复:

当取出队首元素后，同时将水壶情况加入去重集合中，即六种水壶水量情况必须满足其情况不能在去重集合中，否则不将此中情况加入队列中。

**算法伪代码描述：**

**def run(self):**

**"""**

**广度优先遍历**

**:return:**

**"""**

**while len(self.queue) != 0:**

**point = self.delqueue()**

**# 遍历六种倒水可能**

**for i in range(3):**

**for j in range(1, 3):**

**if not self.pour(point, i, (i + j) % 3):**

**self.createPath()**

**return True**

**print("不存在")**

**return False**

**def pour(self, situation: list, a, b):**

**"""**

**a ->b 倒水**

**"""**

**# 去除灌水壶的水量为0和被灌水壶容量满的情况的情况**

**if situation[a] == 0 or situation[b] == self.capacity[b]:**

**return True**

**# b 需要多少水能倒满**

**b\_need = self.capacity[b] - situation[b]**

**# a 需要给多少才能为空**

**a\_take = situation[a]**

**# 如果能倒空**

**temp = situation.copy()**

**# 灌满或灌空**

**if a\_take >= b\_need:**

**temp[a] = a\_take - b\_need**

**temp[b] = self.capacity[b]**

**temp[5] = b\_need**

**else:**

**temp[a] = 0**

**temp[b] = temp[b] + a\_take**

**temp[5] = a\_take**

**temp[3], temp[4] = a, b**

**# 判断该情况是否出现过**

**if tuple(temp) not in self.pointList:**

**self.enqueue(temp) # 加入队列**

**self.enPointList(situation, temp) # 加如情况存在字典**

**if self.endValue in temp:**

**return False**

**return True**

### 算法实现与复杂度分析

算法用队列进行实现，对每种可能进行遍历，每次都需要进行3\*(3-1)种组合可能

时间复杂度：

由于水壶容量的不同，会导致得到的所有水壶水量不同，即不好分析最差情况下的时间复杂度。

空间复杂度:

同理，递归树的高度不好分析，即队列所占的最大空间大小不一定。

### 程序实现及运行结果分析

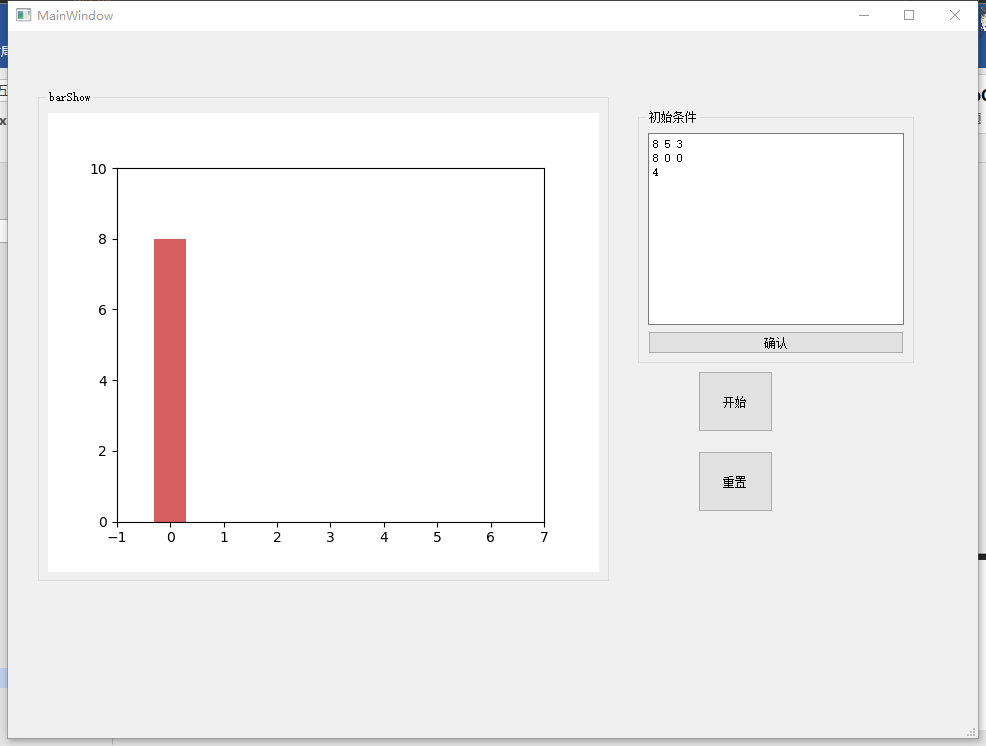


图3 开始时，水壶情况

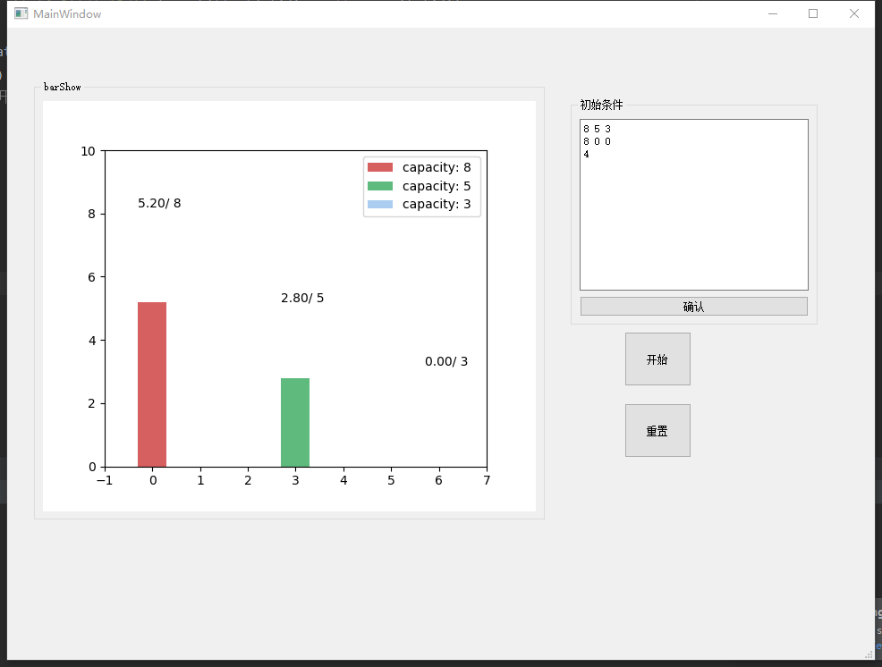


图 4 红色壶向绿色壶倒水

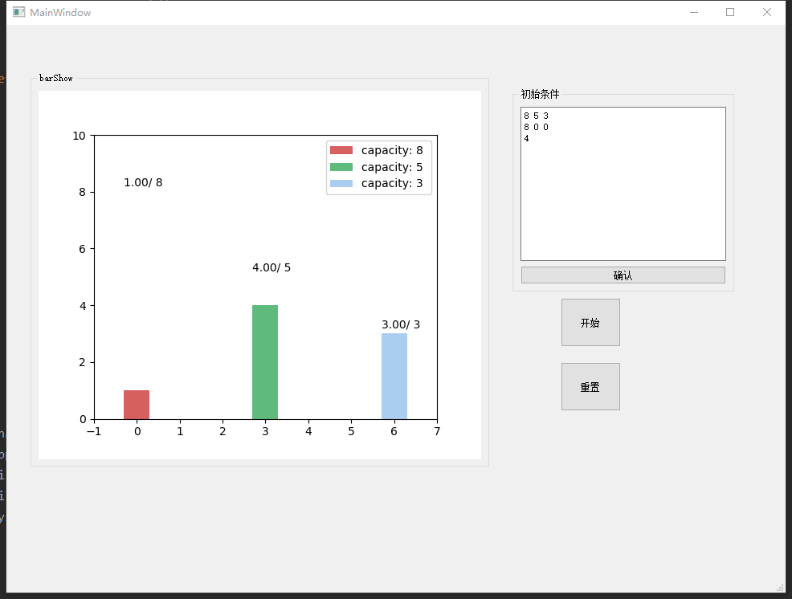


图5 结束时，水壶情况

程序运行分析

输入: 8 5 3

输出:1 4 3

过程:

(8, 0, 0, 0, 0, 0)

(3, 5, 0, 0, 1, 5)

**ps: 0号壶向1号壶倒水:1号壶水满为5,0号壶倒水后为3，即如图5所示**

(3, 2, 3, 1, 2, 3)

(6, 2, 0, 2, 0, 3)

(6, 0, 2, 1, 2, 2)

(1, 5, 2, 0, 1, 5)

(1, 4, 3, 1, 2, 1)

## 2.3 题目B2：树塔问题

### 2.3.1问题描述

将正整数排成等边三角形（也叫数塔），三角形的底边有个数，下图给出了的一个例子。从三角形顶点出发通过一系列相邻整数（在图中用正方形表示），如何使得到达底边时的总和最大？

### 2.3.2问题分析与解决思路

解决方案是自底至顶分析，自上而下计算。因此我们从第四层的四个数据开始分析：

如果最优路径经过2，那么一定经过19

如果最优路径经过18，那么一定经过10

如果最优路径经过9，那么一定经过10

如果最优路径经过5，那么一定经过16

因此，我们总结出规律：如果最有路径经过当前点 ，那么从当前点到路径尾节点的数字之和 将是当前点的值加上左右孩子其中较大的值，即：

dp[i][j] = max(dp[i+1][j],dp[i+1][j+1])+date[i][j]

### 2.3.3模型建立与算法描述

***步骤如下:***

(1) 初始化距离数组dp，令距离dp的最后一行复制树塔的最后一行的值  
(2) 从树塔倒数第二行开始，自底向上计算  
(3) 判断x点的左右孩子的大小，对应的距离dp = 左右孩子中的较大值加上树塔对应位置值  
(4) 重复2、3步骤，直到计算完树塔顶端

***伪代码描述如下:***

def run(self):

index = len(self.pagoda) - 1

for j, value in enumerate(self.pagoda[-1]):

yield self.getIndex(index, j), 0, value

for i in range(len(self.pagoda) - 2, -1, -1): # 自底向上求得最优值

layer = self.pagoda[i]

for j in range(len(layer)):

if layer[j] == 0:

break

self.find(i, j)

yield self.getIndex(i, j), self.getIndex(\*self.next[(i, j)]), self.dp[i, j]

### 2.3.4 算法实现与复杂度分析

设定M为树塔高度，N为树塔底部的长度

算法实现的主要数据结构:数组矩阵

时间复杂度:

使用了动态规划计算结果，即时间复杂度为树塔元素个数。

空间复杂度:

算法使用的一个M,N的数组存放树塔数据，使用一个M\*N的数组存放最短距离

即空间复杂度2\*M\*N

### 2.3.5程序实现及运行结果分析

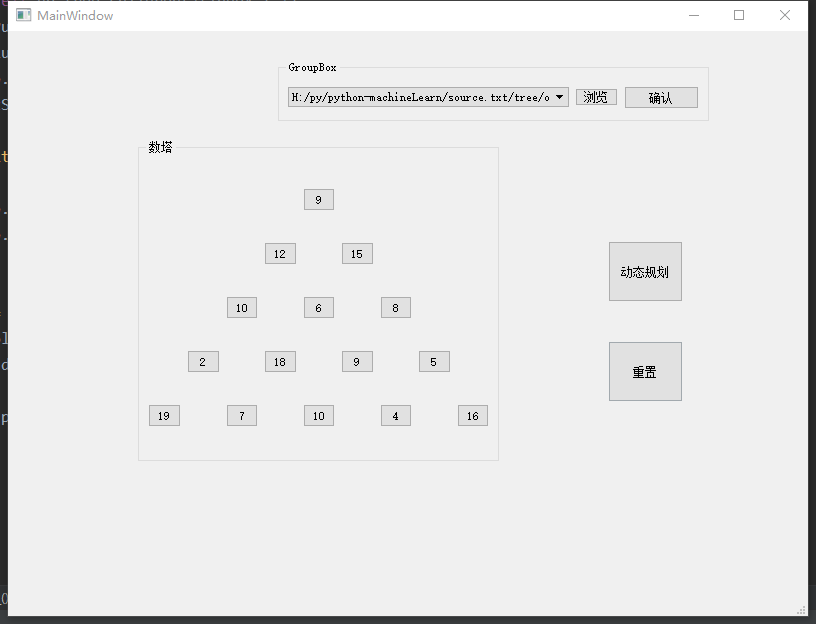


图 6 初始化的树塔

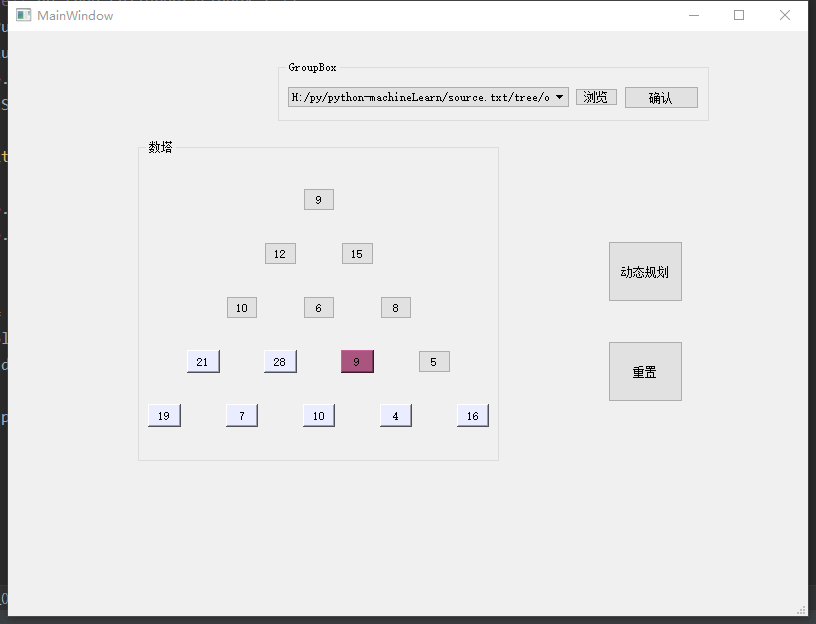


图7 在倒数第二排时的树塔变化情况

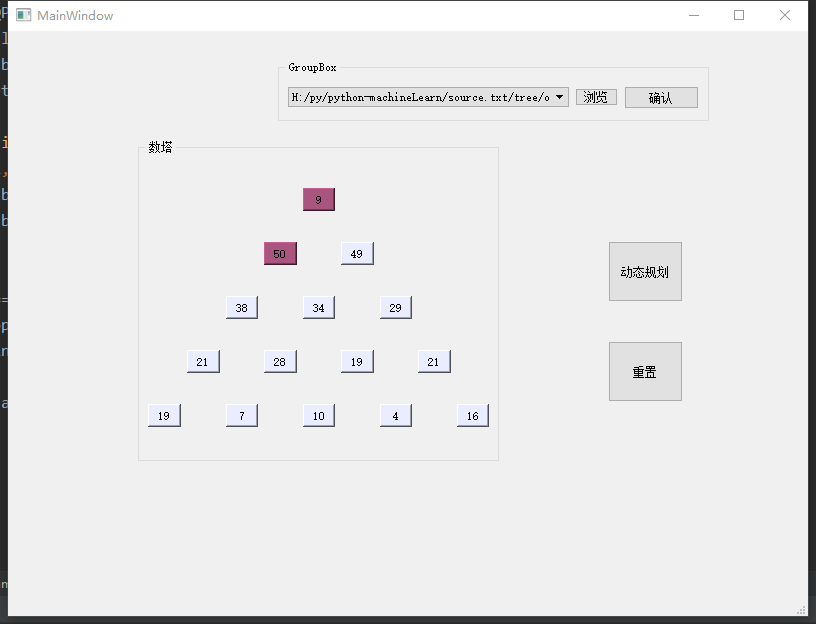


图8 选择有孩纸中较大者

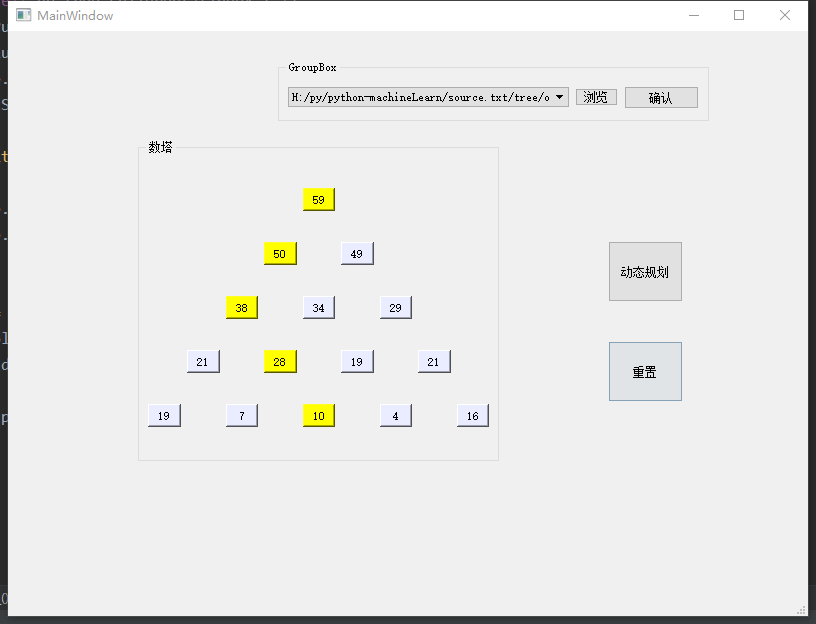


图 9 自底向上遍历完后树塔状态

运行结果分析：

动态规划核心:从解决并保存的子问题得到当前问题的最优解，如图8所示，选择了50作为下一方块(紫色填充表示选择该方块)

由图5、6知算法自底向上，遍历生成如图7所示的每个点的最大收获量数组。

对于图9最后的黄色方块代表路径，为对动态规划生成的最大收获数组递归回溯所得的最优路径：

59->50->38->28->10

## 2.4 题目C2地图着色问题

### 2.4.1 问题描述

已知中国地图，对各省进行着色，要求相邻省所使用的颜色不同，并保证使用的颜色总数最少。设计对图进行着色的算法，分析该算法的时间空间复杂度，并利用该算法实现对中国地图着色。

### 2.4.2问题分析与解决思路

首先将地图区域之间的邻接关系抽象为图上点与点的邻接关系，所以可以地图着色问题可以转换为一个图问题：已知一个图，要求给图上每个点上色，并保证该点的颜色与它的邻接点的颜色都不相同。

我们可以将整个问题划分为更小的子问题：给一个点上色，确保它的颜色与它的邻接点的颜色都不相同。所以可以利用类似广度优先的策略，遍历图上的每一个点，遍历时给该点贪心地选择与邻接点不同的颜色，即总是从低序列开始选择，知道遍历图上的所有点。

### 2.4.3 模型建立与解决思路

***步骤如下：***

（1）输入相邻关系图 记图节点个数n,边的条数e

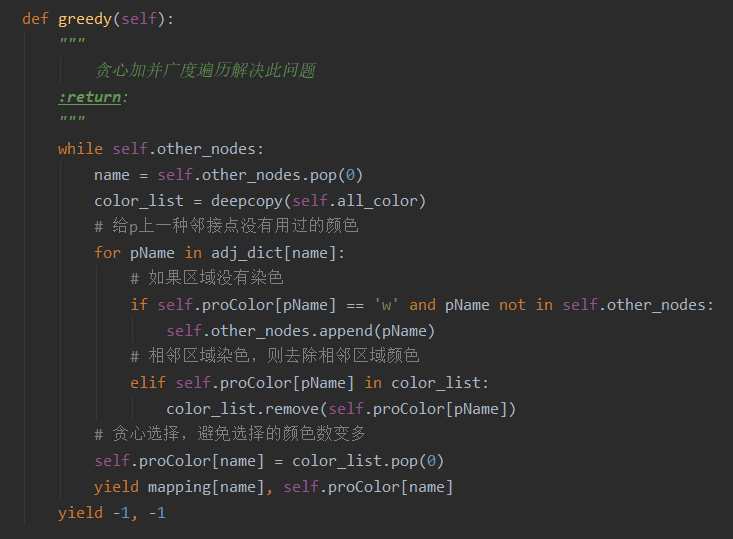
（2）选择一个点为起点加入待遍历队列。

（3）从队列中取出第一个元素，判断它的邻接点的上色情况,

（4）在颜色序列中去除相邻点的颜色，并将未上色点加入队列，取出序列的首元素，为该元素染色(贪心选择)

（5）重复第（2）步直到队列为空。

***伪代码描述如下：***





### 2.4.4 算法实现与复杂度分析

设顶点数，即区域数为v,边数，即顶点数为e

使用到的数据结构为队列。

**复杂度分析**

本算法运算时需要对每个点的每个邻接进行判断，为(2×e)次判断操作，及给没上色的点上色，有v次操作，一共(2×e)+v次操作，由于在着色问题当中，边数（所有邻接省份数）比点数（省份数）多，所以时间复杂度可认为是θ(2\*e)。

运算时创建了一个额外的队列来记录待遍历的点，所以空间复杂度为(e)

### 2.4.5 程序实现与运行结果分析

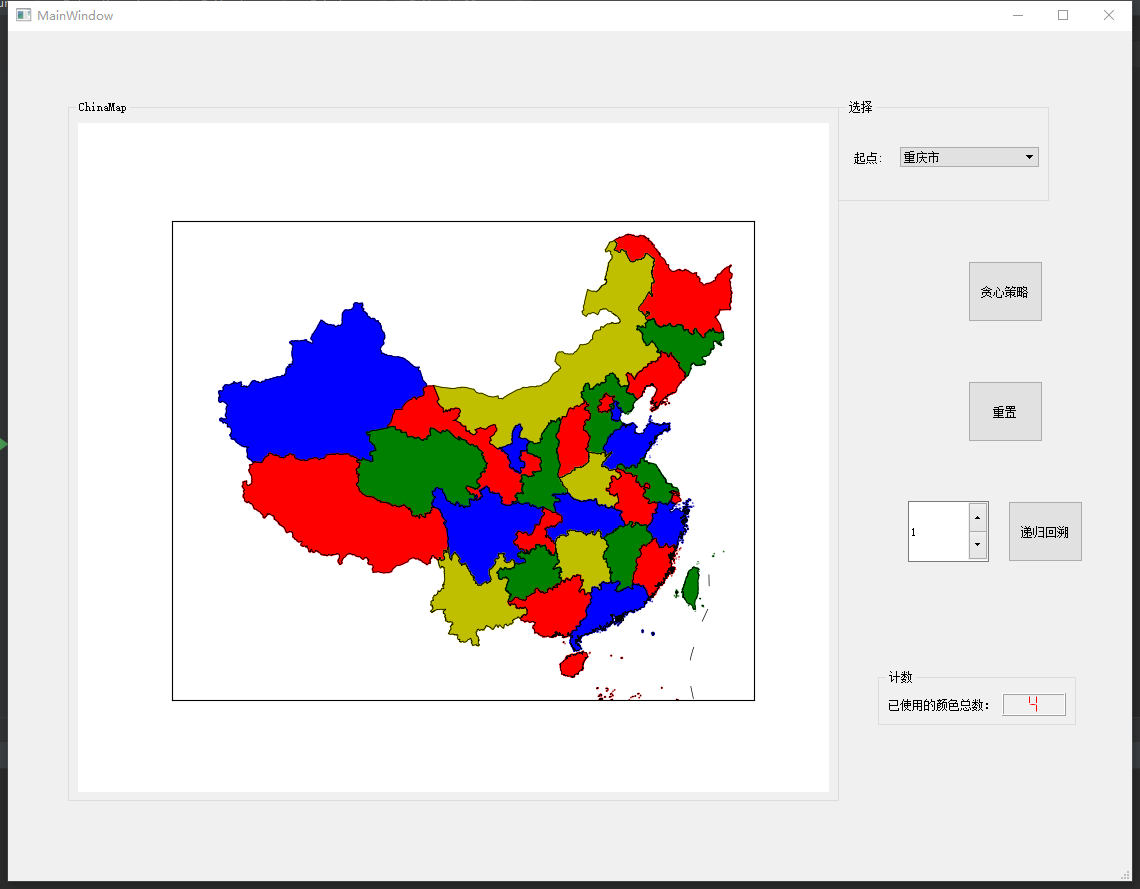


图10 广度遍历(贪心策略)

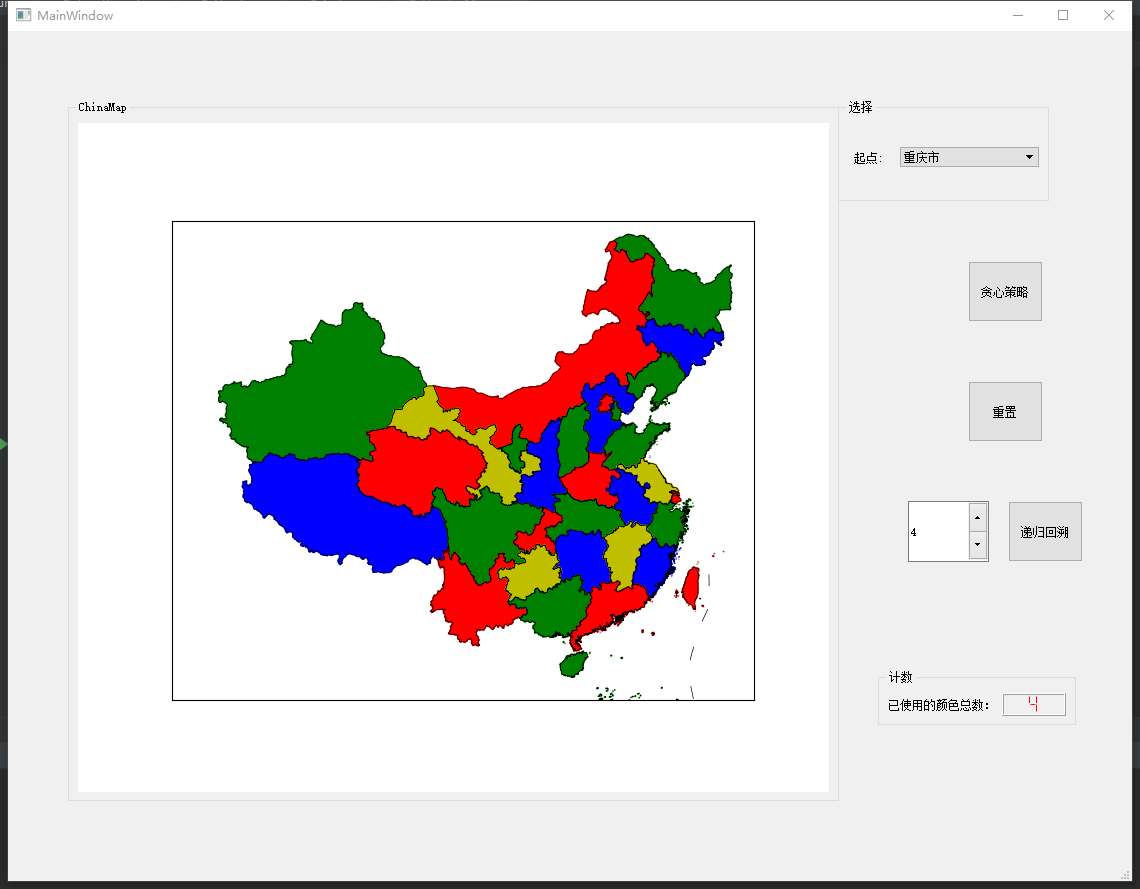


图11 深度遍历(递归回溯)

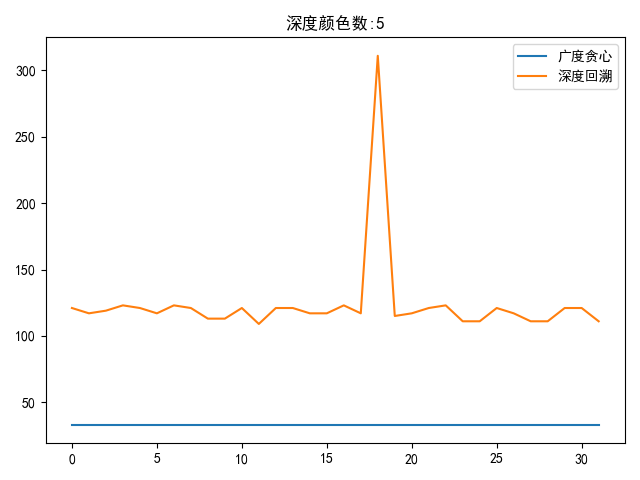


图12深度遍历可使用颜色数:5

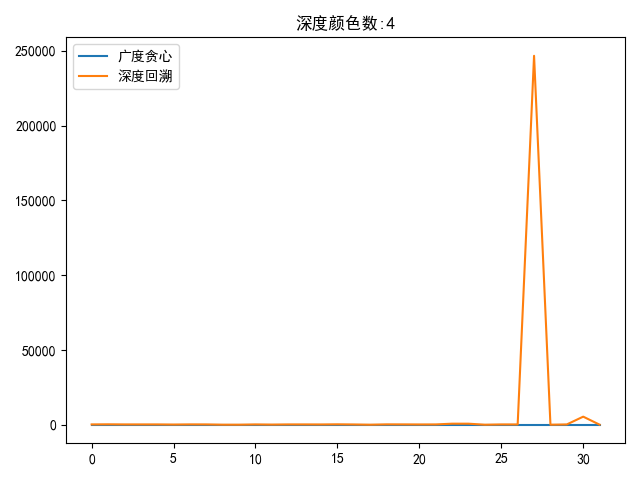


图13 深度遍历可使用颜色数:4

**结果分析:**

图10、图12分别为以重庆市为起点的广度贪心、深度回溯(可使用颜色数4)的地图着色结果。

图12、图13 分别为以各区域为起点，可使用颜色为5，4的各个区域对比图。即横坐标代表了以32个区域(被当作算法起点)，纵坐标代表了将地图染色成功的操作次数（对一个区域染色和取消对一个区域染色都算一次操作）

对于图12，蓝色线(广度贪心策略)为常数32：贪心心算法的核心：一旦做出决定，就无法改变，即广度贪心算法只有染色一种操作、即对32个区域染色的染色次数为32,所以不管起点是哪个区域，最终染色的操作数都为32；橙色线(深度回缩)线段趋势陡峭，有染色和撤销染色两种操作，即起点不同，操作次数也不可预知。

对于图12、图13可使用颜色分别为5,4的深度回溯的操作对比，可使用的颜色越少，算法对区域颜色的要求越高，即可用的排列组合(符合要求的各区域颜色组合)就越少。图13表现尤其明显，当可用颜色种类为4(最少颜色数)，以某区域为起点，需要整整25万次操作才能将按要求将中国染色（才深度遍历到可用组合）。

即对实际情况，能用贪心就用贪心，速度快，且能得到次优解，甚至对于某些问题，如最小生成树问题其一定能得到最优解。

## 2.5 题目B1：基因序列比较

### 2.5.1问题描述

设计算法，计算两给定基因序列的相似程度。

人类基因由4种核苷酸，分别用字母ACTG表示。要求编写一个程序，按以下规则比较两个基因序列并确定它们的相似程度。即给出两个基因序列AGTGATG和GTTAG，它们有多相似呢？测量两个基因相似度的一种方法称为对齐。使用对齐方法可以在基因的适当位置加入空格，让两个基因的长度相等，然后根据基因的分值矩阵计算分数

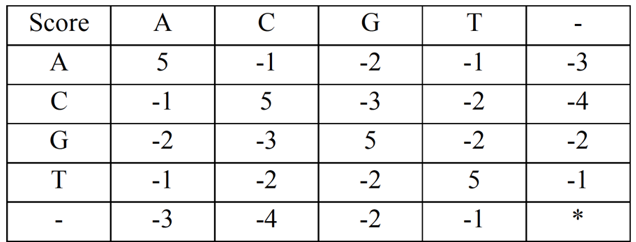


图10 基因的分值矩阵表

例：比较AGTGATG与GTTAG

第一种对齐方案为：

首先可以给AGTGATG插入一个空格得：AGTGAT-G

GTTAG插入3个空格即得：-GT--TAG

上面的匹配分值为:-3+5+5+(-2)+(-3)+5+(-3)+5=9.

第二种对齐方案为：

AGTGATG

-GTTA-G

得到的分值为：(-3)+5+5+(-2)+5+(-1)+5=14.

### 2.5.2 问题分析与解决思路

设两条序列分别为X序列和Y序列，长度分别为m和n。

定义一个表结构来抽象该问题。比较两个序列的相似度，可以理解为寻找一条由坐标（0,0）到坐标（m-1，n-1），并且只能向下、向右或者向右下延伸的路径，使得这条路径得分最高。

如下图是例子中，第二中对齐方案（图中“↓”意味着X序列的该核苷酸与Y序列的“-”匹配，“→”则相反）：

抽象后的序列相似度问题：

如果用暴力搜索方法求解该问题，就要穷举X的所有子序列和Y的所有子序列，并对它们进行匹配，记录所有匹配情况下，两条序列的得分。

根据分析，可以得出基因序列比较问题具有最优子结构性质。规模为m，n的问题的解，可以由规模为m-1，n-1的子问题，规模为m-1，n的子问题，和规模为m，n-1的子问题中，分别加上规模为规模为m，n时最后一个字符的得分结果中，最大的那一个。用公式表示为：

s1 = result[i-1][j-1] + get\_score(X[i], Y[j])

s2 = result[i-1][j] + get\_score(X[i], '-')

s3 = result[i][j-1] + get\_score('-', Y[j])

result[i][j] = max(s1,s2,s3)

### 2.5.3模型建立与算法描述

记两条序列分别为X序列和Y序列，长度分别为m和n。

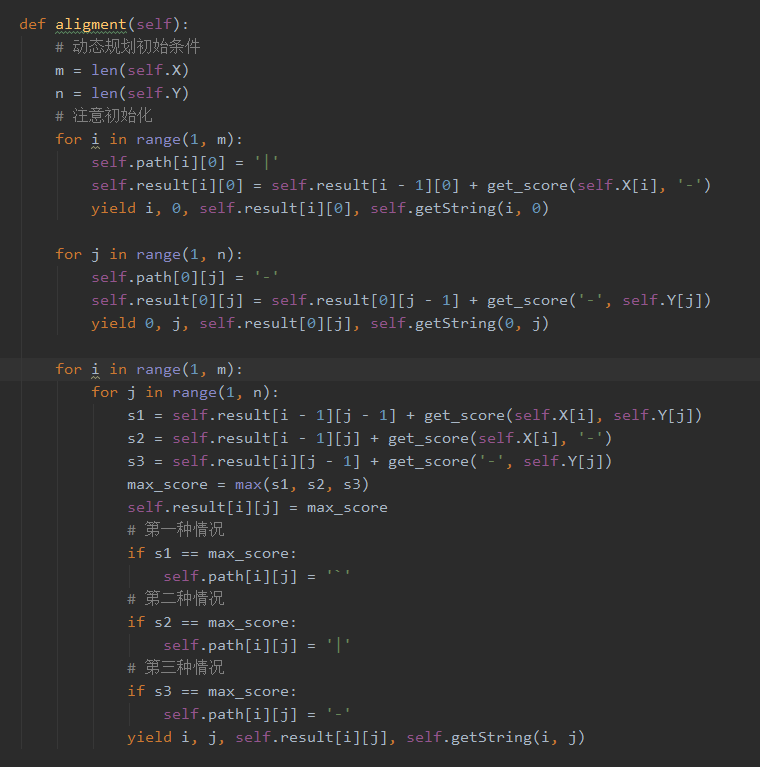
我们将上述分析过程和解决的思路进一步归纳为以下步骤：

（1）初始化结果数组result的第一行及第一列。由于当X的规模m等于1时，意味着出了一个匹配的位置之外，Y序列的其他元素需要和空格匹配，所以该情况下，子问题由result[i][j] = max(s1,s2,s3)改变为result[i][0] = result[i-1][0] + get\_score(X[i], '-')。当Y的规模n等于1时同理。

（2）从i等于2和j等于2开始，根据result[i][j] = max(s1,s2,s3)，由底向上地计算结果，同时也可以利用另一个表用于记录路径。

（3）返回结果表，其中result[m-1][n-1]为该问题的结果。

**核心算法:**



### 2.5.4 算法实现与复杂度分析

初始化时有m+n次赋值操作，加上运算时(m×n)次赋值操作，一共(m×n)+m+n次赋值，所以时间复杂度为θ(m×n)。

运算时创建了一个额外的二维数组——结果表，所以空间复杂度为(m×n)+(m+n)×2，所以空间复杂度为θ(m×n)。

### 2.5.5 程序实现与运行结果分析：

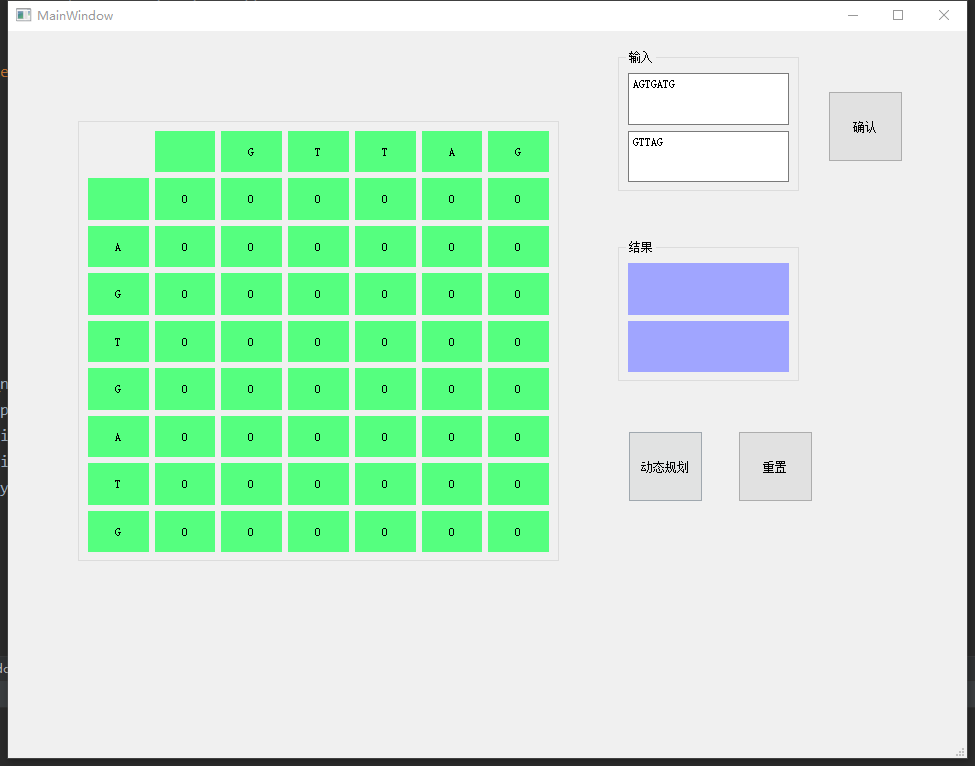


图11 动态规划前

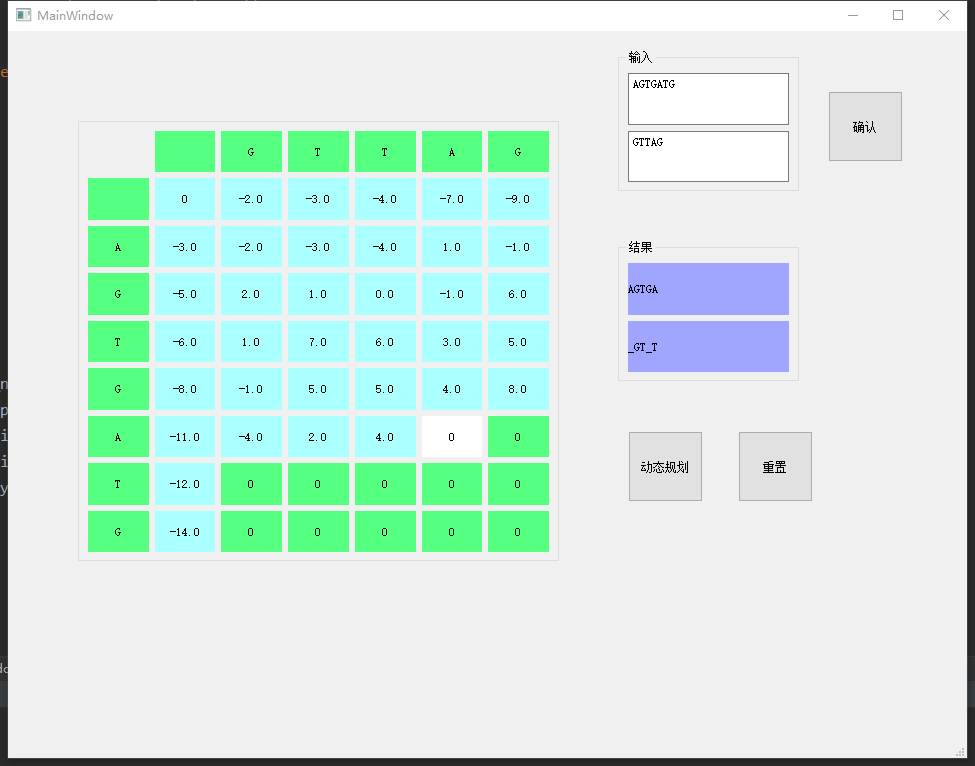


图12动态规划中

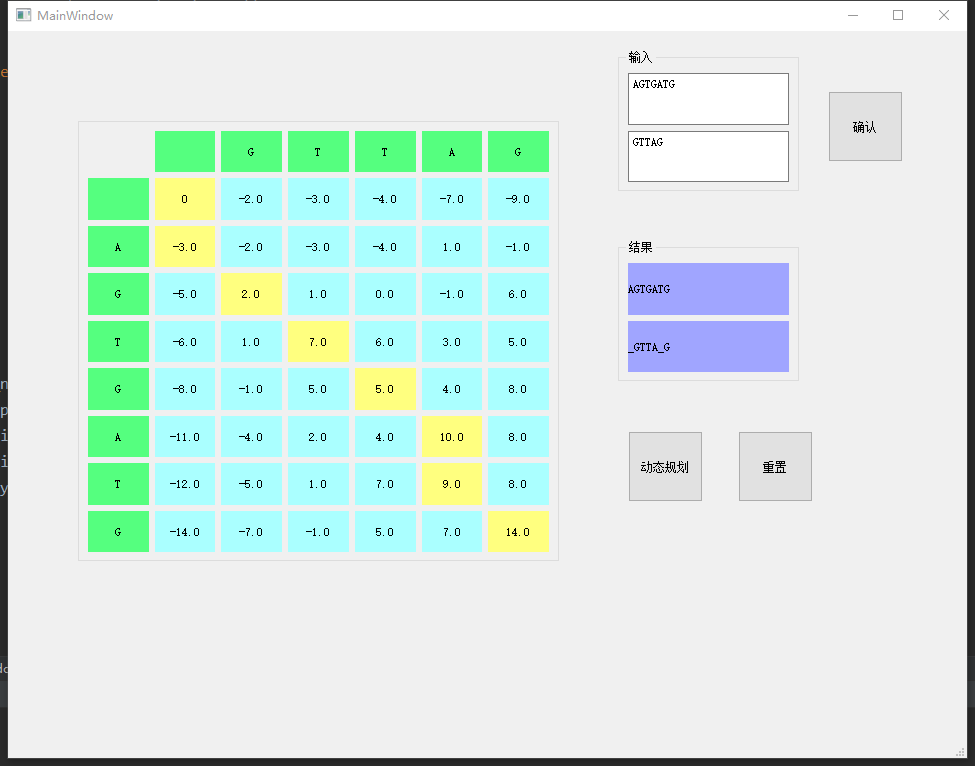


图13 动态规划后



图14 最高得分对齐结果

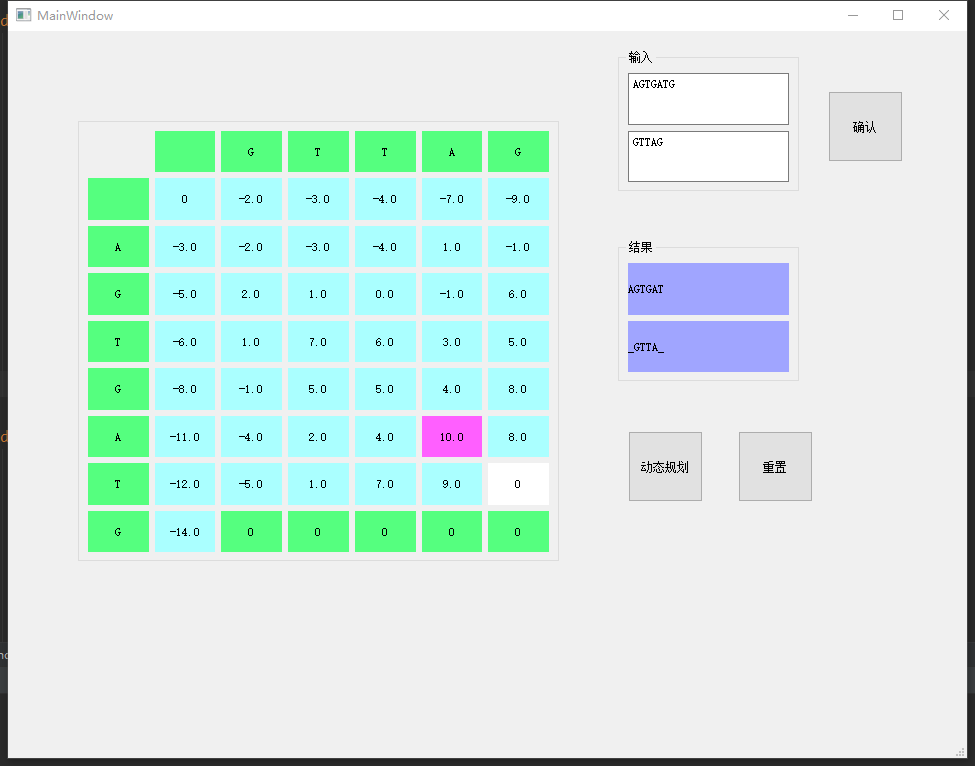


图 15 判断当前结果从哪里来

运行结果分析:

动态规划的核心优势:将已解决的子问题保存下来，如图15所示，算出白色区域的值，不用递归到起始点结束，仅对比上、斜上、左的值加上到达白色区域所得值得最大值即可，即粉红色为算法判断从粉红处所得结果最优。

# 3、总结

## 3.1所遇问题及解决方法

### 3.1.1地图着色问题

问题：刚开始，我采用了广度遍历+贪心的策略来给地图着色，发现以不同的起点出发，最后得到的使用颜色种类数不同（4，5），即这种方式虽然能很快对地图着色，但不一定得到最优解。

解决：采用递归回溯方案，几乎是尝试所有颜色组合可能，当遇到可将地图全都染色时，结束，即该算法能得到结果，这说明使用设定的颜色数能按条件将地图染色，即从可染色数k=1开始计算，若能得出结果则此时k为颜色种类数最小，否则k+1,继续递归回溯，直到能得到结果

### 3.1.2 广度算法的时间复杂度问题

问题：情况不同导致了树高度不同，即最深的路径无法分析。导致最差时间效率无法分析。

## 3.2 心得体会

本次实验，算法可视化界面的展示向我展现了程序的核心--算法--独一无二的魅力。美轮美奂、精致且巧妙的图像的背后是几行简单易理解的几行递归算法。立志最优最快的广度优先策略，从斑驳世界中苦行僧般寻找真理的深度优先策略。不做重复事如同优秀程序员般的动态规划算法，从不瞻前顾后、眼里只有大道的贪心算法。

有了算法，才有了欣欣向荣的计算机世界。