目录

全国信息学奥林匹克联赛（NOIP2007）复赛提高组

命题与解题报告

作者：吉林省实验中学 高二(6)班 沙渺
指导教师：吉林省实验中学信息组 曹玉峰

综述

近几年的 NOIP 处在变革之中。题目难度、考察点、考察广度都在不断地起起伏伏。不可否认这给选手的复习备考造成了很大的困难，但是对于后起的信息学奥赛来说，这也许是一个必须经历的过程。

今年的题目相对 2005/2006 年的题目，难度作了很大的降低。这无疑吸引了更多选手参加信息学奥赛，但分数线随之的提高，也给在信息学奥赛上已经有了一定基础的选手提出了新的挑战。

难度的降低，就要求选手不能死钻难题，而要在学习风格上更细致、更精准，培养良好的编程习惯，先夯实基础再尝试拔高。同时，对问题本质的观察力也是十分重要的。

解题报告中所有的时间数据均是在 AMD Sempron(1.5GHz)处理器，256MB 内存环境下测定的。

1. 统计数字（count）

命题

【问题描述】

某次科研调查时得到了 n 个自然数，每个数均不超过 1500000000 ($1.5*10^9$)。已知不相同的数不超过 10000 个，现在需要统计这些自然数各自出现的次数，并按照自然数从达到小的顺序输出统计结果。

【输入】

输入文件 count.in 包含 n+1 行：

第 1 行是整数 n，表示自然数的个数。第 2~n+1 行每行一个自然数。

【输出】

输出文件 count.out 包含 m 行（m 为 n 个自然数中不相同数的个数），按照自然数从大到小的顺序输出。每行输出两个证书，分别是自然数和该数出现的次数，其间用一个空格隔开。

【输入输出样例】

count.in	count.out
8 2 4 2 4 5 100 2 100	2 3 4 2 5 1 100 2

【限制】

40%的数据满足：1<=n<=1000

80%的数据满足：1<=n<=50000

100%的数据满足：1<=n<=200000，每个数均不超过 1 500 000 000（$1.5 * 10^9$）

题解

本题的思路有两种：①所有元素先排序，再从前到后计数输出。②先统计出现次数，再按顺序输出。

"先排序再输出"是几乎所有参赛选手选用的方法。本题的数据规模为 200000，显然要使用 $O(n\log_2 n)$的排序算法。建议使用快速排序或堆排序。快速排序时间效率高，但要注意容易超时的特殊情况，比如所有元素全相同、升序、降序等，如果考虑不周，时间效率有退化到 $O(n^2)$的风险。堆排序虽然交换次数多，时间花费大（实测较快速排序慢 50~150ms），但由于时间复杂度最坏情况也不会超过 $O(n\log_2 n)$，所以不需要对极限数据做特别的优化，编码容易。从比赛策略的角度，是最值得推荐的方法。

"先计数再输出"的思路中，本题"已知不相同的数不超过 10000 个"的条件，说明空间复杂度的 n 最多 10000，内存使用上可以接受。但即使如此，数据本身也实在太大，达到了 $1.5*10^9$，需要一种良好的数据结构进

行寻址、存储。哈希表、二叉查找树都是比较好的办法。处理一遍数据，哈希表能做到 O(n)，二叉查找树也能 $O(n\log_2 n)$。

总结

本题是一道能让选手发挥出自己水平，展现编程个性的简单题。对于简单题，在比赛策略上就要求稳，要对各种算法的优缺点有基本的了解。即使数据不考察，也要考虑各种影响算法效率的特殊情况，选择最容易编码和调试，风险最小的方法。

参考程序(C++)

/* 以堆排序作为实现方式。*data 指向一块 sizeof(int) * n 的动态内存，作为数组使用。*/

```
#include <cstdlib>
#include <iostream>
#include <fstream>
using namespace std;
void heap_shiftdown(int L[], int len, int id);
int main()
{
  int n, i, *data;
  ifstream fin("count.in");
  fin >> n;
  data = new int[n+1];
  for (i=1; i<=n; i++)
    fin >> data[i];
  fin.close();

  //建小顶堆
  for (i=n/2; i>=1; i--)
    heap_shiftdown(data, n, i);

  //每次抛出一个最小的数据，并计数输出
  ofstream fout("count.out");
  int iVal, iCnt;
  iVal = data[1];
  iCnt = 0;
  for (i=n; i>=1; i--)
  {
    if (data[1] == iVal)
    {
      iCnt++;
    }
    else
    {
```

```
      fout << iVal << " " << iCnt << endl;
      iVal = data[1];
      iCnt = 1;
    }
    swap(data[1], data[i]);
    heap_shiftdown(data, i-1, 1);
  }
  fout << iVal << " " << iCnt << endl;
  fout.close();

  delete []data;
  data = NULL;
  return 0;
}

void heap_shiftdown(int L[], int len, int id)
{
  int minval, minid;
  while (id*2 <= len)
  {
    //先设最小的子结点是自身
    minval = L[id];
    minid = id;
    if (L[id*2] < minval) //如果左子结点更小
    {
      minval = L[id*2];
      minid = id*2;
    }
    if (id*2+1 <= len) //如果存在右子结点
    {
      if (L[id*2+1] < minval) //如果右子结点更小
      {
        minval = L[id*2+1];
        minid = id*2+1;
      }
    }
    swap(L[id], L[minid]); //最小元素到位
    if (id != minid) //如果自身下移
    {
      id = minid; //指针到位
    }
    else
    {
      break; //向下整堆完毕
    }
  }
}
```

2． 字符串的展开（expand）

命题

【问题描述】

在初赛普及组的“阅读程序写结果”的问题中，我们曾给出一个字符串展开的例子：如果在输入的字符串中，含有类似于“d-h”或者“4-8”的字串，我们就把它当作一种简写，输出时，用连续递增的字母获数字串替代其中的减号，即，将上面两个子串分别输出为“defgh”和“45678”。在本题中，我们通过增加一些参数的设置，使字符串的展开更为灵活。具体约定如下：

(1) 遇到下面的情况需要做字符串的展开：在输入的字符串中，出现了减号“-”，减号两侧同为小写字母或同为数字，且按照 ASCII 码的顺序，减号右边的字符严格大于左边的字符。

(2) 参数 p1：展开方式。p1=1 时，对于字母子串，填充小写字母；p1=2 时，对于字母子串，填充大写字母。这两种情况下数字子串的填充方式相同。p1=3 时，不论是字母子串还是数字字串，都用与要填充的字母个数相同的星号“*”来填充。

(3) 参数 p2：填充字符的重复个数。p2=k 表示同一个字符要连续填充 k 个。例如，当 p2=3 时，子串“d-h”应扩展为“deeefffgggh”。减号两边的字符不变。

(4) 参数 p3：是否改为逆序：p3=1 表示维持原来顺序，p3=2 表示采用逆序输出，注意这时候仍然不包括减号两端的字符。例如当 p1=1、p2=2、p3=2 时，子串“d-h”应扩展为“dggffeeh”。

(5) 如果减号右边的字符恰好是左边字符的后继，只删除中间的减号，例如：“d-e”应输出为“de”，“ 3-4”应输出为“34”。如果减号右边的字符按照 ASCII 码的顺序小于或等于左边字符，输出时，要保留中间的减号，例如：“d-d”应输出为“d-d”，“ 3-1”应输出为“3-1”。

【输入】

输入文件 expand.in 包括两行：

第 1 行为用空格隔开的 3 个正整数，一次表示参数 p1，p2，p3。

第 2 行为一行字符串，仅由数字、小写字母和减号“-”组成。行首和行

末均无空格。

【输出】

输出文件 expand.out 只有一行，为展开后的字符串。

【输入输出样例 1】

expand.in	expand.out
1 2 1	abcsttuuvvw1234556677889s-4zz
abcs-w1234-9s-4zz	

【输入输出样例 2】

expand.in	expand.out
2 3 2	aCCCBBBd-d
a-d-d	

【输入输出样例 3】

expand.in	expand.out
3 4 2	dijkstra2************6
di-jkstra2-6	

【限制】

40%的数据满足：字符串长度不超过 5

100%的数据满足：1<=p1<=3，1<=p2<=8，1<=p3<=2。字符串长度不超过 100

题解

虽然 NOIP 以前也有和字符串有关的题，但都是从字符串中获取信息，只涉及字符串的存储、判断、输出，而不对字符串本身做修改处理。本题作为 NOIP 的第一道字符串处理题，应该引起选手备考的重视。

本题算法简单。用模拟法，从前到后查找，找到一个“－”时，就进行判断，如果满足填充条件，就将后边的内容后移，在空位中填充字符。循环重复这个过程直到查找到末尾。

一般用字符数组存储字符串。但也可以使用各种编程语言所提供的字符串数据结构，如 C++的<string>标准类，或 PASCAL 的 ansistring。

但在程序实现上尤其要注意，字符串的首尾都有可能是“－”，所以请在访问一个“－”的左右字符时，务必注意下标越界的问题。在 10 个测试数据中，一共有 3 个测试点的首字符为“－”，评测中很多程序在这 3 个测试

点由于下标越界而崩溃。

C++的<string>标准类

对于 C++使用者，本题可以使用 C++ STL 的 string 标准类实现。string 类的成员函数为程序的实现提供了极大的方便。这里对 string 标准类的部分函数作简要介绍。

首先，使用 string 类，需要加#include <string>。

string 类的变量以 string var; 定义。

1. 读入、输出

string 类重载了输入输出流的“<<”和“>>”运算符，可以从<iostream>所定义的所有标准流中取出字符串，也可以写入输出流，或者作为字符串流的缓冲区。

字符串的输入以空格、TAB、回车换行符等作为分隔符。对于本题的输入字符串，由于不存在空格和回车换行符，所以可以直接输入。

示例：

```
ifstream fin("expand.in");
string sExp;
fin >> sExp;
```

2. 操作单个字符：var[]

与使用字符数组的语法相同，可以用 var[i]请求字符串 var 的第 i 个字符。注意下标从 0 起。同时 var[i]是可以读写的。

示例：

```
Var="<string>";
     Var[0]='*'; Var[5]='#'; //此时 Var == "*stri#g>"
Var="I love C++ <string>!";
     cout << Var[0] << Var[7]; //输出：IC
     for (i=12; i<18; i++) cout << Var[i]; //输出：string
```

3. 字符串长度：var.length()

以成员函数 length()获得字符串的长度。这个长度不包含结尾的’\0’。实际上，在操作 string 类时，用户不必（也不能）手动管理字符串结束标志’\0’。这是 string 类相对于字符数组而言的最大优点。

示例：

```
Var="abcdefg";        // 此时 Var.length() == 7
```

```
Var="x";                // 此时 Var.length() == 1
Var="";                 // 此时 Var.length() == 0
```

4. 查找函数：var.find(c, pos)

在原字符串 var 中，从第 pos 字符开始，查找子串 c 第一次出现的位置。

参数 c 是待查找的子字符串。可以是单个字符、字符数组，或者 string 类字符串。pos 是开始位置，默认值 0。

返回值是子串在原字符串中第一次出现的位置。如果查找不到子字符串，则返回-1。

对于本题，用 var.find(k+1, ‘-’);查找下一个“－”的位置。

示例：

```
设 Var="xBCDExFG"
Var.find('B');              //==1
Var.find('x');              //==0
Var.find('x', 1);           //==5
Var.find('x', 5);           //==5
Var.find("x", 6);           //==-1
Var.find("CDE");            //==2
Var.find("Gladius Sha");    //==-1
```

5. 替换函数：var. replace(pos, n, str)

将字符串 var 中，从第 pos 个字符起，由 n 个字符组成的子串，替换成字符串 str。同时值得一提的是，当 n==0 时，可以实现将 str 插入 var 中。

本题中，构造好应该插入的内容 sIns，以 var.replace(k, 1, sIns); 替换掉“－”即可。这个过程中不必考虑 str 的长度问题，replace 函数会自动对右侧的字符进行移位，这是非常方便的。

示例：

```
/* 注：每次替换前重新对 Var 赋值 */
Var="abcde"; Var.replace(2, 0, "****");     //Var == “ab****cde”
Var="abcde"; Var.replace(2, 1, "XYZ");    //Var == “abXYZde”
Var="abcde"; Var.replace(1, 3, "UVWXYZ");   //Var == “aUVWXYZe”
Var="abcde"; Var.replace(0, 5, "N"); //Var == “N”
```

C++ string 标准类的更多用法，可以参考介绍 C++ STL 的有关书籍。

总结

1. 字符串处理题的普遍特点就是“麻烦”。应对这样的题目，需要参赛选手对编程语言非常熟悉，熟练掌握跟踪、调试的技巧。

2. 选手要了解编程语言所提供的库函数，平时的学习中，要尝试事半功倍的办法。引用一句 ACM 的名言：“不要发明已经存在的东西”。

3. 请了解比赛举办方对参赛语言的限制。使用 C/C++时，不要用非标准库函数。今年比赛中，有许多含有 tolower()、toupper()、strrev()等非标准库函数的程序，在评测过程中出现编译错误，被判为 0 分。

同时，对于 C++用户，也请在比赛规则允许范围内使用 C++ STL。例如 <string>、<iostream>、<fstream> 就是 NOIP 所允许的，但 <algorithm>、<stack>、<queue>等可能会受到限制。

参考程序(C++)

```
/* 这个程序使用了 string 标准类实现字符串处理。
程序的核心在于.find()和.replace()成员函数的使用。*/
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
//参数。Method:方法，Repeat:重复，Desc:逆序(Descending)
int MethodPara, RepeatPara, DescPara;
string sExp;

int main()
{
    ifstream fin("expand.in");
    fin >> MethodPara >> RepeatPara >> DescPara;
    fin >> sExp;
    fin.close();

    char cLeft, cRight;
    bool bBothLetter, bBothNumber, bRightGreater; //展开的条件
    string sReplace;
    int iPos = -1;
    char i; //这个循环变量对应的是字符
    int j;
    while (true)
    {
        iPos = sExp.find('-', iPos + 1);
        //请务必对第一个/最后一个字符给予特别注意
        if (iPos == 0 || iPos == sExp.length() - 1)
        {
            continue;
        }
```

```
        else if (iPos != -1)
        {
            cLeft = sExp[iPos - 1];
            cRight = sExp[iPos + 1];
                bBothLetter = (cLeft>='a' && cLeft<='z') &&
(cRight>='a' && cRight<='z');
                bBothNumber = (cLeft>='0' && cLeft<='9') &&
(cRight>='0' && cRight<='9');
            bRightGreater = cRight > cLeft;
                      if ((bBothLetter || bBothNumber) &&
bRightGreater)
            { //都是字母或都是数字，并且右>左时，需要展开
                sReplace = "";
                if (MethodPara == 2 && bBothLetter)
                {
                    cLeft -= char('a' - 'A');
                    cRight -= char('a' - 'A');
                }
                for (i=cLeft+1; i<cRight; i++)
                {
                    for (j=0; j<RepeatPara; j++)
                    {
                        if (MethodPara == 3) //填*, 顺序不论
                        {
                            sReplace += '*';
                        }
                        else if (DescPara == 2) //逆序
                        {
                             sReplace = i + sReplace; //插到前
边
                        }
                        else
                        {
                            sReplace += i; //插到后边
                        }
                    }
                }
                sExp.replace(iPos, 1, sReplace);
            }
        }
        else
        {
            break;
        }
    }

    ofstream fout("expand.out");
    fout << sExp;
```

```
    fout.close();
    return 0;
}
```

3. 矩阵取数游戏（game）

命题

【问题描述】

帅帅经常跟同学玩一个矩阵取数游戏：对于一个给定的 n*m 的矩阵，矩阵中的每个元素 a_{ij} 均为非负整数。游戏规则如下：

1. 每次取数时须从每行各取走一个元素，共 n 个。m 次后取完矩阵所有的元素；

2. 每次取走的各个元素只能是该元素所在行的行首或行尾；

3. 每次取数都有一个得分值，为每行取数的得分之和；每行取数的得分 = 被取走的元素值*2^i，其中 i 表示第 i 次取数（从 1 开始编号）；

4. 游戏结束总得分为 m 次取数得分之和。

帅帅想请你帮忙写一个程序，对于任意矩阵，可以求出取数后的最大得分。

【输入】

输入文件 game.in 包括 n+1 行；

第一行为两个用空格隔开的整数 n 和 m。

第 2~n+1 行为 n*m 矩阵，其中每行有 m 个用单个空格隔开

【输出】

输出文件 game.out 仅包含 1 行，为一个整数，即输入矩阵取数后的最大的分。

【输入输出样例 1】

game.in	game.out
2 3 1 2 3 3 4 2	82

【输入输出样例 1 解释】

第 1 次：第一行取行首元素，第二行取行尾元素，本次的氛围 1*21+2*21=6

第 2 次：两行均取行首元素，本次得分为 2*22+3*22=20

第 3 次：得分为 3*23+4*23=56。总得分为 6+20+56=82

【输入输出样例 2】

game.in	game.out
1 4 4 5 0 5	122

【输入输出样例 3】

game.in	game.out
2 10 96 56 54 46 86 12 23 88 80 43 16 95 18 29 30 53 88 83 64 67	316994

【限制】

60%的数据满足：1<=n, m<=30，答案不超过 10^{16}

100%的数据满足：1<=n, m<=80，0<=a_{ij}<=1000

问题实质

本题属于博弈论中典型的"取数问题"的一种。取数问题一般是 1 人、2 人或多人轮流从一个序列中按一定的规则取数，求必胜/必负局，或者某一方的最高得分等问题。

题目虽然以矩阵的形式给出，但不难发现，矩阵的每一行都是相互独立的。也就是说，**每一行都可以以一局新游戏来看待**，题目要求的解实际上是每一局游戏最高得分的总和。这就是本题的实质。抓住这一点，本题就可以转化为一个"单人首尾取数问题"。

题解

可以看到，取到的数都是非负数，并且每次的得分单纯相加得到最终得分。所以，每一次取数的最优解，总能由前一次取数的最优解得出，符合最优子结构性质。并且，下一步如何取，与先前的得分与取法无关，满足无后效性。这样，可以用动态规划解决。

一般地，对序列[i→j] (i<j)选取一次能得到剩余序列[i+1→j]或[i→j-1]。为方便子问题向更小的规模递归转化，最好以首尾元素标记状态。

设 f[i, j] (1≤i≤j≤m)表示将序列[i→j]全部取完的最高得分。

现在考虑 2 的方次问题。设对于序列[i→j]的下一步决策，是整局游戏中第 k 次选取。则在选取之前，剩余的元素数量是 m-(k-1)；而序列[i→j]中当前一定剩余 j-i+1 个元素。∴m-(k-1)=j-i+1。解得 k=m-j+i。

根据题意，k 是 2 的指数，则对应的首尾元素 a_i, a_j 都要乘 2^{m-j+i}。

所以得到状态转移方程：

$$\underset{1\le i\le j\le m}{f[i,j]}=\begin{cases}(i=j)a_i\times 2^m\\(i<j)\max\begin{Bmatrix}a_i\times 2^{m-j+i}+f[i+1,j],\\a_j\times 2^{m-j+i}+f[i,j-1]\end{Bmatrix}\end{cases}$$

边界条件是 i=j 时，序列[i→j]的最优得分就是取走 $a_i(a_j)$的得分。

处理一行需要 $O(m^2)$，总时间复杂度 $O(nm^2)$。

由于涉及高精度运算的问题，在实现方法上，建议使用从边界条件开始递推的过程。

有关高精度运算

比赛时，部分选手在使用高精度运算时出现计算速度过慢的现象。高精度运算是一种很慢的操作，所以在具体实现上，有一些可供参考的建议：

1. 预估数据范围

由于本题的时间复杂度达到 $O(nm^2)$，增长趋势接近 n^3，所以估计高精度运算能达到的数据规模，对时间的优化是有意义的。

以本题的极限数据考虑，当 n=80，m=80，(任意的)a=1000 时，可得最高得分 $score=m\sum_{i=1}^{n}2^i\cdot a=2.41785\times 10^{30}$，$\lceil \log_{10} score\rceil=31$。所以，本题理论极限数据的结果有 31 个十进制位。所以在高精度运算过程中，按常用的 4 位一存，也只需要开 8 位的数组。从测试数据来看，答案最长的数据(game9.ans)准确地达到了 31 位，与理论估计相符。

2. 定义常量

每次算得分时都连乘 i 次算出 2^i 的值，时间上是很不经济的。所以应该将 2^i 以 const 常量开出表格，使用时直接查表。

3. 避免递归

递归调用中不可避免地会发生重复运算。并且，在递归调用中传递高精度数据也比较麻烦。所以递归不仅浪费运行时间，并且反而不如从边界条件递推容易写。

总结

应对动态规划问题，一定要注意思考、结合经典题型考虑问题。

要抓住问题本质。本题只要看到“每一行都是独立的游戏”这一点，就已经算成功了一半。

参考程序(C++)

/* 本程序以递推实现动态规划。

高精度的实现方法是在每个数据上再开一个[8]的维度存储各个数位，4 位一存，最低 4 位在 0 下标的位置。*/

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;
const int GROUP = 8;
const int LIMIT = 1e4;
const int MN_MAX = 80;
//2^k 乘方表 (0<=k<=80)
int TP[81][8] =
{
    1,   0,   0,   0,   0,   0,   0,   0,
    2,   0,   0,   0,   0,   0,   0,   0,
    4,   0,   0,   0,   0,   0,   0,   0,
    8,   0,   0,   0,   0,   0,   0,   0,
   16,   0,   0,   0,   0,   0,   0,   0,
……
//略去大部分
……
 6544,9367,6572,4903,3145,3022,   0,   0,
 3088,8735,3145,9807,6290,6044,   0,   0,
 6176,7470,6291,9614,2581,2089,   1,   0,
};
int v[MN_MAX];
int w[MN_MAX][ MN_MAX][GROUP];
int sum[GROUP];
int iGameCnt, iNumCnt;

void Mul(int a[], int b, int r[]);
void Plu(int a[], int b[], int r[]);
```

```
bool Greater(int a[], int b[]);

int main()
{
    int i, j, l, r;
    ifstream fin("game.in");
    fin >> iGameCnt >> iNumCnt;
    for (i=0; i<GROUP; i++)
    {
        sum[i] = 0;
    }

    //递推求解
    int iGame, iDiff;
    int iAnotherChoice[GROUP], iPoweredV[GROUP];
    for (iGame=0; iGame<iGameCnt; iGame++)
    {
        for (i=0; i<iNumCnt; i++)
        {
            fin >> v[i];
        }
        for (i=iNumCnt; i>=1; i--)
        {
            l = 0;
            r = iNumCnt - i;
            while (l<iNumCnt && r<iNumCnt)
            {
                if (l == r)
                {
                    Mul(TP[i], v[l], w[l][r]);
                }
                else
                {
                    //取左侧的选择 (暂时赋给w[l][r])
                    Mul(TP[i], v[l], iPoweredV);
                    Plu(w[l+1][r], iPoweredV, w[l][r]);
                    //取右侧的选择 (赋给临时变量iAnotherChoice)
                    Mul(TP[i], v[r], iPoweredV);
                                   Plu(w[l][r-1],  iPoweredV,
iAnotherChoice);
                    //比较两种选择谁更优
                    if (Greater(iAnotherChoice, w[l][r]))
                    {
                        for (j=0; j<GROUP; j++)
                        {
                            w[l][r][j] = iAnotherChoice[j];
                        }
                    }
```

```
                }
                l++; r++;
            }
        }
        Plu(sum, w[0][iNumCnt - 1], sum);
    }
    fin.close();
   ofstream fout("game.out");
   //高精度数据的显示
    int seg;
    seg = 0; //保证找不到任何0时能输出个位的0
    for (i=GROUP-1; i>=0; i--)
    {
        if (sum[i] != 0 || i == 0)
        {
            fout << sum[i]; //高位的一组
            seg = i - 1;
            break;
        }
    }
    for (i=seg; i>=0; i--)
    {
        fout << setfill('0') << setw(4) << sum[i];
    }
    fout.close();
    return 0;
}

void Mul(int a[], int b, int r[])
{
    int i;
    for (i=0; i<GROUP; i++)
    {
        r[i] = a[i] * b;
    }
    for (i=1; i<GROUP; i++)
    {
        r[i] += r[i-1] / LIMIT;
        r[i-1] %= LIMIT;
    }
}

void Plu(int a[], int b[], int r[])
{
    int i;
    for (i=0; i<GROUP; i++)
    {
        r[i] = a[i] + b[i];
```

```
    }
    for (i=1; i<GROUP; i++)
    {
        r[i] += r[i-1] / LIMIT;
        r[i-1] %= LIMIT;
    }
}

bool Greater(int a[], int b[])
{
    int i;
    bool sol = false;
    for (i=GROUP-1; i>=0; i--)
    {
        if (a[i] > b[i])
        {
            sol = true;
            break;
        }
        else if (a[i] < b[i])
        {
            sol = false;
            break;
        }
    }
    return sol;
}
```

4. 树网的核 (core)

命题

【问题描述】

设 T=(V, E, W) 是一个无圈且连通的无向图（也称为无根树），每条边带有正整数的权，我们称 T 为树网（treenetwork），其中 V，E 分别表示结点与边的集合，W 表示各边长度的集合，并设 T 有 n 个结点。

路径：树网中任何两结点 a，b 都存在唯一的一条简单路径，用 d(a, b)表示以 a, b 为端点的路径的长度，它是该路径上各边长度之和。我们称 d(a, b)为 a, b 两结点间的距离。

D(v, P)=min{d(v, u), u 为路径 P 上的结点}。

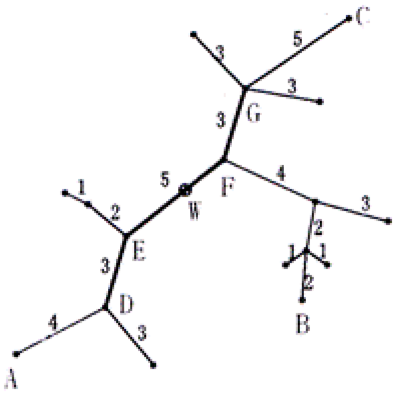
树网的直径：树网中最长的路径成为树网的直径。对于给定的树网 T，直径不一定是唯一的，但可以证明：各直径的中点（不一定恰好是某个结点，可能在某条边的内部）是唯一的，我们称该点为树网的中心。

偏心距 ECC(F)：树网 T 中距路径 F 最远的结点到路径 F 的距离，即

ECC(F)=max{d(v, F)，v∈V}

任务：对于给定的树网 T=(V, E, W)和非负整数 s，求一个路径 F，他是某直径上的一段路径（该路径两端均为树网中的结点），其长度不超过 s（可以等于 s），使偏心距 ECC(F)最小。我们称这个路径为树网 T=(V, E, W)的核（Core）。必要时，F 可以退化为某个结点。一般来说，在上述定义下，核不一定只有一个，但最小偏心距是唯一的。

下面的图给出了树网的一个实例。图中，A-B 与 A-C 是两条直径，长度均为 20。点 W 是树网的中心，EF 边的长度为 5。如果指定 s=11，则树网的核为路径 DEFG（也可以取为路径 DEF），偏心距为 8。如果指定 s=0（或 s=1、s=2），则树网的核为结点 F，偏心距为 12。



【输入】

输入文件 core.in 包含 n 行：

第 1 行，两个正整数 n 和 s，中间用一个空格隔开。其中 n 为树网结点的个数，s 为树网的核的长度的上界。设结点编号以此为 1，2，……，n。

从第 2 行到第 n 行，每行给出 3 个用空格隔开的正整数，依次表示每一条边的两个端点编号和长度。例如，“2 4 7”表示连接结点 2 与 4 的边的长度为 7。

所给的数据都是正确的，不必检验。

【输出】

输出文件 core.out 只有一个非负整数，为指定意义下的最小偏心距。

【输入输出样例 1】

core.in	core.out
5 2 1 2 5 2 3 2 2 4 4 2 5 3	5

【输入输出样例 2】

core.in	core.out
8 6 1 3 2 2 3 2 3 4 6 4 5 3 4 6 4 4 7 2 7 8 3	5

【限制】

40%的数据满足：5<=n<=15

70%的数据满足：5<=n<=80

100%的数据满足：5<=n<=300，0<=s<=1000。边长度为不超过 1000 的正整数

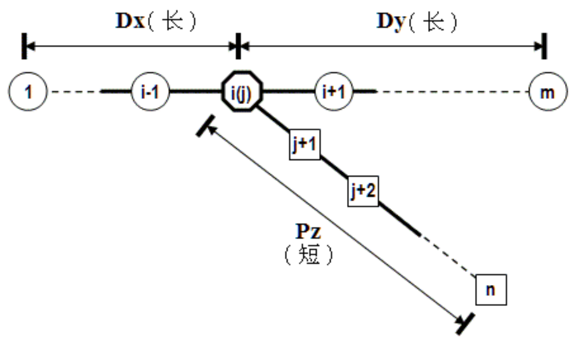
直径的对等性

本题数据规模 300，如果暴力枚举所有的直径与核，时间复杂度是 $O(n^6)$，一定会超时。需要寻找树网中直径的关键性质。

本题所需要的性质如下：

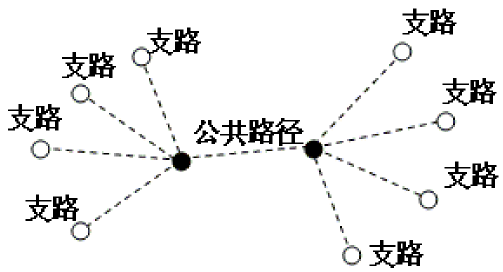
1． 任意两条直径均相交。

2． 如图 1，直径的“支路”不会比直径被其分成的任意一部分长。



（图 1：支路较短性质）

3． 如图 2，所有直径相交于统一的一条路径，公共路径可以退化为一个结点。



（图 2：所有直径相交的形态）

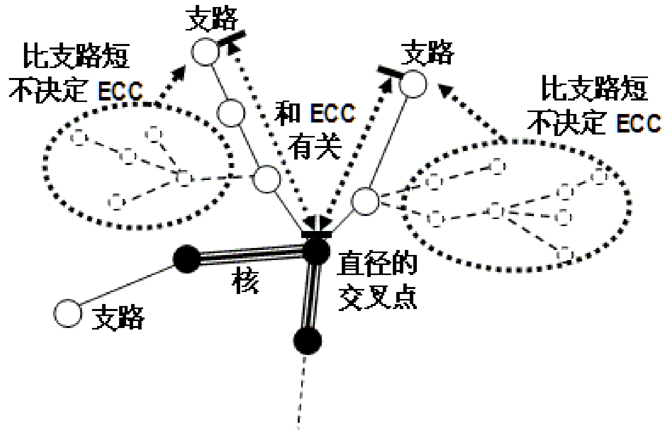
4． 如图 2，所有直径相交时，任一侧伸展开的所有支路等长。

以上的性质都可以用反证法证明，导出违反直径最长的矛盾既可。证明从略。

那么，以图 2 为基础，讨论偏心距的决定因素。

核是简单路径，只能覆盖在 1 条支路上。这样，一定有一条其它支路做了“核外路径”，约束偏心距的下界。而支路上更细微的结构，显然都比支路要短，对偏心距的下界不起约束作用。

图 3 表明了所有直径相交时，各路径对偏心距有无影响的情况。参考图 3，可以知道，所有直径相交时，分叉的任意一侧对偏心距的影响，**只取决于直径在这一侧的支路长度**。



（图 3：与偏心距有无关系的讨论）

所以，最终可以得到结论：求偏心距时，所有的直径在**实质上都是一致的**。无论选取哪一条直径，都能得到唯一正确的解。

所以，直径**不必枚举**，只需**任选一条**。

这是本题最重要的剪枝条件，是以下所有高效算法的基础。

寻找直径

首先显然需要找到任意一条直径。

自然的想法是，先求每对结点间的距离再找直径。这种想法虽然直观，但时间复杂度不理想（Floyd 算法 $O(n^3)$，以每个结点为起点遍历 $O(n^2)$）。并且，只是为了寻找一条直径而求每对结点间距离是不必要的 。

本题可以通过树形动态规划，在 $O(n)$时间内找到一条直径。

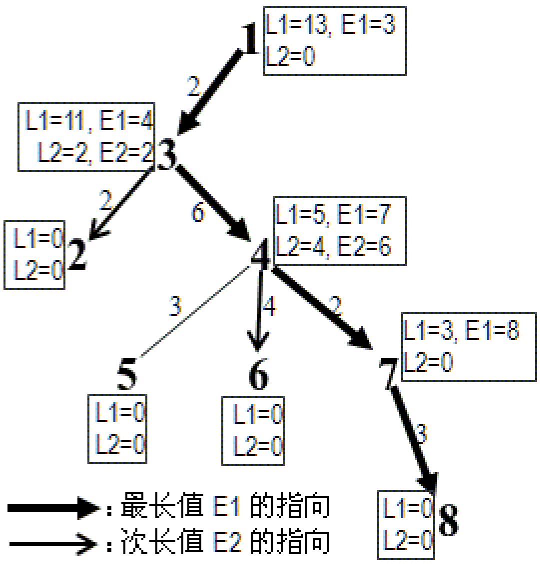
树形动态规划需要将无根树转化为有根树来处理，以任意一个结点（通常取结点 1）为根向下遍历。每一个结点存储以下信息：

(1)从自身向下的最长路径长度 Lmax1 和路径终点 Emax1；

(2)从自身向下的次长路径长度 Lmax2 和路径终点 Emax2。

则对于每一个结点 P 及其下的子树中，包含 P 自身的最长的路径可以表达为：从 Emax1 到 Emax2 的，长度为 Lmax1+Lmax2 的路径。则所有结点的 Lmax1+Lmax2 的最大值就是一条直径。

图 4 是原题样例数据 2，以结点 1 为根的动态规划树。其中，结点 1、3 的 Lmax1+Lmax2 均为最大（=13）它们对应的直径分别为(1, 8)和(2, 8)。



（图 4：树形动态规划求直径）

状态转移方程如下：

Lmax1[P]=max{Lmax1[V] + d(P, V) | V∈P 的子结点}
Lmax2[P]=second_max{Lmax1[V] + d(P, V) | V∈P 的子结点}
如果 P 是叶结点，则 Lmax1[P]=0, Lmax2[P]=0，Emax1[P]=P，Emax2[P]=P。
如果 P 只有 1 个子结点，则 Lmax1[P] 正常求解，Lmax2[P]=0，Emax2[P]=P。
Emax1[P], Emax2[P]指向两个方程中，最大值对应的 V。
Sum[P] = Lmax1[P] + Lmax2[P]，所有 Sum[P]的最大值为直径。

程序实现上，建议从上到下递归实现，求解每一个结点时，遍历求解所有子结点之后再求自身。如果使用递推，需要先预处理，使用栈和队列来确定从下到上的递推顺序，不方便。

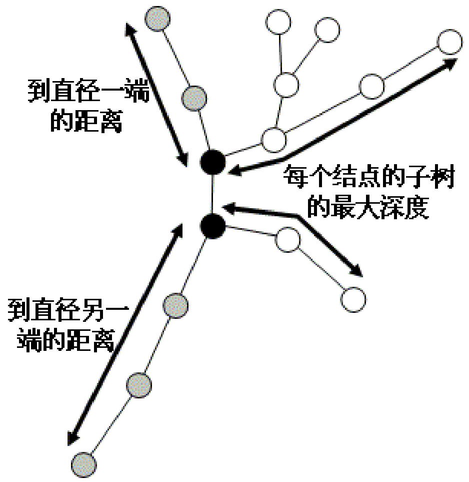
这个动态规划的优化策略，就是尽量选择**结点较少的直径**。这对最后枚举核有利。可以在树型动态规划中，对每个阶段增加“最长/次长路径结点数，Vmax1/Vmax2”来实现。

则状态转移方程中增加：

Vmax1[P]=Vmax1[Emax1[P]]+1
Vmax2[P]=Vmax1[Emax2[P]]+1
如果 P 不存在解 Lmax1/Lmax2，则 Vmax1[P]=1, Lmax2[P]=1。
如果 P 只存在一个解 Lmax1，则 Vmax1[P]正常求解，Vmax2[P]=1。
则对于构成直径的 P，直径结点数 Vdiam[P]= Vmax1[P]+Vmax2[P]-1
对 Sum[P]有最大值的所有点，优先取 Vdiam 较小的。

求解偏心距

偏心距是到一个核的最长距离，是每个结点到核的距离的最大值。



（图 5：偏心距的决定因素。灰色点表示核所在的一条直径，黑色点表示核，白色点表示这条直径外的子结点。）

偏心距的决定因素仅仅是以下两种“关键路径”的长度：

其一是**核的两端到直径对应两端的距离**。

其二是**将直径视为主干时，核上每个结点的子树的最大深度**。

而其它的点和路径对于求解偏心距都没有用，它们一定是“关键路径”的岔路，比“关键路径”要短，对偏心距不起决定影响。

所以，我们将直径的各个结点排成一行，并认为直径的结点区分左右。我们将直径视为**干路**，将直径以外的路径和结点视为**枝叶**。形象的讲，就是把直径横向“展平”，所有的枝叶都以直径的结点为根纵向朝下伸展。

则根据偏心距的决定因素，对每个结点 P，求以下三个值：

(1) P 到直径左端点的距离 L_P；

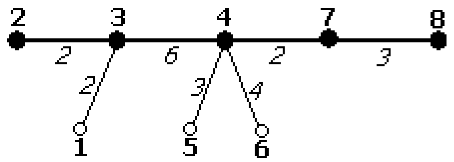
(2) P 到直径右端点的距离 R_P；

(3) 以 P 为根的枝叶子树的最大带权深度 C_P。

则根据偏心距的决定因素，我们可以知道，核 i⇔j（i≤j）的偏心距是 Li，Rj，和 Ci~Cj 的最大值。所以得到 $ECC(i \Leftrightarrow j) = \max\left\{ l_i, r_j, \max\limits_{i \le k \le j}\{c_k\} \right\}$。

图 6 是以样例数据 2 体现的这个过程。

L=0	2	8	10	13
R=13	10	5	3	0
C=0	2	4	0	0



（图 6：求解偏心距。）

所以，只需要枚举所有长度不超过 s 的核，得到最小偏心距就可以了。朴素的枚举时间复杂度可以视为 $O(n^3)$，将每次求 max{Ck}的循环视为一维。

枚举的优化

可以注意到，$O(n^3)$的时间复杂度不理想，有优化的空间。

优化这个枚举算法，需要知道选取核的以下原则：

1. **核越长越好**。更长的核能得到更短的偏心距。
2. **通过中心的核比不通过中心的好**。

   因为不通过中心的核，直径剩余的一侧会成为核外路径，偏心距长度至少是直径一半；而通过中心的核，直径从中点被截断成两部分，所以偏心距长度不会超过直径的一半。

   但原题中 S 太小时，核不得不退化为结点。如果此时中心点在两个结点中间，要注意，假如核**退化为中心点旁边的单个结点**，则这样的核虽然两端点都在中心点同一侧，但也**应该被允许**。

所以，根据这两条性质，可以使用“双指针法”进行枚举。“双指针法”就是在中心点的左右分别设置 L,R 两个指针。初始时 L=中心，R=右端点。L 每次左移 1 个结点（即枚举核的左端点）时，R 指针就相应地不断左移，

直到核(L⇔R)在 S 范围内尽可能最长为止。这个过程直到 L 越过左端，或 R 越过中心点为止。

双指针法的优点不仅在于将遍历的时间降低到 O(n)（因为每一个结点只会被至多请求一次），还在于优化了求解 max{Ci~Cj}的算法。

因为核(i⇔j)一定通过中心点，所以 max{Ci~Cj}一定能以中心点为基准二分。所以，可以预处理，先求出中心点 center 到左侧任意一点 x 的 max{Cx~Ccenter}，和中心点到右侧任意一点 y 的 max{Ccenter~Cy}。这个预处理的时间复杂度 O(n)。而每次查询时，以中心点为基准二分查询 max{max{Ci~Ccenter}, max{Ccenter~Cj}}即可，只需要 O(1)的时间。

所以，双指针法起到了"一箭双雕"的功效，将枚举的总时间复杂度降到了 O(n)。

总结

1. 总结以上的讨论，最后确认本题最优算法的总时间复杂度为 O(n)。相信这是本题的极限。流程如下：(I)树型动态规划寻找直径；(II)预处理求直径有关信息；(III)二分预处理求 max{Ci~Cj}，每次查询 O(1)；(IV)双指针枚举核。

2. 本题的精髓就是"转化转化再转化"。本题条件众多，不可能一步看出问题的核心。而就在一步步转化过程中，问题的表象被一点点砍掉，最终提炼出决定题目答案的本质因素，从而实现了思维上的"剪枝"，可以对问题的"主干"寻找到快速而有效的解法。

本题的表象是各点平等的无根树。而经过一次转化，就变为了直径为主干的有主次的树。再经过转化，枝叶被化为深度值而全部去掉，图形最终转化成了一条直径，问题最终变成了一个线性结构。

"转化"是信息学奥赛中的重要方法，可以说，转化水平高低直接决定了选手能处理多复杂的问题。所以，平时的思维和逻辑能力训练是不可或缺的。引用朱泽园前辈的一句话："思想上的金牌更重要"。

3. 牢牢抓住重要的规律。对于本题，"直径最长"就是一切推导过程和解题策略的中心原则和出发点。

4. 从比赛策略上讲，不妨使用不完全归纳法。

以本题为例。我们可以随机生成一些小数据暴力搜索，发现搜索完第

一条直径后，再搜索其他直径得不到更优的解。所以，可以在 $O(n^6)$暴力搜索的程序中，果断地脱去枚举直径的两层循环，这样，在 300 的数据规模下，就至少可以全过了。

但请小心使用不完全归纳法，并且使用的样本必须足够多，并且要全面，才能确保规律的可靠性。建议只在毫无思路，或朴素算法过慢时，通过这样的办法寻找规律。

最优算法参考程序(C++，O(n))

```
#include <cstdlib>
#include <iostream>
#include <fstream>
using namespace std;
const int N_MAX = 300;
int n, s;
//图的邻接表
void LoadData();
int Gn[N_MAX]; //Gn[n]: 点 i 的邻接点数量
int *Gp[N_MAX]; //Gp[n][Gn[i]]: 点 i 的各个邻接点的编号
int *Gw[N_MAX]; //Gp[n][Gn[i]]: 点 i 到各个邻接点的权值

//图的直径
int dstart, dend; //任意一条直径的两个端点
int Ds; //直径 dstart->dend 长度
int Dn; //直径 dstart->dend 上结点的数量
int Drp; //DP 表格中推得直径的结点
int Dp[N_MAX]; //直径 dstart->dend 上各结点的序号
int Dc[N_MAX]; //以 Dp[i]为根的子树的最大深度
int Dl[N_MAX], Dr[N_MAX]; //直径上结点到左右两端点的距离
bool Df[N_MAX]; //结点 i 是否在路径 dstart->dend 上
int SubTree_Traverse(int Curr, int Depth);


//树型动态规划
void DP_Traverse(int Prev, int Curr);
int Lmax1[N_MAX], Lmax2[N_MAX]; //最长/次长子路径长度
int Emax1[N_MAX], Emax2[N_MAX]; //最长/次长子路径终点
int Vmax1[N_MAX], Vmax2[N_MAX]; //最长/次长子路径结点数
int Nmax1[N_MAX], Nmax2[N_MAX]; //遍历最长/次长子路径的下一个结点

int main()
{
    int i, j;
```

```
LoadData();

//////////////////////////////
//树型DP求直径 (以结点0为根)//
//////////////////////////////
DP_Traverse(0, 0);

////////////
//寻找直径//
////////////
// (Lmax1+Lmax2最长前提下, Vmax1+Vmax2-1最少的结点)
Ds = -1; Dn = 0; dstart = -1; dend = -1; Drp = -1;
for (i=0; i<n; i++)
{
    if ((Lmax1[i] + Lmax2[i] > Ds)
        || (Lmax1[i] + Lmax2[i] == Ds
            && Vmax1[i] + Vmax2[i] - 1 < Dn))
    {
        Drp = i; //DP表格中推得直径的结点
        dstart = Emax1[i]; //直径端点
        dend = Emax2[i]; //直径端点
        Ds = Lmax1[i] + Lmax2[i]; //直径长度
        Dn = Vmax1[i] + Vmax2[i] - 1; //直径结点数
    }
}

//////////////////////
//取得直径的有关信息//
//////////////////////
//直径上各结点编号Dp[Dn] (分左右, 次长路在左, 最长路在右)
int mid = Vmax2[Drp] - 1; //Drp在直径中处于第几个
Dp[mid] = Drp;
int currp;
currp = Nmax2[Drp];
for (i=mid-1; i>=0; i--)
{
    Dp[i] = currp;
    currp = Nmax1[currp];
}
currp = Nmax1[Drp];
for (i=mid+1; i<n; i++)
{
    Dp[i] = currp;
    currp = Nmax1[currp];
}
//整个图中各结点是否在直径上Df[n]
```

```
    for (i=0; i<n; i++)
        Df[i] = false;
    for (i=0; i<Dn; i++)
        Df[Dp[i]] = true;
    //直径上各结点到左端点长度Dl[Dn]
    Dl[mid] = Lmax2[Drp];
    if (mid > 0)
    {
        currp = Nmax2[Drp];
        for (i=mid-1; i>=0; i--)
        {
            Dl[i] = Lmax1[currp];
            currp = Nmax1[currp];
        }
    }
    currp = Drp;
    for (i=mid+1; i<Dn; i++)
    {
                    Dl[i]   =   Dl[i-1]   +   (Lmax1[currp]   -
Lmax1[Nmax1[currp]]);
        currp = Nmax1[currp];
    }
    //直径上各结点到右端点长度Dr[Dn]
    for (i=0; i<Dn; i++)
        Dr[i] = Ds - Dl[i];
    //直径中点两侧的结点Dml, Dmr
    //(如果直径中点落在结点上则Dml==Dmr)
    int Dml, Dmr;
    Dml = 0; Dmr = 1;
    while (!((Dl[Dml] <= Ds / 2) && (Dr[Dmr] <= Ds / 2)))
    {
        Dml++; Dmr++;
    }
    if ((Ds % 2 == 0) && (Dl[Dml] == Ds / 2)) //中点落在结点上
        Dmr = Dml;
    //子树最大深度Dc[Dn]
    for (i=0; i<Dn; i++)
        Dc[i] = SubTree_Traverse(Dp[i], 0);
    //递推求Dc[i~Dml], Dc[Dmr~j]的最大值Dcm[Dn] (不重叠, 存入一个
数组)
    int Dcm[N_MAX];
    Dcm[Dml] = Dc[Dml];
    Dcm[Dmr] = Dc[Dmr];
    for (i=Dml-1; i>=0; i--)
        if (Dc[i] > Dcm[i + 1])
            Dcm[i] = Dc[i];
        else
```

```
            Dcm[i] = Dcm[i + 1];
    for (i=Dmr+1; i<Dn; i++)
        if (Dc[i] > Dcm[i - 1])
            Dcm[i] = Dc[i];
        else
            Dcm[i] = Dcm[i - 1];

    //////////////
    //双指针枚举//
    //////////////
    //最小偏心距ecc
    //由于双指针法略过了单点核Dml/Dmr，所以将其赋为初值
    int ecc;
    ecc = min(Dr[Dml], Dl[Dmr]);
    int l, r;
    l = Dml; r = Dn-1;
    int tMax;
    while (!(l < 0 || r < Dmr))
    {
        //左移r指针使得l<=>r尽可能长
        for (; r>=Dmr; r--)
            if (Dl[r] - Dl[l] <= s)
                break;
        //求max{L[l], R[r], C[l~r]}
        tMax = max(max(Dl[l], Dr[r]), max(Dcm[l], Dcm[r]));
        //替换最优解
        if (tMax < ecc) ecc = tMax;
        //l左移，继续枚举
        l--;
    }

    //输出结果
    ofstream fout("core.out");
    fout << ecc;
    fout.close();

    //释放邻接表空间
    for (i=0; i<n; i++)
    {
        delete Gp[i];
        Gp[i] = NULL;
        delete Gw[i];
        Gw[i] = NULL;
    }

    return 0;
}
```

```
int SubTree_Traverse(int Curr, int Depth)
{
    int i;
    int iTmp, iRetDepth = Depth;
    for (i=0; i<Gn[Curr]; i++)
    {
        if ((!Df[Gp[Curr][i]]) && (Curr!=Gp[Curr][i]))
        {
            Df[Curr] = true;
                iTmp = SubTree_Traverse(Gp[Curr][i], Depth +
Gw[Curr][i]);
            if (iTmp > iRetDepth)
            {
                iRetDepth = iTmp;
            }
        }
    }
    return iRetDepth;
}

void DP_Traverse(int Prev, int Curr)
{
    int i;
    //初值
    Lmax1[Curr] = 0;
    Emax1[Curr] = Curr;
    Vmax1[Curr] = 1;
    Nmax1[Curr] = Curr;
    Lmax2[Curr] = 0;
    Emax2[Curr] = Curr;
    Vmax2[Curr] = 1;
    Nmax2[Curr] = Curr;
    for (i=0; i<Gn[Curr]; i++)
    {
        //略过路径Curr->Curr, 否则会失去防回溯保护(Prev参数)
        if (Gp[Curr][i] != Prev && Gp[Curr][i] != Curr)
        {
            //解子结点
            DP_Traverse(Curr, Gp[Curr][i]);

            //替换当前最优/次优解
                    if ((Lmax1[Gp[Curr][i]] + Gw[Curr][i] >
Lmax1[Curr])
                     || (Lmax1[Gp[Curr][i]] + Gw[Curr][i] ==
Lmax1[Curr]
                                    && Vmax1[Gp[Curr][i]] + 1 <
Vmax1[Curr]))
```

```
            {
                Lmax2[Curr] = Lmax1[Curr];
                Emax2[Curr] = Emax1[Curr];
                Vmax2[Curr] = Vmax1[Curr];
                Nmax2[Curr] = Nmax1[Curr];
                  Lmax1[Curr] = Lmax1[Gp[Curr][i]] + Gw[Curr]
[i];
                Emax1[Curr] = Emax1[Gp[Curr][i]];
                Vmax1[Curr] = Vmax1[Gp[Curr][i]] + 1;
                Nmax1[Curr] = Gp[Curr][i];
            }
                else if ((Lmax1[Gp[Curr][i]] + Gw[Curr][i] >
Lmax2[Curr])
                     || (Lmax1[Gp[Curr][i]] + Gw[Curr][i] ==
Lmax2[Curr]
                    && Vmax1[Gp[Curr][i]] + 1 < Vmax2[Curr]))
            {
                  Lmax2[Curr] = Lmax1[Gp[Curr][i]] + Gw[Curr]
[i];
                Emax2[Curr] = Emax1[Gp[Curr][i]];
                Vmax2[Curr] = Vmax1[Gp[Curr][i]] + 1;
                Nmax2[Curr] = Gp[Curr][i];
            }
        }
    }
}

void LoadData()
{
    int i;
    //读入数据，构建数据结构
    //读入结点数n
    ifstream fin("core.in");
    fin >> n;
    fin >> s;
    //初始化邻接表计数器
    for (i=0; i<n; i++)
        Gn[i] = 1; //自身算作邻接点
    //读入边集，对邻接点计数
    int iVertA[N_MAX], iVertB[N_MAX], iWgt[N_MAX];
    for (i=0; i<n-1; i++)
    {
        fin >> iVertA[i] >> iVertB[i] >> iWgt[i];
        iVertA[i]--; iVertB[i]--;
        Gn[iVertA[i]]++;
        Gn[iVertB[i]]++;
    }
    fin.close();
```

```
    //初始化邻接表
    //分配邻接表内存
    for (i=0; i<n; i++)
    {
        Gp[i] = new int[Gn[i]];
        Gw[i] = new int[Gn[i]];
    }
    //重置邻接点计数器
    for (i=0; i<n; i++)
        Gn[i] = 1;
    //添加自身为邻接点
    for (i=0; i<n; i++)
        Gp[i][0] = i;
    for (i=0; i<n; i++)
        Gw[i][0] = 0;
    //重新读入边集, 构造邻接表, 对邻接点重新计数
    for (i=0; i<n-1; i++)
    {
        //Va -> Vb
        Gp[iVertA[i]][Gn[iVertA[i]]] = iVertB[i];
        Gw[iVertA[i]][Gn[iVertA[i]]] = iWgt[i];
        Gn[iVertA[i]]++;
        //Vb -> Va
        Gp[iVertB[i]][Gn[iVertB[i]]] = iVertA[i];
        Gw[iVertB[i]][Gn[iVertB[i]]] = iWgt[i];
        Gn[iVertB[i]]++;
    }
}
```

朴素算法参考程序(C++，$O(n^2)$)

/* 本题 300 的规模太小，不需要太多的优化就可以全过。所以，给出这个想法简单，实现容易的程序。$O(n^2)$的时间复杂度对于 300 的规模也很不错。

程序流程：

1．每对顶点间距离：以每个结点为起点，遍历整棵树 n 次，$O(n^2)$；

2．寻找直径：朴素算法，扫描每对顶点间距离，$O(n^2)$；

3．枚举的预处理：遍历求 Li, Ri, Ci，$O(n^2)$；

4．求 max{Ci~Cj}：朴素的动态规划，预处理 $O(n^2)$，每次查询 $O(1)$；

5．枚举核：暴力枚举 $O(k^2)$。(k 是直径结点数，k≤n) */

```
#include <cstdlib>
#include <iostream>
#include <fstream>
using namespace std;
const int INF = 999999999;
```

```
int n, s;
//图的邻接表
void LoadData();
void RecycleMem_Graph();
int Gn[300]; //Gn[n]: 点i的邻接点数量
int Gp[300][300]; //Gp[n][Gn[i]]: 点i的各个邻接点的编号
int Gw[300][300]; //Gp[n][Gn[i]]: 点i到各个邻接点的权值

//图的直径
int dstart, dend; //任意一条直径的两个端点
int Ds; //直径dstart->dend长度
int Dn; //直径dstart->dend上结点的数量
int Dcl, Dcr; //距直径中心点最近的两结点
int Dp[300]; //直径dstart->dend上各结点的序号.
int Dc[300]; //以Dp[i]为根的子树的最大深度
int Dcmax[300][300]; //Dc[i]~Dc[j]的最大值
int Dl[300], Dr[300]; //到左右端点的距离.
bool Df[300]; //结点i是否在路径dstart->dend上
int SubTree_Traverse(int Src, int Prev, int Curr);

//路径
void Route_Traverse(int Src, int Prev, int Curr, int VertexCount, int Len);
int L[300][300]; //L[i][j] i->j路径长度
int P[300][300]; //P[i][j] i->j路径的前驱 - i->(P[i][j])路径能导出i->j路径
int V[300][300]; //V[i][j] i->j路径上结点的个数

int main()
{
    int i, j;
    LoadData();

    //遍历求路径
    for (i=0; i<n; i++)
    {
        Route_Traverse(i, i, i, 1, 0);
    }

    //直径初始化
    //直径起始点dstart, dend; 直径长度Ds
    Ds = 0;
    for (i=0; i<n; i++)
    {
        for (j=0; j<n; j++)
```

```
        {
            if (L[i][j] > Ds)
            {
                Ds = L[i][j];
                dstart = i;
                dend = j;
            }
        }
    }
    //结点数量 Dn
    Dn = V[dstart][dend];
    //结点编号 Dp[Dn]
    j = dend;
    for (i=Dn-1; i>=0; i--)
    {
        Dp[i] = j;
        j = P[dstart][j];
    }
    //结点标志位 Df[n]
    for (i=0; i<n; i++)
    {
        Df[i] = false;
    }
    for (i=0; i<Dn; i++)
    {
        Df[Dp[i]] = true;
    }
    //左右长度 Dl[Dn], Dr[Dn]
    for (i=0; i<Dn; i++)
    {
        Dl[i] = L[dstart][Dp[i]];
        Dr[i] = Ds - Dl[i];
    }
    //中心结点 Dcl, Dcr
    for (i=0; i<Dn; i++)
    {
        if (Dl[i] * 2 > Ds) //中心结点有 2 个
        {
            Dcr = i;
            Dcl = i - 1;
            break;
        }
         else if (Dl[i] * 2 == Ds) //中心结点只有 1 个(与中心点重
合)
        {
            Dcl = Dcr = i;
            break;
```

```
        }
        else
        {
            continue;
        }
    }
    //子树最大深度Dc[Dn]
    for (i=0; i<Dn; i++)
    {
        Dc[i] = SubTree_Traverse(Dp[i], Dp[i], Dp[i]);
    }
    //Dc[i]~Dc[j]最大值Dcmax[Dn][Dn]
    for (i=0; i<Dn; i++)
    {
        Dcmax[i][i] = Dc[i];
    }
    int t;
    for (t=1; t<Dn; t++)
    {
        i = 0; j = t;
        while (i < Dn && j < Dn)
        {
            Dcmax[i][j] = max(Dcmax[i][j - 1], Dc[j]);
            Dcmax[j][i] = Dcmax[i][j];
            i++; j++;
        }
    }

    //枚举核
    int ecc = INF;
    int tmax;
    for (i=0; i<Dn; i++)
    {
        for (j=0; j<Dn; j++)
        {
                        //ecc(current)  =  max{Dl[i],  Dr[j],
max{Dc[i]~Dc[j]}}
            tmax = Dcmax[i][j];
            if (Dl[i] > tmax) tmax = Dl[i];
            if (Dr[j] > tmax) tmax = Dr[j];
            if (tmax < ecc)
            {
                ecc = tmax;
            }
        }
    }

    ofstream fout("core.out");
```

```
    fout << ecc;
    fout.close();
    return 0;
}

int SubTree_Traverse(int Src, int Prev, int Curr)
{
    int i;
    int iDepth, iTmp;
    iDepth = L[Src][Curr];
    for (i=0; i<Gn[Curr]; i++)
    {
        //直径上的结点不遍历
        //略过路径Curr->Curr, 否则会失去防回溯保护(Prev参数)
         if (Gp[Curr][i] != Prev && Gp[Curr][i] != Curr && (!
Df[Gp[Curr][i]]))
        {
            iTmp = SubTree_Traverse(Src, Curr, Gp[Curr][i]);
            if (iTmp > iDepth) iDepth = iTmp;
        }
    }
    return iDepth;
}

void  Route_Traverse(int  Src,  int  Prev,  int  Curr,  int
VertexCount, int Len)
{
    int i;
    L[Src][Curr] = Len;
    P[Src][Curr] = Prev;
    V[Src][Curr] = VertexCount;
    for (i=0; i<Gn[Curr]; i++)
    {
        //略过路径Curr->Curr, 否则会失去防回溯保护(Prev参数)
        if (Gp[Curr][i] != Prev && Gp[Curr][i] != Curr)
        {
            Route_Traverse(Src, Curr, Gp[Curr][i],
                                  VertexCount + 1, Len + Gw[Curr]
[i]);
        }
    }
}

void LoadData()
{
    int i;
    //读入数据, 构建数据结构
    //读入结点数n
```

```
    ifstream fin("core.in");
    fin >> n;
    fin >> s;
    //初始化邻接表计数器
    for (i=0; i<n; i++)
        Gn[i] = 1; //自身算作邻接点
    //读入边集，对各个点邻接边进行计数
    int iVertA[300], iVertB[300], iWgt[300];
    for (i=0; i<n-1; i++)
    {
        fin >> iVertA[i] >> iVertB[i] >> iWgt[i];
        iVertA[i]--; iVertB[i]--;
        Gn[iVertA[i]]++;
        Gn[iVertB[i]]++;
    }
    fin.close();
    //初始化邻接表
    //添加自身为邻接点
    for (i=0; i<n; i++)
        Gn[i] = 1;
    for (i=0; i<n; i++)
        Gp[i][0] = i;
    for (i=0; i<n; i++)
        Gw[i][0] = 0;
    //重新读入边集，构造邻接表
    for (i=0; i<n-1; i++)
    {
        //Va -> Vb
        Gp[iVertA[i]][Gn[iVertA[i]]] = iVertB[i];
        Gw[iVertA[i]][Gn[iVertA[i]]] = iWgt[i];
        Gn[iVertA[i]]++;
        //Vb -> Va
        Gp[iVertB[i]][Gn[iVertB[i]]] = iVertA[i];
        Gw[iVertB[i]][Gn[iVertB[i]]] = iWgt[i];
        Gn[iVertB[i]]++;
    }
}
```