

**数学与信息学院学生实验报告**

**实验课程名称：** 动态规划算法设计与应用 **教师： \_\_**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **实验项目名称** | **实验二 递归算法设计与应用** | | | **实验成绩** |  |
| **学生姓名** |  | **学 号** | **132** | **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。

# 2. 实验记录

## 2.1 最长递增子序列

### 2.1.1 递推公式

Li = max(0 < j < i)Lj + 1 (Si > Sj) Si为元素

L0 = 0

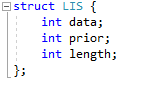
### 2.1.2 算法设想

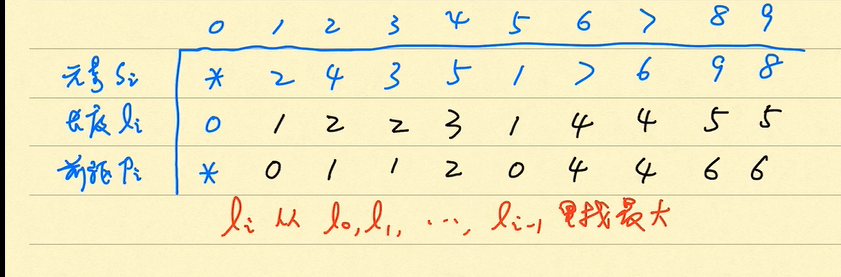
数组从1开始装数据，a[0]装一个负无穷

然后再计算每个元素前面有多少个小于它的数，再记录前驱元素位置

这里用一个结构体，结构体中设置一个prior用来记录前驱元素在该数组

中是什么位置





（上图截取自教学视频）

理论上来说这个算法的复杂度应该是O（n²）,因为要把原序列中的所有元素遍历一遍然后再往一个记录表中插入数据，比较出最长递增子序列，所以要O（n²）

### 2.1.3 实验代码

#include <stdio.h>

#include <malloc.h>

struct LIS {

int data;

int prior;

int length;

};

LIS \* function(LIS \* lis, int position) {

//边界条件，如果这个时候的数字是第一个数字，则将其返回

if (position == 1) {

(lis + position)->prior = 0;

(lis + position)->length = 1;

return lis;

}

//如果不是第一个数字，则向下递归

lis = function(lis, position - 1);

//下面的for循环是为了找出当前position所指向的数字的最长子序列

for (int j = 1; j < position; j++) {

if ((lis + j)->data < (lis + position)->data) {

if (((lis + j)->length + 1) > (lis + position)->length) {

(lis + position)->length = (lis + j)->length + 1;

(lis + position)->prior = j;

}

}

}

return lis;

}

int main()

{

int n = 0;

printf("请输入长度:\n");

scanf\_s("%d", &n);

if (n > 0) {

//如果n大于零，则创造一个LIS型数组,其长度为n + 1

LIS \* lis = (LIS \*)malloc((n + 1) \* sizeof(LIS)); //这里注意，只有结构体指针能够接受这个强制类型转换

//lis->data = -10000; //这里将数组第一个元素赋值为一个很小的数

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

scanf\_s("%d", &(lis + i)->data); //给每个元素赋值

(lis + i)->length = 1;

(lis + i)->prior = 0;

}

lis = function(lis, n);

int maxLength = 1;

int maxPosition = 1;

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

if ((lis + i)->length > maxLength) {

maxLength = (lis + i)->length;

maxPosition = i;

}

}

int \*result = (int \*)malloc(sizeof(int) \* maxLength);

int i = maxPosition;

int j = maxLength - 1;

while ((lis + i)->prior != 0) {

\*(result + j) = (lis + i)->data;

i = (lis + i)->prior;

j--;

}

if ((lis + i)->prior == 0) {

\*result = (lis + i)->data;

}

printf("%d\n", maxLength);

for (int i = 0; i < maxLength; i++) {

printf("%d ", \*(result + i));

}

}

return 0;

}

## 2.1.4 实现结果

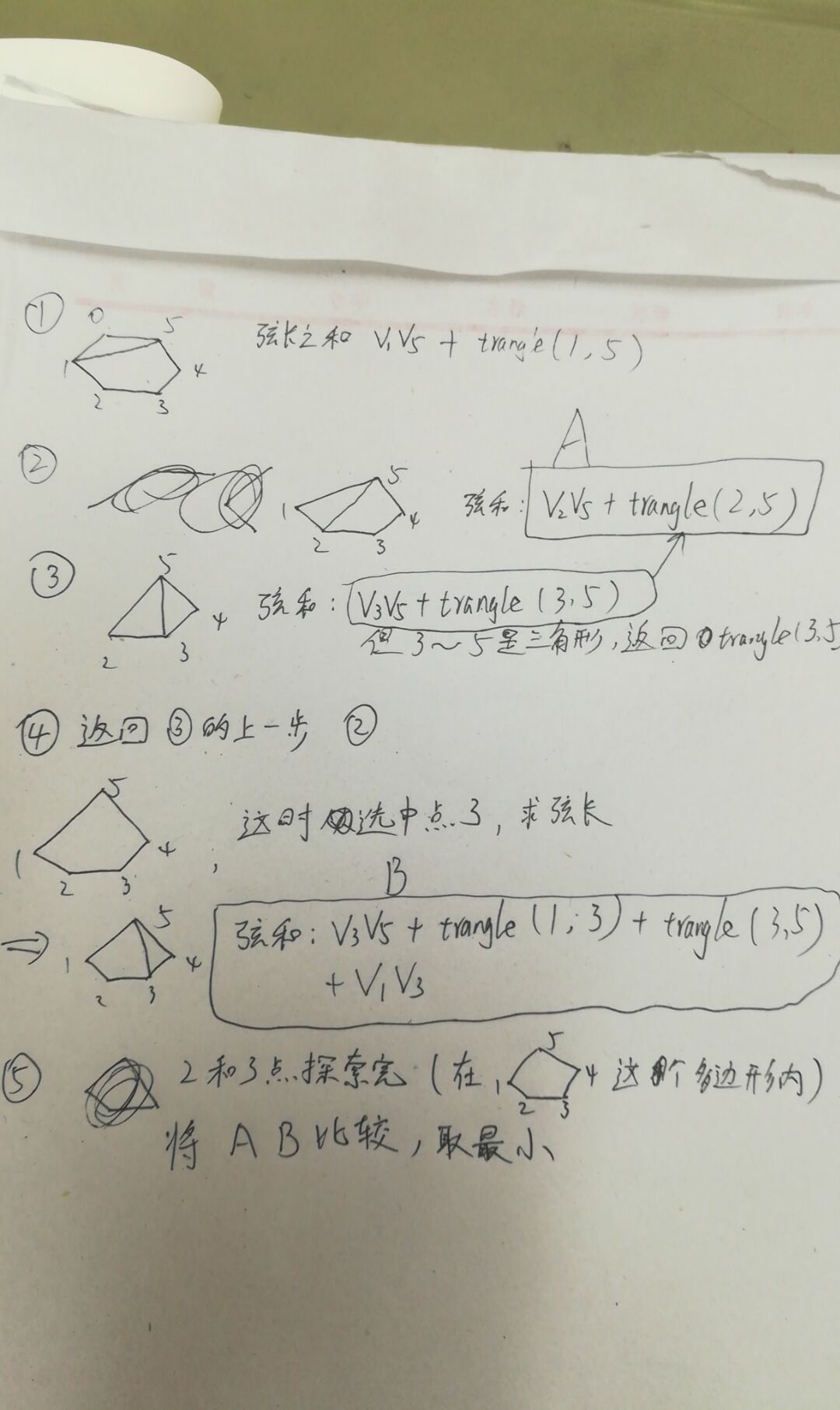


输入序列：2，4，3，5，1，7，6，9，8

返回结果：2，4，5，7，9

# 1 凸多边形的三角剖分

# 3.1.1算法形成过程：



一开始以一个六边形为例子，要在其中找出一个三角剖分该怎么做？

1. 在六边形当中选定某一条弦，如上图中的v1v5
2. v1v5将六边形分为一个三角形和一个五边形，那么这个

六边形的某个三角剖分就等于v1v5 + 五边形的三角剖分

1. 在上图的五边形v1v2v3v4v5中又选中弦v2v5,得到五边

形的三角剖分：v2v5 + 四边形的三角剖分

1. 在四边形中又选中一条弦，然后剩下两个三角形，三角形

返回0

1. 由于四边形中还有一条弦，该四边形的剖分还可以是另一

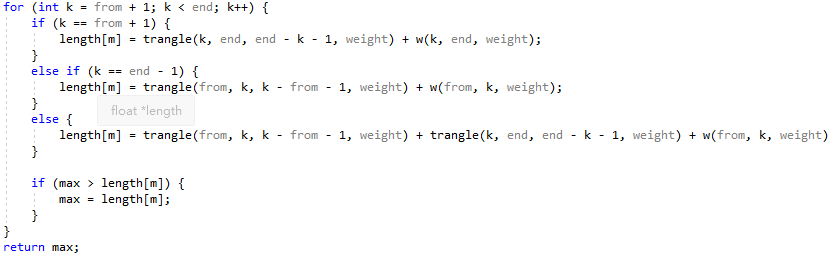
弦加上剩下的两个三角形，所以比较两个剖分哪个更短，将更短的

返回给五边形

1. 同第五步，五边形中找出最短剖分再回到六边形，六边形也

依法炮制，最终得出答案

对于这个算法的复杂度，下面贴出剖分的代码



**3.1.2实验代码**

w函数是返回两点之间的距离的函数。在下面的代码中，多边形每两个点的距离都

在一个二维数组中，比如weight[0][5]表示顶点v0到顶点v5 的距离

#include <stdio.h>

#include <malloc.h>

#include <math.h>

float w(int a, int c, float \*\*weight) {

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

return weight[a][c];

}

float trangle(int from, int end, int step, float \*\*weight) {

int m = 0;

float \*length = (float \*)malloc(step \* sizeof(float));

float max = 10000;

if (from + 2 == end) { //如果最后剖分的就是一个三角形，则没有继续剖分的必要，返回0，结束递归，开始逐层返回剖分结果

return 0;

}

for (int k = from + 1; k < end; k++) {

if (k == from + 1) {

length[m] = trangle(k, end, end - k - 1, weight) + w(k, end, weight);

}

else if (k == end - 1) {

length[m] = trangle(from, k, k - from - 1, weight) + w(from, k, weight);

}

else {

length[m] = trangle(from, k, k - from - 1, weight) + trangle(k, end, end - k - 1, weight) + w(from, k, weight) + w(k, end, weight);

}

if (max > length[m]) {

max = length[m];

}

}

return max;

}

int main()

{

float result = 0;

printf("输入:\n");

int n = 0;

scanf\_s("%d", &n);

float \* arr = (float \*)malloc(2 \* n \* sizeof(float));

for (int i = 0; i < n \* 2; i++) { //输入n个定点的x，y坐标

scanf\_s("%f", arr + i);

}

//接下来申请一个二维数组的地址，这个二维数组用来放置每个顶点到其他顶点的距离权重

float \*\*weight = (float \*\*)malloc(n \* sizeof(float \*)); //动态申请一个一维数组首地址

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

weight[i] = (float \*)malloc(n \* sizeof(float)); //动态申请一维数组的n个元素

}

for (int i = 0; i < n; i++) { //这里计算每个顶点到其它顶点的距离

for (int j = 0; j < n; j++) {

if (j == i) {

weight[i][j] = 0;

}

else if (j != i) {

weight[i][j] = (float)sqrt(pow((arr[i \* 2] - arr[j \* 2]), 2) + pow((arr[i \* 2 + 1] - arr[j \* 2 + 1]), 2));

}

}

}

result = trangle(0, n - 1, n - 2, weight);

printf("最优值:%.3f\n", result);

return 0;

}

**3.1.3实验结果**



# 1 0-1背包问题

**4.1.1实验过程**

参照书本上的伪代码，构造n行C列的数表，得到最优值，其时间复杂度是

O（n\*C）。

至于确定物品是否有被装入背包，则要逆推得到数表的过程

公式 V(i, j) = V(i, j - si) + vi 的含义是，在第i件物品装入背包时，前i-1

件物品装在j-si的背包里。

故判断 if V(i, j) > V(i - 1, j) ，则第i件物品装入背包

then j = j - si

这样就能将数表逆推，找到哪些物品被装入背包

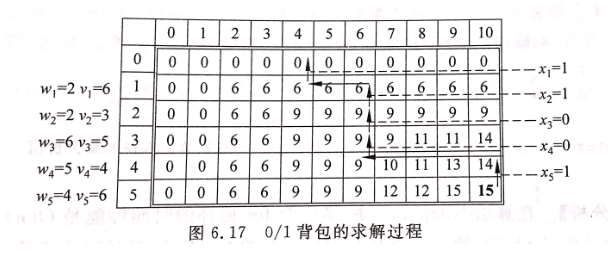
代码实现：

if (result[i][weight] > result[i - 1][weight]) {

state[i - 1] = 1;

weight = weight - goods[i - 1].weight;

}



**4.1.2 实验代码**

**核心算法的代码：**

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

for (int j = 1; j < c + 1; j++) {

result[i][j] = result[i - 1][j];

if (goods[i - 1].weight <= j) {

int temp = 0;

temp = result[i - 1][j - goods[i - 1].weight] + goods[i - 1].value;

if (temp > result[i][j]) {

result[i][j] = temp;

}

}

}

}

全部代码

#include <stdio.h>

#include <malloc.h>

struct good {

int weight;

int value;

};

int Knapsack(good \* goods, int n, int c, int \* state) {

int \*\* result = (int \*\*)malloc(sizeof(int) \* (n + 1));

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

result[i] = (int \*)malloc(sizeof(int) \* (c + 1));

}

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

result[i][0] = 0;

}

for (int j = 0; j < c + 1; j++) {

result[0][j] = 0;

}

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

for (int j = 1; j < c + 1; j++) {

result[i][j] = result[i - 1][j];

if (goods[i - 1].weight <= j) {

int temp = 0;

temp = result[i - 1][j - goods[i - 1].weight] + goods[i - 1].value;

if (temp > result[i][j]) {

result[i][j] = temp;

}

}

//printf("%d, %d, 的结果%d\n", i, j, result[i][j]);

}

}

//printf("最后结果为%d", result[n][c]);

/\*

下面的函数用来查找是否装入背包

从数表右下角，result[n][c]开始，和result[n-1][c]比较

如果大于n-1，c 就说明第n件物品装入背包中，其它的物品共用剩下的c-goods[n].weight的空间

state[n] = 1

然后用result[n-1][c - goods[n].weight]和result[n-2][c - goods[n].weight]比较

如果大于，说明第n-1件物品也装入了，再做同样的操作

state[n-1] = 1

\*/

int weight = c;

for (int i = n; i > 0; i--) {

if (result[i][weight] > result[i - 1][weight]) {

state[i - 1] = 1;

weight = weight - goods[i - 1].weight;

}

else {

state[i - 1] = 0;

}

}

return result[n][c];

}

int main() {

int turn = 0;

printf("输入：");

scanf\_s("%d", &turn);

while (turn > 0) {

int n, c;

scanf\_s("%d %d", &n, &c); //这里输入物品个数n 和 背包容量c

good \* goods = (good \*)malloc(sizeof(good) \* n); //动态分配n个good结构体

int \* state = (int \*)malloc(sizeof(int) \* n);

for (int i = 0; i < n; i++) { //输入每个物品的重量

scanf\_s("%d", &goods[i].weight);

}

for (int i = 0; i < n; i++) { //输入每个物品的价值

scanf\_s("%d", &goods[i].value);

}

int answer = 0;

answer = Knapsack(goods, n, c, state);

printf("最大价值:%d\n", answer);

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

printf("%d ", state[i]);

}

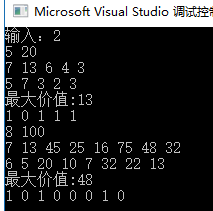
printf("\n");

turn--;

}

return 0;

}



# 实验总结

本次学习动态规划的过程非常不顺畅。

最长递增子序列，这个实验一开始做的时候困难重重。虽然能够理解题目的要求，但是在自己举例的过程中，始终没有很好的方法去准确找出究竟哪个递增子序列是最长的。刚开始的想法也很单纯，计划从第一个元素开始，遍历和比较其它元素，试图直接列出所有递增子序列，但是在实际的操作过程中却发现总是不能把每种情况都判断清楚。

后来在别人的 教学视频中学到可以不要从每个元素开始，而是要以每个元素为结尾，逆向寻找子序列，把问题分解为子问题，等等等等。

同样的，第二题的三角剖分也花了非常多的时间，从着手开始做，到真正完成，断断续续做了有七八天。。。。。

在这个过程中，不断地查找资料，视频，但很少有符合实验需求的思维方法。在不断失败的过程中，总算开始逐渐掌握动态规划的核心。

虽然在课堂上听得很清楚，将大问题分解为同类型的子问题，再存储某些解的答案，每个子问题得到最优，到最后回到大问题，得出最后的答案。但是，这个过程的实现确实需要一定的经验。

只有在不断的尝试下，才能够理解动态规划的思维。