

**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** | **148** | **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：windows10操作系统

2.编译环境：CodeBlock（使用C++11）

## 1.3 实验要求

**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：3

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

输出的标准答案：3

1 2 3

4

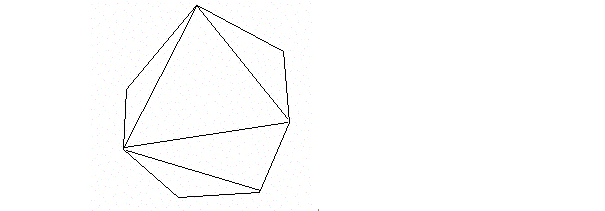
1 3 5 6

10

1 2 3 5 10 12 16 23 31 47

**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

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

输出的标准答案：

5.606

4928.722

**3.0/1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

57 3 2 3

8100

7 13 45 25 16 75 48 32

6 5 20 10 7 32 22 13

输出的标准答案：

13

1 0 1 1 1

48

1 0 1 0 0 0 1 0

# 2. 实验记录

## 2.1 动态规划的介绍

### 2.1.1动态规划概念及理解

动态规划(通常我简写为DP)是运筹学的一个分支，是求解决策过程最优化的数学方法。动态规划可以通过把原问题分解为相对简单的子问题的方式求解复杂问题的方法。动态规划常常适用于有重叠子问题和最优子结构性质的问题。其中，最优子结构指当问题的最优解包含了其子问题的最优解时，称该问题具有最优子结构性质。重叠子问题指在用递归算法自顶向下解问题时，每次产生的子问题并不总是新问题，有些子问题被反复计算多次。所以动态规划可以有效的节约时间和空间，复杂度比一般的暴力算法要低。

举个例子：假设要你求最大严格递减子序列，按照暴力算法，就是枚举所有子串超出最长的，复杂度O（2^n）。然而，如果用DP来做这道题，有复杂度O（n^2）和O（nlogn）两种算法，这两者完全不是一个级别的，足以看出DP的应用性有多好。我们直接设DP[i]为取到i结尾的最长递减子序列，然后每次维护这个DP[i]就行了。关于这个问题在下面的最长递增子序列详细讲述。

### 2.1.2动态规划的实现方法和步骤

由于我的DP学得不好，所以只能根据大佬的博客和自己的理解对DP 的解决步骤进行总结：

1. 描述最优解的结构，分析最优解性质，看看有没有最优子结构。
2. 递归定义最优解的值，找出状态转移方程。
3. 按自底向上的方式计算最优解的值，由已知推出未知。
4. 由计算出的结果构造一个最优解。

以上四点基本就能解决一些DP问题，尤其是第一第二点，当你确定出状态转移方程后，基本上就已经能够求解出最优解了。从以上四个步骤我们就能给出DP的基本实现方法：先找出最优子结构，列出状态转移方程，然后设出一个合适的DP数组表示最优子结构，然后用代码描写状态转移方程。

## 2.2 实验过程

### 2.2.1最长递增子序列问题的实验过程

1.实验思路：

最长递增子序列可以说是动态规划最基础最简单的题目了，刚接触ACM的时候LIS就是当例题讲的。这道题的最终目的是求解最长递增子序列的长度和位置，比我刚打ACM时只需要求长度升级了一点，但是不用急，还是很简单。要用DP来解决这道题，首先我们要设DP，我们设DP[i]为以当前位置i结尾的最大递增子序列长度，那么就有状态转移方程DP[i] = max(DP[j] + 1, DP[i])且a[i] > a[j]，i> j。显然我们每次解决DP[i]都需要遍历i前的所有数，而i==n，那么这个算法的复杂度是O（n^2），**这里我分析了这个算法的复杂度。**如果第一次接触DP的人可能对这个方程有点懵，那我就稍微详细的解释一下。我们从第一个开始判，判断到最后一个，当判断到第i个时，**我们已经得到了所有以i前面数结尾的递增子序列DP[1]~DP[i -1]**，如果a[i]>a[j]，那么我们就能加到这个j的后面形成新的递增子序列，但我们要求最长，所以在DP[1]~DP[i -1]找一个最大的加在后面。这样一来，我们找到DP数组的最大值就是最长的递增子序列。

但是还有一个问题，怎么找到所有序列中的数呢？回溯。我们用pre储存当前i排在谁的后面，初始时所有人都排在自己后面。我们找到最长递增子序列的末尾然后不断回溯，直到遇到某个i是排在自己后面时，结束回溯，那么就找完了所有的序列上的值。那为什么不用上课时候讲的后继表示，而要用回溯呢？因为这里的DP你找不到第一个是谁，但能用O（n）确定出最末尾的是谁，那直接回溯到最初。

2.程序代码：

**#include<set>**

**#include<map>**

**#include<stack>**

**#include<cmath>**

**#include<queue>**

**#include<vector>**

**#include<string>**

**#include<cstdio>**

**#include<cstring>**

**#include<sstream>**

**#include<iostream>**

**#include<algorithm>**

**typedef long longll;**

**using namespace std;**

**constintmaxn = 500 + 10;**

**constint MOD = 1e9 + 7;**

**constint INF = 0x3f3f3f3f;**

**intdp[maxn], pre[maxn], a[maxn];**

**//dp[i]：以第i个数字结尾的最大递增子序列长度**

**//pre：序列中当前数的前一位（前驱）**

**void dfs(int u){ //递归（回溯）答案**

**if(u == pre[u]){**

**printf("%d ", a[u]);**

**return;**

**}**

**dfs(pre[u]);**

**printf("%d ", a[u]);**

**}**

**int main(){**

**int n, t;**

**scanf("%d", &t);**

**while(t--){**

**scanf("%d", &n);**

**for(inti = 1; i<= n; i++){**

**scanf("%d", &a[i]);**

**pre[i] = i;**

**dp[i] = 1; //初始化**

**}**

**for(inti = 1; i<= n; i++){**

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

**if(a[i] > a[j] &&dp[j]+ 1 >dp[i]){**

**//状态转移**

**dp[i] = dp[j] + 1;**

**pre[i] = j;**

**}**

**}**

**}**

**intans = -1, pos;**

**for(inti = 1; i<= n; i++){**

**if(ans<dp[i]){**

**ans = dp[i];**

**pos = i; //LIS的结尾**

**}**

**}**

**if(t != 0) printf("\n");**

**printf("%d\n", ans);**

**dfs(pos);**

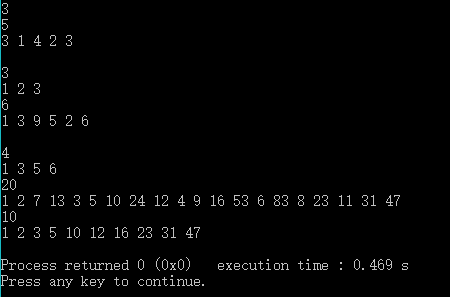
**printf("\n");**

**}**

**return 0;**

**}**

3.实现结果：



实验结果正确无误。

### 2.2.2凸多边形的三角剖分

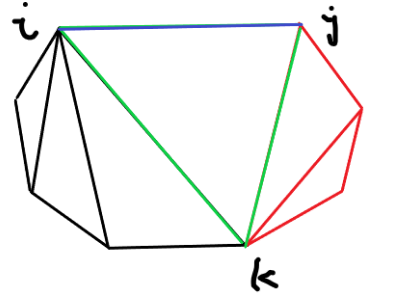
1.实验思路：

这道题有点难，问了一下队里的银牌大佬，告诉我是区间DP。区间DP原理是求解在一个区间上的最优解，那么我把这个区间分割成多个小区间（通常是两个），求解每个小区间的最优解，再合并小区间得到大区间的最优解，讲道理这道题有点难。但是经过我的讲解，就会比较容易了。

现在讲一下具体思路。

我们设DP[i][j]为点i到j这个区间的最优三角剖分，那么显然，如果存在一个点k满足i< k < j，那么三角形ikj也是最优的，否则能找出更优解法显然和DP[i][j]的假设不符。所以我们可以通过不断枚举这个k，使得i~j这个大区间得到最优。**这里我们就找到了最优子结构了。**那么这个问题的最原始状态就是dp[i][i+1] = dp[i][i] = dp[i – 1][i] = 0，即三个点以下不能剖分。那么这个问题的状态转移方程为dp[i][j] = min(dp[i][j],dp[i][k] + dp[k][j] + i到j的距离 + k到j的距离)。

为了防止听不懂，我就画了下面这个图：



假如我们现在合并i`k也就是dp[i][k]（黑色）和k~j也就是dp[k][j]（红色）这两个小区间，显然他们内部各自已经分好了（这由最优子结构可以得到），那么每次我们去合并小区间，只需要加上每个小区间**各自三角剖分的权值**加上**i~k和k~j的距离（绿色）**就能组成新的大区间i~j。为什么不用加上蓝色的i~j？因为i~j最终会形成多边形的边而不是弦，所以我们计算时不用管他。

所以这个DP要先枚举区间长度，再枚举每种长度的起始点的位置和终点的位置，再枚举分成小区间的k的位置，所以时间复杂度O（n^3）,**这里我分析了这个算法的复杂度。**显然区间DP的复杂度很高，所以只适合用在数据量比较小的地方，但是对于直接暴力，复杂度已经小很多很多很多了。

2.程序代码：

**#include<cmath>**

**#include<cstdio>**

**#include<vector>**

**#include<cstring>**

**#include <iostream>**

**#include<algorithm>**

**using namespace std;**

**typedef long longll;**

**constintmaxn = 500 + 10;**

**const double INF = 0x3f3f3f3f;**

**double x[maxn], y[maxn], dp[maxn][maxn];**

**//dp[i][j]表示i到j 区间的最优解**

**double dis(int a, int b){ //ab两点距离**

**if(a + 1 == b) return 0; //相邻不行**

**returnsqrt((x[a] - x[b]) \* (x[a] - x[b]) +**

**(y[a] - y[b]) \* (y[a] - y[b]));**

**}**

**int main(){**

**int t;**

**scanf("%d", &t);**

**while(t--){**

**int n;**

**scanf("%d", &n);**

**for(inti = 1; i<= n; i++){**

**for(int j = 1; j <= n; j++){**

**dp[i][j] = INF; //初始化**

**}**

**}**

**for(inti = 1; i<= n; i++){**

**dp[i][i] = dp[i - 1][i] = dp[i][i + 1] = 0; //不能剖分**

**scanf("%lf%lf", &x[i], &y[i]);**

**}**

**//区间DP**

**for(intlen = 2; len<= n; len++){ //枚举区间长度**

**for(inti = 1; i + len - 1 <= n; i++){ //枚举起始位置**

**int j = i + len - 1; //区间末尾**

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

**//通过子区间状态转移**

**dp[i][j] = min(dp[i][j],**

**dp[i][k] + dp[k][j] + dis(i, k) + dis(k, j));**

**}**

**}**

**}**

**printf("%.3lf\n", dp[1][n]);**

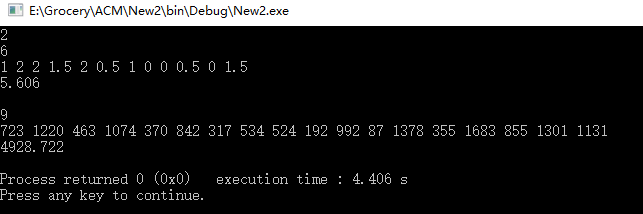
**if(t != 0) puts("");**

**}**

**return 0;**

**}**

3.实现结果：



实验结果正确无误。完美。

### 2.2.30/1背包问题

1.实验思路：

01背包问题也是DP问题的一种基础题型。基础题型中我们通常只需计算最多能放几个，实验报告中的这题新增了要我们输出哪些东西被放进了背包，算是01背包的小变种，其实也很简单，我们只需在状态转移时处理一下即可。这里我先简单介绍一下01背包的基础版，即时间复杂度和空间复杂度均为O（n \* C）的版本。我们设dp[i][j]为选择完第i个物品后容量j的背包的最大价值，也就是对于每个i物品我们可以选择放或者不放。那么得出状态转移方dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - w[i]] + v[i])。dp[i - 1][j]代表不放，dp[i - 1][j - w[i]] + v[i]代表放i物体之后所获得的价值为j - w[i]容量背包的价值加上i的价值，两者取最优。显然，我们需要对i和C进行遍历，**时间复杂度O（n \* C）**。

如果先看我下面的代码可以发现，我遍历j的时候是先从C开始降序遍历的，这个顺序是不能改变的。为什么呢？因为如果我先从小的开始遍历，可能造成背包空间还有剩余但是我塞不下大的物品的情况，所以我们优先塞大的背包，使得所有空间都能遍历。

然后讲一下怎么判断某物品有没有被选。我们可以直接用vector放每个背包选择的物品，如果出现状态转移，我就更新这个vector。

2.程序代码：

**#include<set>**

**#include<map>**

**#include<stack>**

**#include<cmath>**

**#include<queue>**

**#include<vector>**

**#include<string>**

**#include<cstdio>**

**#include<cstring>**

**#include<sstream>**

**#include<iostream>**

**#include<algorithm>**

**typedef long longll;**

**using namespace std;**

**constintmaxn = 5000 + 10;**

**constint MOD = 1e9 + 7;**

**constint INF = 0x3f3f3f3f;**

**int max(int x, int y) {**

**return x > y? x : y;**

**}**

**int w[maxn], v[maxn];**

**//体积，价值**

**intdp[maxn][maxn]; //dp[i][j]为选择完第i个物品后容量j的背包的最大价值**

**vector<int> have[maxn];**

**//have[i]表示容量为i的背包所装的物体编号**

**void change(int from, intpos, int to){//change:把from背包和pos放进to背包**

**have[to].clear();**

**for(inti = 0; i< have[from].size(); i++){**

**have[to].push\_back(have[from][i]);**

**}**

**have[to].push\_back(pos);**

**}**

**int main(){**

**int C, t, n;**

**scanf("%d", &t);**

**while(t--){**

**scanf("%d%d", &n, &C);**

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

**scanf("%d", &w[i]);**

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

**scanf("%d", &v[i]);**

**for(inti = 0; i<= C; i++)**

**have[i].clear(); //背包内容初始化**

**for(inti = 1; i<= n; i++){**

**for(int j = C; j >= 0; j--){**

**dp[i][j] = dp[i - 1][j];**

**if(j >= w[i]){ //放的进背包**

**if(dp[i][j] <dp[i - 1][j - w[i]] + v[i]){**

**dp[i][j] = dp[i - 1][j - w[i]] + v[i];**

**change(j - w[i], i, j);**

**//change:把j - w[i]背包和i放进j背包**

**}**

**}**

**}**

**}**

**if(t != 0) puts("");**

**printf("%d\n", dp[n][C]);**

**for(inti = 1; i<= n; i++){**

**if(i != 1) printf(" ");**

**if(count(have[C].begin(), have[C].end(), i)) printf("1");**

**elseprintf("0");**

**}**

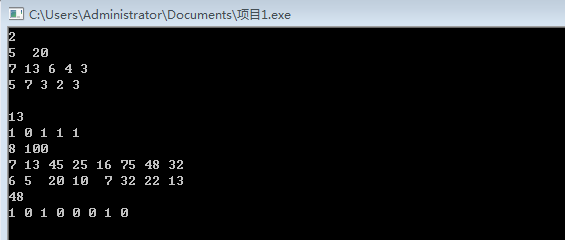
**puts("");**

**}**

**return 0;**

**}**

3.实现结果：



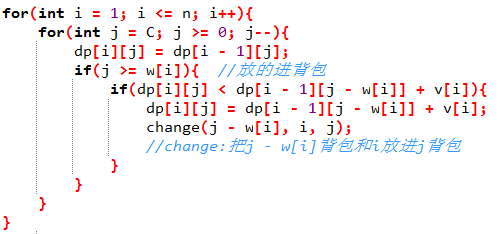
实验结果正确无误。完美。

# 3. 实验总结

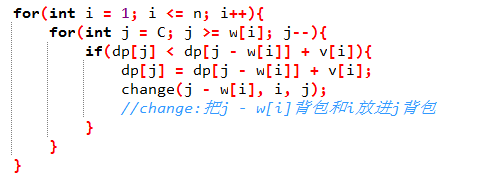
动态规划可以说是算法中比较难的一种类型，因为很抽象，有些题目特别难找到最优子结构，所以动态规划衍生出了很多类型，比如区间DP、概率DP、树形DP来帮助我们更好地掌握DP。在ACM中，对DP的理解不同也能造成写出来的算法复杂度的不同。我通常写DP的步骤是先找最优解的解法，然后设出状态转移方程，然后实现。但是因为抽象思维不是很好，所以写的比较菜。

第一题的LIS还有另一种做法，就是可以用贪心思想和栈模拟优化到O（nlogn）的复杂度，但是和动态规划无关这里就不讲了。第二题难度有点大，最难的地方在于想到状态转移方程的实现。但经过我画的那个图之后我觉得就很好理解了，从那个图可以很容易想到状态转移方程（如果觉得不容易可以多看看书上那个矩阵链乘法）。第三题的01背包我开了二维数组，因为那样理解起来比较简单，但这里其实可以进行空间优化。

我们仔细观察状态转移时的代码：



我们可以发现什么？假如j >= w[i]时，那么就从dp[i - 1][j]和dp[i– 1][j – w[i]] + v[i]中选一个最大值，否则直接等于dp[i - 1][j]。因为我的i是从1遍历到n的，并且i的dp值只和i– 1有关，也就是说我i的结果只会继承自i - 1，显然我可以直接舍弃放i的这一维数组（或者用一个滚动数组去维护），那么空间复杂度就从O（n \* C）降为O（C），修改为如下代码。



为了防止觉得我偷懒没有实现代码，我将完整代码贴在了PasteMe：<https://pasteme.cn/5871>，因为代码直接贴在word里比较丑。最终结果和预期完全相同，感兴趣可以看一看。

动态规划确实很难，除了要掌握对的方法，还要有勤于思考的脑袋，否则很难找到最优子结构。所以我在每一题的实验思路这一栏写了很多很多很多很多的过程，就是希望没有接触过的人通过我的实验报告也能大体了解我的解题过程和思考过程。