# 2015级算法第三次上机解题报告（助教版）

## 目 录

一、引言 ........................................ 1

二、解题报告 ........................................ 1

A Nintendo Switch 生产车间 ........................................ 1

B I have a tree ........................................ 3

C Magry 的朋友很多 – 零食篇 ........................................ 5

D Longest Common Subsequence ........................................ 7

E 身可死，武士之名不可弃 ........................................ 9

F Magry 猎奇的省钱策略 ........................................ 12

# 2015 级算法第三次上机解题报告（助教版）

马国瑞

## 一、引言

较前两次上机相比，本次上机难度偏大，上机 Board 原始平均分 2.54，最高分 4.2，E、F 题有效提交截止时间延长到 23：00 并对 F 题分数做出处理后平均分 2.70，最高分 5.6.

本次上机重点考查动态规划的知识点，上机期间动态规划部分老师只讲了一半，加之动态规划本身也是上机题的一大难点，因此本次上机难度和前两次相比大不少。

本篇助教版解题报告中，大家上交的解题报告的内容比重和上一篇相比比重更大。在此向所有提交解题报告的同学们的学习积极性提出表扬与感谢。

此外，欢迎大家对这份解题报告内容中出现的问题批评指正。

## 二、解题报告

### A Nintendo Switch 生产车间

#### 思路分析

本题和第二次上机 B 题解题方法如出一辙，同样对 m 道工序贪心地分别得出最小值然后求和即可，唯一不同的是这次是对 m 个数组的最小值进行相加。伪代码如下：

```
Ans=0;
for i=1 to m
    Min[i]=a[i][0];
    for j=1 to n
        if(a[i][j]<Min[i]) Min[i]=a[i][j];
    Ans+=Min[i];
return Ans;
```

特别地，本题在求最小值赋初值的时候需要设为数组的第一个元素或 int 范围的最大值，否则当所有的数都是 INT_MAX 的时候得到的结果并不正确。

对于题末提到的思考题，是老师上课讲动态规划的第一个例子，在此不再赘述。

#### 参考代码

```
#include<cstdio>
#include<cstring>
```

```
long long buf[1007];
long long a[1007][1007];
int main()
{
    int T;
    scanf("%d",&T);
    for(int cnt=1;cnt<=T;cnt++){
        int n,m;
        scanf("%d%d",&n,&m);
        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++){
                scanf("%lld",&a[j][i]);
                if(i==0) buf[j]=a[j][0];   //赋初值操作
            }
        }
        //求最小值操作
        for(int j=0;j<m;j++){
            for(int i=1;i<n;i++){
                if(buf[j]>a[j][i]) buf[j]=a[j][i];
            }
        }
        //相加操作
        long long ans=0;
        for(int j=0;j<m;j++) ans+=buf[j];
        printf("Case #%d: %lld\n",cnt,ans);
    }
}
```

## B I have a tree

### 思路分析

**方法一** 自顶向下法

很简单的，我们能够不花费多少时间得到这个结果——

ans = a[1]+$ans_{左子树} + ans_{右子树}$

同样代码编写也很简单

```
int solve（int i，int j) {
   if(i==n)
      return a[i][j];
   return a[i][j] + max( solve(i+1,j), solve(i+1,j+1) );
}
```

但是这样做效率是极其低的，当树的层数增多，重复计算将增大到一个不能忍受的级别！这是因为发生了重复调用，当递归层数增多时重复调用会增加的更多，耗费的时间指数增长。

不过，我们可以通过以空间换时间、记忆化搜索的方式来解决这个问题。

```
int solve（int i，int j) {
   if(d[i][j]>0)
      return d[i][j];
   if(i==n)
      d[i][j] = a[i][j];
   else d[i][j] = a[i][j] + max( solve(i+1,j), solve(i+1,j+1) );
   return d[i][j];
}
```

**方法二** 自底向上法

```
for(int i = 1; i <= n; i++)
   d[n][i] = a[n][i];
for(int i = n-1; i >= 1; i--)
   for(int j = 1; j <= i; j++)
      d[i][j] = a[i][j] + max(d[i+1][j],d[i+1][j+1]);
```

由于 i 是逆序枚举，所以在计算 d[i][j]之前，d[i+1][j]和 d[i+1][j+1]已经提前计算完成了。不会发生重复计算。

核心状态转移方程：d[i][j] = a[i][j] + max(d[i+1][j],d[i+1][j+1]);

## 算法分析

使用动态规划法解决这个问题的时间复杂度为 $O(n^2)$.

## 参考代码

略

## C Magry 的朋友很多 – 零食篇

### 思路分析

Author: 15211115 陈瀚清

本题为 01 背包的改编问题，只比 01 背包问题多了一些判断条件，基本思路不变，具体可以参见《背包九讲》。

同样我们考虑最优子结构，假设手里的钱数为 j，先不考虑喜不喜欢和好吃程度为负这两个特殊条件，那么要使好吃程度最高，对于每个商品（假设价格为 w）而言，要么是不买这个商品就使钱数为 j 时好吃程度最高，要么是先用 j-w 买能买到的好吃程度最高的商品（子问题）再买这个商品使钱数为 j 时好吃程度最高；动态转移方程 d[i][j]=min(d[i-1][j],d[i-1][j-w])，i 为考虑的商品序号，j 为手里的钱数。

当然，本问题可以减小时间复杂度，只使用一维数组就可以解决：商品的序号递增至 j 时，数组里存下来的就是只考虑了 i-1 个商品的最优解，因此我们无需使用二维数组，减小了空间复杂度至 O（n）。

本题的特殊条件为好吃程度为负或不喜欢时，不考虑此商品，只需在循环商品序号时，先判断此商品是否符合条件即可排除。

### 算法分析

时间复杂度为 O(nk)，空间复杂度 O(n)。

### 参考代码

```
//Source: 15211051 王子烈
#include <cstdio>
#include <cstring>

int weight[10007];
int value[10007];
int canpick[10007];
long long mxvalue[100007];

int main() {
    int n;
    while(scanf("%d", &n) > 0) {
        for(int i=1;i<=n;i++) {
            scanf("%d%d%d", &weight[i],&value[i],&canpick[i]);
            if(!canpick[i] || value[i] <= 0) {
                n--;
                i--;
```

```
        }
    }
    memset(mxvalue,0,sizeof(mxvalue));
    int k;
    scanf("%d", &k);
    long long pick;
    for(int i=1;i<=n;i++) {
        for(int j=k;j>=weight[i];j--) { //这里使用了刷表的办法（每次表中代表的值都是当前 i 物品之前的状况），小于 weight[i]的子问题一定是不选第 i 个物品的，所以不会被更新。
            pick = mxvalue[j-weight[i]] + value[i]; //选了的情况
            if(mxvalue[j] < pick) {
                mxvalue[j] = pick;
            }

        }
    }
    printf("%lld\n", mxvalue[k]);
  }
}
```

## D Longest Common Subsequence

### 思路分析

Author: 15211088 王意如

如果是求最公共长子序列的长度，或是求一个最长公共子序列，做法非常简单，用 dp[i][j]表示匹配到 a[i]和 b[j]时最长子序列的长度，有:

```
dp[i][j] = Max(dp[i - 1][j],dp[i][j - 1],dp[i-1][j-1]|a[j]==b[j]);
```

DP 过程完成后，dp[len1][len2]即为最长公共子序列的长度。

要求出所有最长公共子序列，可以对得到的 dp 数组进行溯源，反向找到由 dp[0][0]通向 dp[len1][len2]的路径，过程中每个值发生变化的点就是子序列的一个元素。事实上，在最坏情况下，每个结点都会引出两条路径，复杂度非常高，为 $O(2^n)$.

在宫洁卿的论文《利用矩阵搜索求所有最长公共子序列的算法》中，提出了一个（能将传统算法的指数级复杂度降低到 max{O(mn),O(ck)},k 为最大公共子序列的个数）的算法，使用了双栈来存储部分匹配的串的内容，但是实现较为复杂，不适合在分秒必争的上机环境内使用。

### 算法分析

对于动态规划法求解最长公共子序列的长度，时间复杂度大致为 O(len1*len2)，空间复杂度 O(len1*len2).

### 参考代码

```
//Source: 15211088 王意如
#include <iostream>
#include <cstring>
#include <set>
#include <algorithm>

using namespace std;

const int MaxN = 100 + 7;
int len1, len2, len;
int dp[MaxN][MaxN];
set<string> ans;
char s1[MaxN] = " ", s2[MaxN] = " ";

#define Max(a,b) (((a)>(b))?(a):(b))
//DP 求 LCS 长度
void LCS(char A[],char B[]) {
```

```
    memset(dp, 0, sizeof(dp));
    len1 = (int)strlen(A), len2 = (int)strlen(B);
    for (int i = 1; i <= len1; i++)
        for (int j = 1; j <= len2; j++) {
            dp[i][j] = Max(dp[i - 1][j],dp[i][j - 1]);
            if (A[i] == B[j]) dp[i][j] = Max(dp[i][j],dp[i - 1][j - 1] + 1);
        }
    len = dp[len1][len2];
}
//类似 DFS 求出所有串
void go(int i, int j, string s) {
    if (i <= 0 || j <= 0) return;
    if (s1[i] == s2[j]) {
        s.push_back(s1[i]);
        if (s.length() == len) reverse(s.begin(), s.end()), ans.insert(s);
        else go(i - 1, j - 1, s);
    } else {
        if (dp[i - 1][j] >= dp[i][j - 1]) go(i - 1, j, s);
        if (dp[i][j - 1] >= dp[i - 1][j]) go(i, j - 1, s);
    }
}

int main() {
    while (cin >> s1 + 1 >> s2 + 1) {
        LCS(s1, s2);
        ans.clear();
        go(len1, len2, "");
        for (auto s:ans)
            cout<<s<<"\n";
    }
}
```

## E 身可死，武士之名不可弃

### 思路分析

Author: 14211079 王媛媛

这道题考察的是动态规划中的双调欧几里得旅行商问题（算法导论 231 页的思考题 15-3 中有提出）。

**求解过程：**

（1）首先将各点按照 x 坐标从小到大排列，时间复杂度为 O(nlgn)。

（2）寻找子结构：定义从 Pi 到 Pj 的路径为：从 Pi 开始，从右到左一直到 P1，然后从左到右一直到 Pj。在这个路径上，会经过 P1 到 Pmax(i,j)之间的所有点且只经过一次。

在定义 d(i,j)为满足这一条件的最短路径。我们只考虑 i>=j 的情况。

同时，定义 dist(i,j)为点 Pi 到 Pj 之间的直线距离。

（3）最优解：我们需要求的是 d(n,n)。

关于子问题 d(i,j)的求解，分三种情况：

A、当 j < i - 1 时，d(i,j) = d(i-1,j) + dist(i - 1,i)。

由定义可知，点 Pi-1 一定在路径 Pi-Pj 上，而且又由于 j<i-1,因此 Pi 的左边的相邻点一定是 Pi-1.因此可以得出上述等式。

B、当 j = i - 1 时，与 Pi 左相邻的那个点可能是 P1 到 Pi-1 总的任何一个。因此需要递归求出最小的那个路径：

`d(i,j) = d(i,i-1) = min{d(k,j) + dist(i,k)},其中 1 <= k <= j`

C、当 j=i 时，路径上最后相连的两个点可能是 P1-Pi、P2-Pi...Pi-1-Pi。

因此有：

`d(i,i) = min{d(i,1)+dist(1,i),...,d(i,i-1),dist(i-1,i)}`

### 算法分析

算法的时间复杂度为 $O(n^2)$，空间复杂度 $O(n^2)$.

### 参考代码

```
//Source: 15211088 王意如
#include <iostream>
#include <cstring>
```

```
#include <algorithm>
#include <iomanip>

using namespace std;

#define x first
#define y second
#define sqr(a) ((a)*(a))
#define Min(a, b) (((a)<(b))?(a):(b))
typedef long long ll;
typedef pair<ll, ll> pll;   //存点

const int MaxN = 107;
int n;
pll p[MaxN];
double dist[MaxN][MaxN];
double dp[MaxN][MaxN];

//两点间距离
inline double Dist(pll &a, pll &b) {
   return sqrt(sqr(a.x - b.x) + sqr(a.y - b.y));
}

int main() {
   ios::sync_with_stdio(false);
   cin.tie(0);
   cout.tie(0);
   while (cin >> n) {
      //读入，并按照 x 为第一关键字排序
      for (int i = 0; i < n; i++)
         cin >> p[i].x >> p[i].y;
      sort(p, p + n);

      //预处理出所有距离
      for (int i = 0; i < n; i++)
         for (int j = 0; j < n; j++)
            dist[i][j] = Dist(p[i], p[j]);
      memset(dp, 0, sizeof(dp));
      dp[0][0] = 0;
      dp[1][0] = dist[1][0];
```

```
        //状态转移
        for (int i = 2; i < n; i++) {
            for (int j = 0; j <= i; j++) {
                dp[i][j] = 1e20;
                if (j < i - 1)
                    dp[i][j] = dp[i - 1][j] + dist[i - 1][i];
                if (j == i - 1)
                    for (int u = 0; u < i - 1; u++)
                        dp[i][j] = Min(dp[i][j], dp[i - 1][u] + dist[u][i]);
                if (j == i)
                    for (int u = 0; u < i - 1; u++)
                        dp[i][j]  =  Min(dp[i][j],  dp[i  -  1][u]  +  dist[u][i]  +
dist[i - 1][i]);
            }
        }
        cout << fixed << setprecision(2) << dp[n - 1][n - 1] << "\n";
    }
}
```

# F Magry 猎奇的省钱策略

## 思路分析

Author: 15151165 马宇航

这道题初一看让人摸不到头绪，但仔细分析后还是一道 01 背包题，也有些类似依赖背包。

对一个指定的顺序，操作结果将确定。关键在于哪些商品选择被售货员扫，我们可以把这些商品看成一个个篮子，而售后员专注时间 time[i]可以看成其容量。而其余每件商品可以看为质量为 1 的待装包物体。这里用 Dp[m][n]表示对于前 m 个物体，所有篮子的空间为 n 时助教需要付出的最小代价。

当 n=0 时，Dp[m][n]=0 购买 0 个为 0.

对于第 i 件物品。有两种情况：

- 作为物品装入 Dp[i][j]=Dp[i-1][j]。
- 作为篮子装物品，Dp[i][j]=Dp[i-1][j-time[i]]+value[i]。

选择较小的那种情况装入。

最后在 Dp[n][1]到 Dp[n][n]中找最小值输出即可。

若想使 Dp[n][n]直接为最小值，只需将第二个方程改为 Dp[i][j]=Dp[i-1][j-time[i]-1]+value[i]即可。参见背包九讲。

以上为状态转移方程。故时间的复杂度为 $O(n^2)$，空间复杂度为 $O(n^2)$，经优化空间可以达到 $O(n)$。

## 参考代码

```
#include<iostream>
#include<cstring>
#define INF 0x7FFFFFFFFFFF
using namespace std;
long long dp[4010];
int t[2010];
long long c[2010];
void solve(int n, int limit)
{
    //初始化，解决的问题是需要的花费刚刚好为 i 时的最小花费
    for(int i=0;i<4010;i++)
        dp[i]=INF;
    dp[0]=0;
    for(int i=1;i<=n;i++){      //01 背包模型处理问题
```

```
        for(int j=limit+2000;j>t[i];j--){
            if(j-t[i]<1&&dp[j]>c[i]) dp[j]=c[i];
            else if(dp[j]>dp[j-t[i]-1]+c[i]){
                dp[j]=dp[j-t[i]-1]+c[i];
            }
        }
    }
}
int main()
{
    t[0]=0;
    c[0]=0;
    int n;
    while(cin>>n){
        long long sum=0;
        int buft=0;
        for(int i=1;i<=n;i++){
            cin>>c[i]>>t[i];
        }
        solve(n,n);
        long long ans=dp[n];
        for(int i=n+1;i<=n+2000;i++){
            if(ans>dp[i]) ans=dp[i];
        }
        cout<<ans<<endl;
    }
}
```