

**数学与信息学院学生实验报告**

**实验课程名称：** 算法分析与设计基础 **教师： \_\_**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **实验项目名称** | **实验二、动态规划算法设计与应用** | | | **实验成绩** |  |
| **学生姓名** |  | **学 号** | **123** | **年级专业班级** |  |
| **小组成员** | **无** | | | **实验日期** | **2019年4 月** |

# 1. 实验目的和要求

## 1.1 实验目的

1.加深对动态规划算法的基本原理的理解，掌握用动态规划方法求解最优化问题的方法步骤及应用；

2.用动态规划设计整数序列的最长递增子序列问题的算法，分析其复杂性，并实现；

3.用动态规划设计求凸多边形的三角剖分问题的算法，分析其复杂性，并实现。

4.选做题：用动态规划设计求解0/1背包问题的算法，分析其复杂性，并实现。

## 1.2 实验软硬件环境

① 操作系统 win10

② 编译环境 java

## 1.3 实验要求

### 1.3.1 最长递增子序列问题

1.问题描述

求一个由n个整数组成的整数序列的最长递增子序列。一个整数序列的递增子序列可以是序列中非连续的数按照原序列顺序排列而成的。 最长递增子序列是其递增子序列中长度最长的。

2. 具体要求（若在ACM平台上提交程序，必须按此要求）――平台上1700题

输入：输入的第一行是一个正整数n，表示测试例个数。接下来几行是n个测试例的数据，每个测试例的数据由两行组成，其中第一行为一个正整数k (k<=500)，表示整数序列的长度，第二行给出整数序列，整数之间用一个空格隔开。（设给出的每个整数序列的最长递增子序列都是唯一的。）

输出：对于每个测试例输出两行，第一行为最长递增子序列的长度，第二行为最长递增子序列，整数之间用一个空格隔开。两个测试例的输出数据之间用一个空行隔开，最后一个测试例后无空行。

3. 测试数据

输入：2

5

3 1 4 2 3

6

1 3 9 5 2 6

20

1 2 7 13 3 5 10 24 12 4 9 16 53 6 83 8 23 11 31 47

输出：3

1 2 3

4

1 3 5 6

10

1 2 3 5 10 12 16 23 31 47

4. 设计与实现的提示

(1) 寻找最优子结构、写出递归方程是问题的关键。

(2) 以Ai为末元素的最长递增子序列(记为S(i))，等于以使S(j), (j=1～i), 最大的那个Aj为末元素的递增子序列最末再加上Ai；如果这样的元素不存在，那么Ai自身构成一个长度为1的以Ai为末元素的递增子序列。

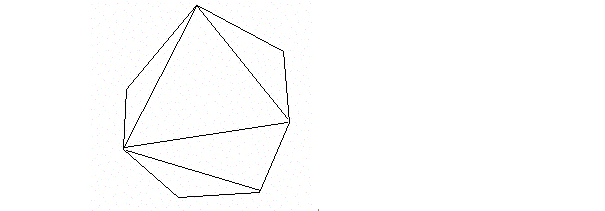
(3) 最优解的信息在此是以Ai为末元素的最长递增子序列的前驱元素，应当记录下来。

5. 扩展内容

本题可以采用多种方法求解，可以尝试用不同思路求解。

### 1.3.2 凸多边形的三角剖分

1. 问题描述

设P是一个有n个顶点的凸多边形，P中的弦是P中连接两个非相邻顶点的线段。用P中的(n-3)条弦将P剖分成(n-2)个三角形（如下图所示）。使得(n-3)条弦的长度之和最小的三角形剖分称为最优三角剖分。

2. 具体要求（若在ACM平台上提交程序，必须按此要求）――平台上1701题

输入：输入的第一行是一个正整数m，表示测试例个数，接下来几行是m个测试例的数据，每个测试例的数据由两行组成，第一行含一个正整数n (n<=500)，表示凸多边形的顶点个数；第二行含2n个实数x1 , y1 , x2 , y2 , …xn , yn ，按顺时针方向依次给出n个顶点的坐标值(xi, yi) i=1, 2, …, n，整数之间用一个空格隔开。

输出：对于每个测试例输出一行，含一个实数（精确到小数点后三位），表示最优三角剖分的n-3条弦的长度之和。两个测试例的输出数据之间用一个空行隔开，最后一个测试例后无空行。

3. 测试数据

输入：

2

6

1 2 2 1.5 2 0.5 1 0 0 0.5 0 1.5

9

723 1220 463 1074 370 842 317 534 524 192 992 87 1378 355 1683 855 1301 1131

输出：

5.606

4928.722

4. 设计与实现的提示

(1) 凸(n+1)边形的最优三角剖分是该凸多边形的所有三角剖分中弦长最短的那个剖分。

(2) 本题与课本7.3节中的矩阵链相乘问题非常相似。寻找最优子结构、写出递归方程是问题的关键，注意递归方程的参数的设置。

(3) 将凸(n+1)边形三角剖分时，注意弦长不要重复计算。

5. 扩展内容

本题只要求计算出最优值。如果要求最优解，在求最优值过程中，要记录下改最优值对应的三角剖分的信息。

### 1.3.3 选做题――0/1背包问题

1. 问题描述

设有一个容量为C的背包，n个物品的集合U={u1, u2, …, un}，物品uj的体积和价值分别为sj和vj，C, sj, vj都是正整数。在U中选择物品装入背包，使得装入背包的物品总价值最大。设每种物品或完全装入或完全不装入背包。

2. 具体要求

输入：输入的第一行是一个正整数m，表示测试例个数，接下来几行是m个测试例的数据。每个测试例的数据由三行组成，第一行含两个正整数n和C，其中， n (n<=100)表示给定的是n个物品的集合U={u1, u2, …, un}，C(C<=5000)表示背包的容量；第二行含n个整数s1 , s2 , …sn，表示n个物品的体积；第三行含n个整数v1 , v2 , …vn，表示n个物品的价值，整数之间用一个空格隔开。

输出：对于每个测试例输出2行数据，其中，第一行含一个整数，表示装入背包物品的最大总价值；第2行含n个整数x1 , x2 , …xn，表示u1, u2, …, un 这n个物品是否放入背包。其中xi ={1, 0} , (i=1,2, …,n)。 如果xi =1，表示物品ui放入背包；如果xi =0，表示物品ui不放入背包。每个xi之间用一个空格隔开，两个测试例的输出数据之间用一个空行隔开，最后一个测试例后无空行。

3. 测试数据

输入：2

5 20

7 13 6 4 3

1. 7 3 2 3
2. 100

7 13 45 25 16 75 48 32

6 5 20 10 7 32 22 13

输出：13

1 0 1 1

48

1 0 1 0 0 0 1 0

# 2. 实验记录

## 2.1 基本原理

动态规划是一种非常重要的程序设计方法，常用于求解最优化问题。最优化问题：给定若干个约束条件和一个目标函数，在某指定集合中求满足所有约束条件的且使得目标函数值达最大或最小的元素和相应的目标函数值，即：问题的最优值和最优解。

适用动态规划求解的问题的基本要素：

(1)满足最优性原理：即

一个最优化问题的最优解包含了其子问题的最优解。

(2)

无后向性：即某阶段状态一旦确定，就不受这个状态以后决策的影响。也即，某状态以后的过程不会影响以前的状态，只与当前状态有关，这种特性也被称为无后效性。

(2)具有重叠的子问题：即问题被分解成的子问题存在互相重叠。动态规划方法对于这些重叠的子问题只求解一次，以提高算法的效率。

## 2.2 该类算法设计与实现的要点

动态规划算法求解最优化问题的步骤：

(1) 找出问题的最优子结构。分析问题的最优解（最优值）的结构特征。

(2) 递归地定义最优值。 根据最优子结构，确定最优值所满足的递归公式。

(3) 计算最优值。根据最优值的递归公式，采用自底向上的迭代或自顶向下的递归，计算最优值。

(4) 构造最优解。在求解最优值的过程中要记录下得到最优值的相应最优解的信息，并根据该信息构造最优解。

注意：在计算最优值时应保存相应的信息：

(a) 已经求出的子问题的最优值（避免重复计算）。

(b) 最优解的有关信息。

动态规划算法求解其它问题的步骤：

(1) 根据最优化原理分析问题的解的结构。

(2) 递归地定义问题的解。

(3) 计算问题的解。 根据解的递归公式，自底向上或自顶向下地计算解，计算过程中注意保存已经求出的子问题的解。

其中，自底向上方法通过迭代来实现，适用于所有的子问题都需要解的情况，实现时要注意根据递归公式正确确定子问题的求解顺序。自顶向下方法通过递归来实现，适用于不必解所有的子问题的情况，实现时要注意标记子问题是否计算过，同一个子问题只在第一次递归调用时计算并存储结果。

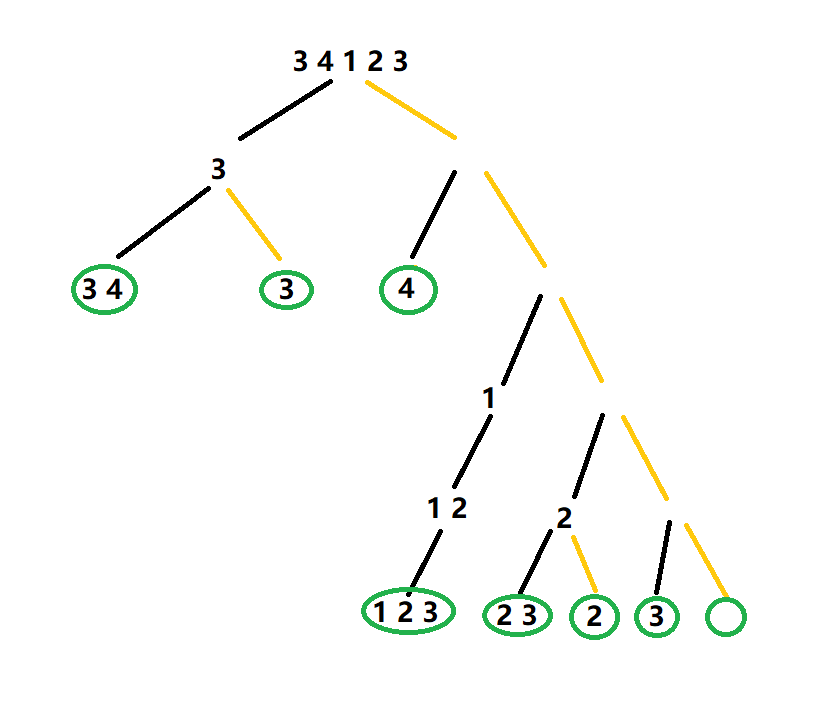
## 2.3 实验过程

### 2.3.1 实验思路

#### 2.3.1.1 最长递增子序列

**递归实现思路：**

创建一个结果字符串，用于保存最长子序列。对于原序列中的每个数，若满足条件可以选择放入或不放入，若不满足条件，则不可放入。通过递归的思想，可以很快将序列的所有可能结果得出，再通过判断其大小，从而选出最大的序列。



黑线代表放入符合条件的下一个数，黄线表示不放入，绿色椭圆里的序列即为最后可能得到的结果（上图得到123的支线还没画完整）。

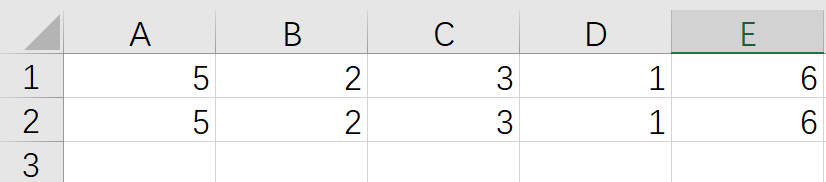
以34123为例，一开始可以选择往答案序列放入3或不放入3，放入3后可以选择放入4或不放入4（放入的数必须大于前一个放入的数），一直这样下去，直到读到原始序列的最后一个数或再也找不出符合条件的数，最后通过比较，得到答案序列最长的序列。

**迭代实现思路：**

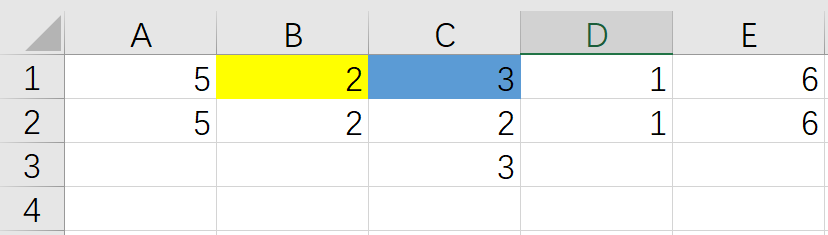
最长递增子序列问题中，递归和迭代的实现思路是不一样的，迭代需要通过数组来保存我们需要的一些数据，并在其他子问题中运用这些数据。

创建一个二维数组，用来保存每个数的最长子序列，通过遍历数组，将每个数的最长子序列搬到更大的数下面，形成更长的序列。可以用一个一维数组来保存每个数的最长子序列长度，在程序最后进行遍历判断，即可输出最长子序列。

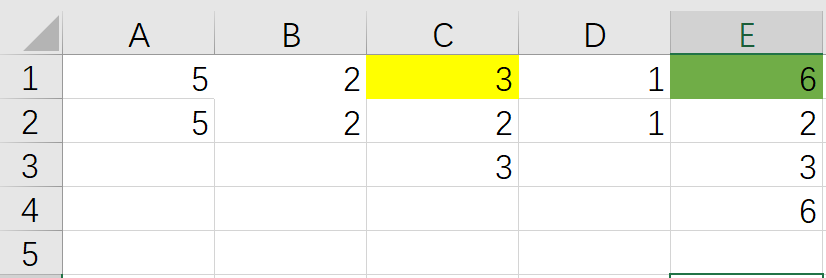
执行前的数组：

****

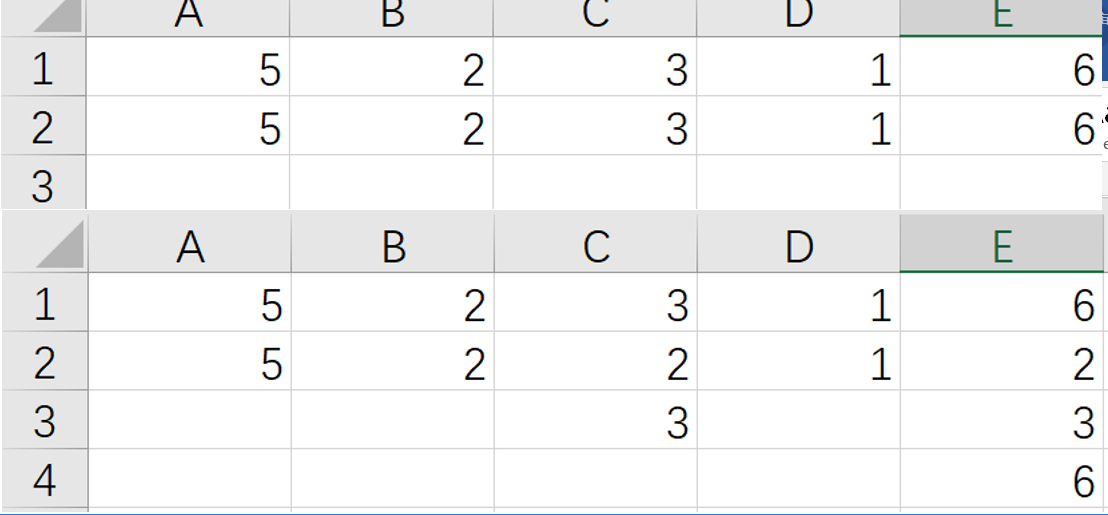
**2<3：将2列的数搬到3下面，再重新将3写入3列的结尾**

****

**3<6：将3列的数搬到6下面，再重新将6写入6列的结尾**

****

执行前后数组对比：



#### 2.3.1.2 0/1背包问题

背包问题与之前的递增子序列问题类似，对于每个物品，若背包容量足够，可以选择放入或不放入，放入后背包的容量应减去物品的体积，容量不足则不可放入该物品。按照之前的思路，就可以沿用前面的方法：用递归的方式得到每种可能的结果，再判断结果得到的价值总量，选出价值总量最大的即可。

#### 2.3.1.3 最优三角剖分

三角剖分问题与另外两题有些不同，每次要连接的点可以有多种选择，所以不得不将它们一一保存下来。相似的是都要找出所有情况，然后进行比较。

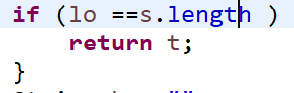
### 2.3.2 程序代码（包含递归与迭代两种实现）

#### 2.3.2.1 最长递增子序列

递归关键代码：



递归出口：



递归代码：

public class Long\_list **{**

public String list**(**char s**[])** **{**

String t **=** ""**;**

**return** l**(**s**,** t**,** 0**);**

**}**

private String l**(**char s**[],** String t**,** int lo**)** **{**// 从头到尾判断符合条件的每一位放入与不放入的情况

**if** **(**lo **>** s**.**length **-** 1**)** **{**// 最后一位已经判断完 出口1

**return** t**;**

**}**

String k **=** ""**;**

**for** **(**int i **=** lo**;** i **<** s**.**length**;** i**++)** **{**

**if** **((**t**.**length**()** **==** 0**)** **||** **((**t**.**length**()** **-** 1 **>=** 0**)** **&&** t**.**charAt**(**t**.**length**()** **-** 1**)** **<** s**[**i**]))** **{**

k **=** t **+** s**[**i**];**// 连接

**break;**

**}**

**}**

**return** max**(**l**(**s**,** t**,** lo **+** 1**),** l**(**s**,** k**,** lo **+** 1**));**

**}**

private String max**(**String l**,** String l2**)** **{**

**return** l**.**length**()** **>=** l2**.**length**()** **?** l **:** l2**;**

**}**

public static void main**(**String**[]** args**)** **{**

char c**[]** **=** **{** '3'**,** '1'**,** '4'**,** '2'**,** '3' **};**

Long\_list ll **=** **new** Long\_list**();**

System**.**out**.**println**(**ll**.**list**(**c**));**

**}**

**}**

**迭代实现代码：**

public class te **{**

public static void main**(**String**[]** args**)** **{**

int**[]** randArr **=** **new** int**[]** **{**3**,**4**,**1**,**2**,**3**};**

maxAscLen**(**randArr**);**

**}**

public static void maxAscLen**(**int**[]** arr**)** **{**

int n **=** arr**.**length**;**

int**[]** lens **=** **new** int**[**n**];**// 保存以每个元素i结尾的递增子序列长度

int**[][]** lensArr **=** **new** int**[**n**][**n**];**// 保存以每个元素i结尾的递增子序列

**for** **(**int i **=** 0**;** i **<=** n **-** 1**;** i**++)** **{**// 初始化辅助空间

lens**[**i**]** **=** 1**;**

lensArr**[**i**][**0**]** **=** arr**[**i**];**

**}**

int curMaxLen**;**

**for** **(**int i **=** 0**;** i **<=** n **-** 1**;** i**++)** **{**//3 4 1 2 3

curMaxLen **=** 1**;**

**for** **(**int j **=** i **-** 1**;** j **>=** 0**;** j**--)** **{**// 从后往前寻找

**if** **(**arr**[**i**]** **>** arr**[**j**]** **&&** lens**[**j**]** **+** 1 **>** curMaxLen**)** **{**

curMaxLen **=** lens**[**j**]** **+** 1**;**// 更新以元素i结尾的最长递增子序列长度

lens**[**i**]** **=** curMaxLen**;**

**for** **(**int k **=** 0**;** k **<** lens**[**j**];** k**++)** **{**// 把以元素j结尾的最长递增子序列拷贝到以元素i结尾的最长递增子序列中

lensArr**[**i**][**k**]** **=** lensArr**[**j**][**k**];**

**}**

lensArr**[**i**][**lens**[**i**]** **-** 1**]** **=** arr**[**i**];**

**}**

**}**

**}**

// 寻找最大递增子序列长度的元素下标

// 这里只能找到第一个最长递增子序列的下标

int index **=** 0**;**

**for** **(**int i **=** 0**;** i **<=** n **-** 1**;** i**++)** **{**

**if** **(**lens**[**index**]** **<** lens**[**i**])** **{**

index **=** i**;**

**}**

**}**

// 这里只能输出一个最长递增子序列

System**.**out**.**println**(**"最大递增子序列长度：" **+** lens**[**index**]);**

System**.**out**.**print**(**"最大递增子序列为："**);**

**for** **(**int i **=** 0**;** i **<=** lens**[**index**]** **-** 1**;** i**++)** **{**

System**.**out**.**print**(**lensArr**[**index**][**i**]** **+** " "**);**

**}**

System**.**out**.**println**();**

// return lens[index]; //可以返回最大最大递增子序列长度

**}**

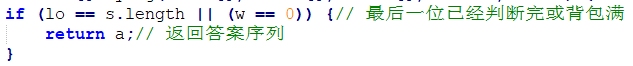
**}**

#### 2.3.2.2 0/1背包问题

递归关键代码：



递归出口：



递归代码：

**import** java**.**util**.**Scanner**;**

public class Bag **{**

public int**[]** Mybag**(**int s**[],** int v**[])** **{**

int**[]** ans **=** **new** int**[**s**.**length**];**

int w **=** 20**;**

**return** mybag**(**s**,** v**,** ans**,** 0**,** w**);**

**}**

private int**[]** mybag**(**int s**[],** int v**[],** int a**[],** int lo**,** int w**)** **{**

**if** **(**lo **==** s**.**length **||** **(**w **==** 0**))** **{**// 最后一位已经判断完或背包满 出口1

**return** a**;**// 返回答案序列

**}**

int**[]** last **=** a**.**clone**();**

int w2 **=** w**;**

**for** **(**int i **=** lo**;** i **<** v**.**length**;** i**++)** **{**

**if** **(**w **>=** s**[**i**])** **{**

last**[**i**]** **=** v**[**i**];**

lo **=** i**;**// lo跳跃

w2 **-=** s**[**i**];**

**break;**

**}**

**}**

**return** max**(**mybag**(**s**,** v**,** a**,** lo **+** 1**,** w**),** mybag**(**s**,** v**,** last**,** lo **+** 1**,** w2**));**

**}**

private int**[]** max**(**int n1**[],** int n2**[])** **{**

int v1 **=** 0**,** v2 **=** 0**;**

**for** **(**int i **=** 0**;** i **<** 5**;** i**++)** **{**

v1 **+=** n1**[**i**];**

v2 **+=** n2**[**i**];**

**}**

**return** v1 **>=** v2 **?** n1 **:** n2**;**

**}**

public static void main**(**String**[]** args**)** **{**

Bag bag **=** **new** Bag**();**

int s**[]** **=** **{** 7**,** 14**,** 6**,** 4**,** 3 **},** v**[]** **=** **{** 5**,** 8**,** 3**,** 2**,** 3 **};**

int**[]** answer **=** bag**.**Mybag**(**s**,** v**).**clone**();**

**for** **(**int i **=** 0**;** i **<** 5**;** i**++)** **{**

System**.**out**.**println**(**"价值: "**+**answer**[**i**]+**" 重量: "**+**s**[**i**]);**

**}**

**}**

**}**

迭代代码：

#include <iostream>

using namespace std**;**

**typedef** struct commodity

**{**

int value**;** //价值

int weight**;** //重量

**}**commodity**;**

const int N **=** 3**;** //物品数量

const int W **=** 50**;** //背包总重量

commodity goods**[**N**+**1**]={{**0**,**0**},{**230**,**40**},{**100**,**20**},{**120**,**30**}};**

int select**[**N**+**1**][**W**+**1**];**

int max\_value**();**

int main**()**

**{**

int maxvalue **=** max\_value**();**

cout**<<**"The max value is: "**;**

cout**<<**maxvalue**<<**endl**;**

int remainspace **=** W**;**

//打印出选中的物品

**for(**int i**=**N**;** i**>=**1**;** i**--)**

**{**

**if** **(**remainspace **>=** goods**[**i**].**weight**)**

**{**

**if** **((**select**[**i**][**remainspace**]-**select**[**i**-**1**][**remainspace**-**goods**[**i**].**weight**]==**goods**[**i**].**value**))**

**{**

cout **<<** "item " **<<** i **<<** " is selected!" **<<** endl**;**

remainspace **=** remainspace **-** goods**[**i**].**weight**;**//打印出该物品并将当前重量减去物品质量

**}**

**}**

**}**

**return** 0**;**

**}**

int max\_value**()**

**{**//计算最大价值

**for(**int w**=**1**;**w**<=**W**;++**w**)**

select**[**0**][**w**]** **=** 0**;**

**for(**int i**=**1**;**i**<=**N**;++**i**)**

**{**

select**[**i**][**0**]** **=** 0**;**

**for(**int w**=**1**;**w**<=**W**;++**w**)**

**{**

**if(**goods**[**i**].**weight **<=** w**)**

**{**

**if(** **(**goods**[**i**].**value **+** select**[**i**-**1**][**w**-**goods**[**i**].**weight**])** **>** select**[**i**-**1**][**w**]){**

select**[**i**][**w**]** **=** goods**[**i**].**value **+** select**[**i**-**1**][**w**-**goods**[**i**].**weight**];**

**}**

**else**

select**[**i**][**w**]** **=** select**[**i**-**1**][**w**];**

**}**

**else** //不放入

select**[**i**][**w**]** **=** select**[**i**-**1**][**w**];**

**}**

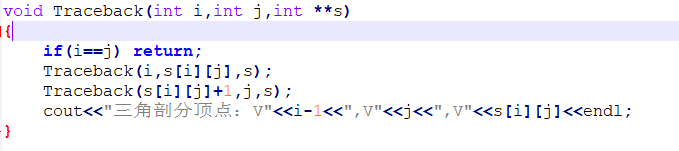
**}**

**return** select**[**N**][**W**];** //返回判断数组

**}**

#### 2.3.2.3 最优三角剖分

关键部分：



#include "stdafx.h"

#include <iostream>

using namespace std**;**

const int N **=** 7**;**//凸多边形边数+1

int weight**[][**N**]** **=** **{{**0**,**2**,**2**,**3**,**1**,**4**},{**2**,**0**,**1**,**5**,**2**,**3**},{**2**,**1**,**0**,**2**,**1**,**4**},{**3**,**5**,**2**,**0**,**6**,**2**},{**1**,**2**,**1**,**6**,**0**,**1**},{**4**,**3**,**4**,**2**,**1**,**0**}};**//凸多边形的权

int MinWeightTriangulation**(**int n**,**int **\*\***t**,**int **\*\***s**);**

void Traceback**(**int i**,**int j**,**int **\*\***s**);**//构造最优解

int Weight**(**int a**,**int b**,**int c**);**//权函数

int main**()**

**{**

int **\*\***s **=** new int **\*[**N**];**

int **\*\***t **=** new int **\*[**N**];**

**for(**int i**=**0**;**i**<**N**;**i**++)**

**{**

s**[**i**]** **=** new int**[**N**];**

t**[**i**]** **=** new int**[**N**];**

**}**

cout**<<**"此多边形的最优三角剖分值为："**<<**MinWeightTriangulation**(**N**-**1**,**t**,**s**)<<**endl**;**

cout**<<**"最优三角剖分结构为："**<<**endl**;**

Traceback**(**1**,**5**,**s**);** //s[i][j]记录了Vi-1和Vj构成三角形的第3个顶点的位置

**return** 0**;**

**}**

int MinWeightTriangulation**(**int n**,**int **\*\***t**,**int **\*\***s**)**

**{**

**for(**int i**=**1**;** i**<=**n**;** i**++)**

**{**

t**[**i**][**i**]** **=** 0**;**

**}**

**for(**int r**=**2**;** r**<=**n**;** r**++)** //r为当前计算的链长（子问题规模）

**{**

**for(**int i**=**1**;** i**<=**n**-**r**+**1**;** i**++)**//n-r+1为最后一个r链的前边界

**{**

int j **=** i**+**r**-**1**;**//计算前边界为r，链长为r的链的后边界

t**[**i**][**j**]** **=** t**[**i**+**1**][**j**]** **+** Weight**(**i**-**1**,**i**,**j**);**//将链ij划分为A(i) \* ( A[i+1:j] )这里实际上就是k=i

s**[**i**][**j**]** **=** i**;**

**for(**int k**=**i**+**1**;** k**<**j**;** k**++)**

**{**

//将链ij划分为( A[i:k] )\* (A[k+1:j])

int u **=** t**[**i**][**k**]** **+** t**[**k**+**1**][**j**]** **+** Weight**(**i**-**1**,**k**,**j**);**

**if(**u**<**t**[**i**][**j**])**

**{**

t**[**i**][**j**]** **=** u**;**

s**[**i**][**j**]** **=** k**;**

**}**

**}**

**}**

**}**

**return** t**[**1**][**N**-**2**];**

**}**

void Traceback**(**int i**,**int j**,**int **\*\***s**)**

**{**

**if(**i**==**j**)** **return;**

Traceback**(**i**,**s**[**i**][**j**],**s**);**

Traceback**(**s**[**i**][**j**]+**1**,**j**,**s**);**

cout**<<**"三角剖分顶点：V"**<<**i**-**1**<<**",V"**<<**j**<<**",V"**<<**s**[**i**][**j**]<<**endl**;**

**}**

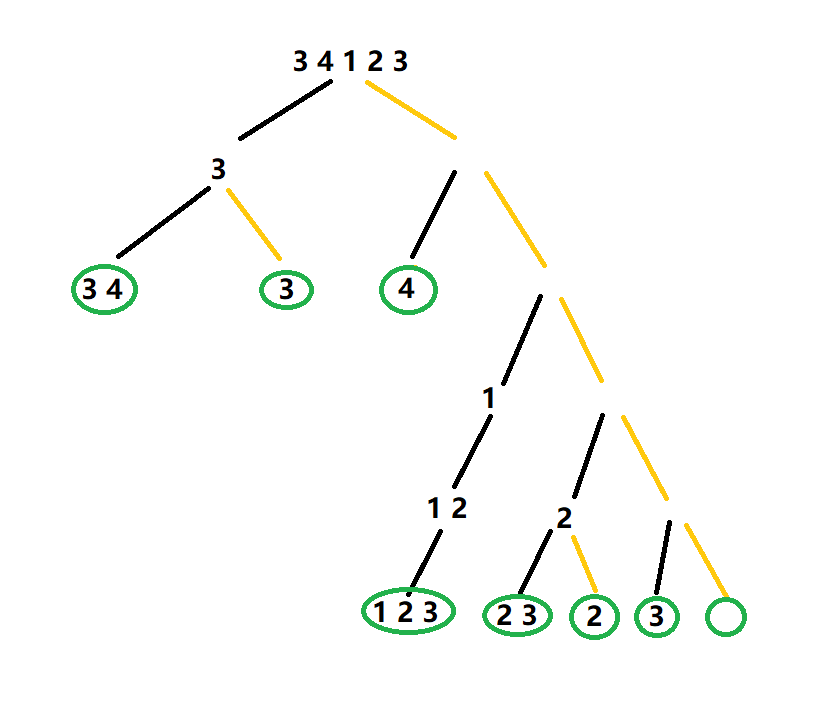
int Weight**(**int a**,**int b**,**int c**)**

**{**

**return** weight**[**a**][**b**]** **+** weight**[**b**][**c**]** **+** weight**[**a**][**c**];**

**}**

### 2.3.3 复杂度分析及代码优化



**最长序列递归时间复杂度**：**O（2^n）**

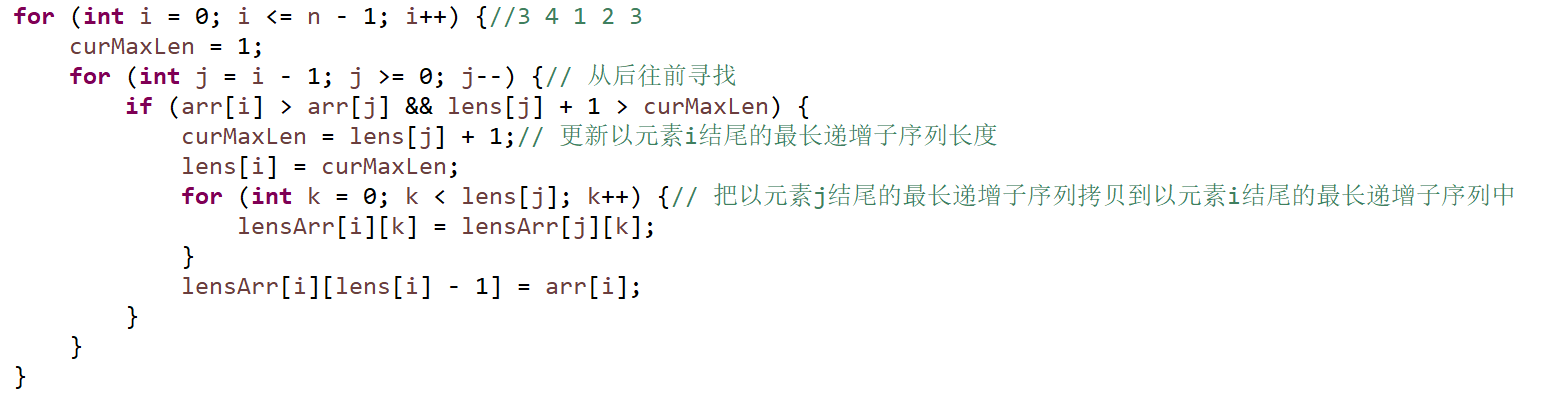
对于子序列和背包问题，把问题最小化，每个子问题都有两种选择，即时间复杂度为

O（2^n）,但在一般情况下，程序并没有对每个子问题都进行处理，在实际情况中，程序在判断了子问题不符合条件时，会跳过该子问题。而且每种选择导致的结果在进入出口的时间也是不同的，这也是造成上图分枝长度不一的原因。所以实际的复杂度也将小于

O（2^n）。

**最长序列迭代时间复杂度：0(n^3)**

迭代核心部分代码：3个for循环完成数组的数据修改。

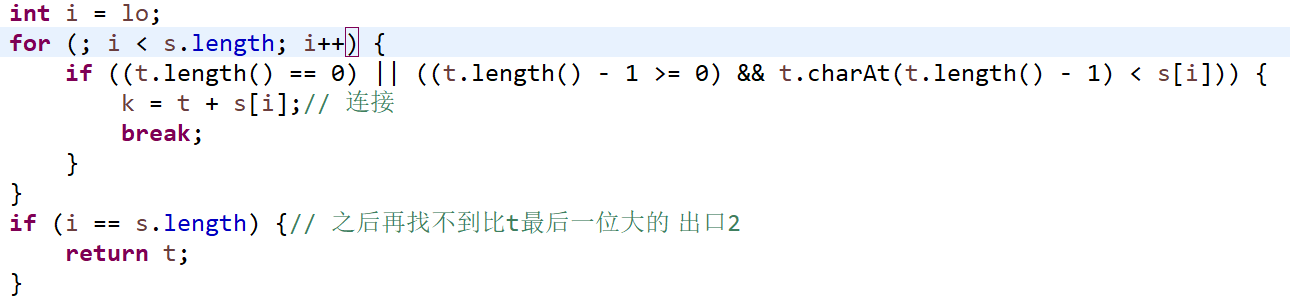
****

**最优三角剖分时间复杂度：0(n^3)**

1. **for**(**int** r=2; r<=n; r++) //r为当前计算的链长（子问题规模）
2. {
3. **for**(**int** i=1; i<=n-r+1; i++)//n-r+1为最后一个r链的前边界
4. {
5. **int** j = i+r-1;//计算前边界为r，链长为r的链的后边界
6. t[i][j] = t[i+1][j] + Weight(i-1,i,j);//将链ij划分为A(i) \* ( A[i+1:j] )这里实际上就是k=i
7. s[i][j] = i;
8. **for**(**int** k=i+1; k<j; k++)
9. {
10. //将链ij划分为( A[i:k] )\* (A[k+1:j])
11. **int** u = t[i][k] + t[k+1][j] + Weight(i-1,k,j);
12. **if**(u<t[i][j])
13. {
14. t[i][j] = u;
15. s[i][j] = k;
16. }
17. }
18. }
19. }

**递归出口问题**：

在经过思考后发现，这两个问题在递归的出口上不止一个，添加多个合理的出口能减小程序的工作量。例如上图序列如果第一个放入的是4，则4已经是序列最大的数，但下标还没到达序列的尾部，继续执行分枝只会浪费时间，此时应也视为程序的出口。



### 2.3.4 实现结果

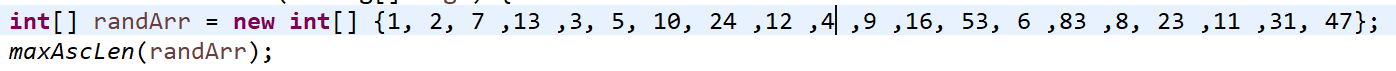
#### 2.3.4.1

初始序列：

递归：

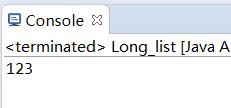


迭代：

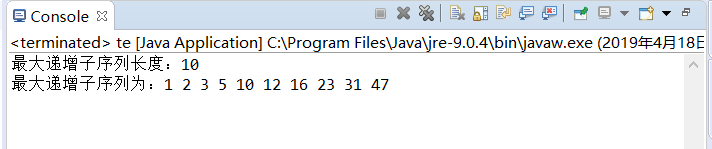


得到结果：

递归：



迭代:



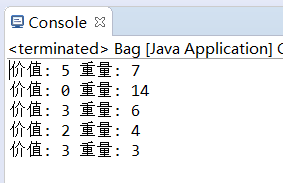
#### 2.3.4.2

物品体积和价值：

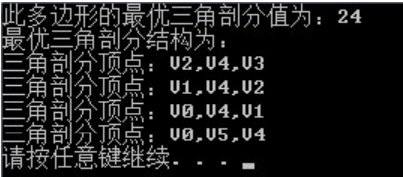
背包容量w：



最后背包情况（价值为0代表不装入）



#### 2.3.4.3

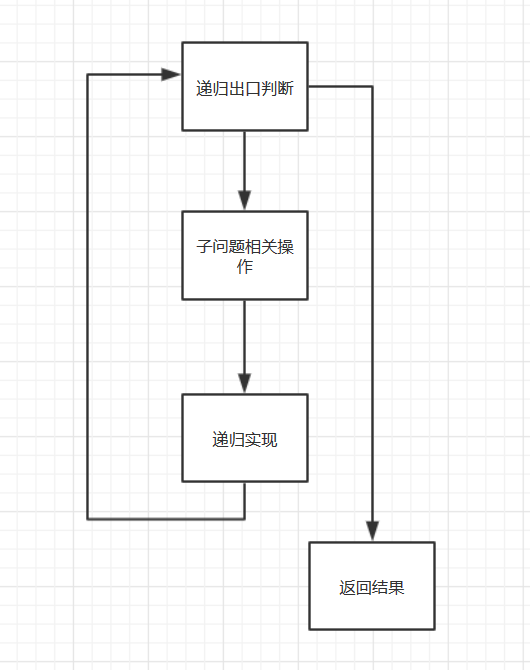


# 3、实验总结

对于动态规划问题，要先理清思路，搞明白程序的出口条件，才可以写出大体的框架，接下来补充上每个子问题该做的事，大概就能把问题写明白。

有时候递归程序有许多变量，利用好这些变量，理解变量之间的关系，甚至利用它们添加新的程序出口，可以节省大量工作。

对同一种情况，递归和迭代两者的思路可能差别很大，迭代一般需要将数据保存下来，这一般需要占用堆的内存空间来保存数据，虽然在空间上做出了一定的牺牲，但程序在整体速度和安全性上可能优于递归，而递归则省下了大量，递归过程中在栈中使用的空间编译器会在递归出口处销毁，但如果数据过多，就会大大加大进行递归的次数，这时可能导致栈溢出。



（递归流程整体思路总结）