

**算法导论实验报告**

|  |  |
| --- | --- |
| 姓 名： |  |
| 学 院： |  |
| 专 业： |  |
| 班 级： |  |
| 学 号： |  |
| 指导教师： |  |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

年 月 日

目 录

[1. 最近点对问题 1](#_Toc501821650)

[1.1. 题目描述 1](#_Toc501821651)

[1.2. 算法设计 1](#_Toc501821652)

[1.3. 源代码 1](#_Toc501821653)

[1.4. 测试分析 1](#_Toc501821654)

[1.5. 技术总结 1](#_Toc501821655)

[2. 大数乘法 2](#_Toc501821656)

[2.1. 题目描述 2](#_Toc501821657)

[2.2. 算法设计 2](#_Toc501821658)

[2.3. 源代码 2](#_Toc501821659)

[2.4. 测试分析 2](#_Toc501821660)

[2.5. 技术总结 2](#_Toc501821661)

[3. 最优二分查找树 3](#_Toc501821662)

[3.1. 题目描述 3](#_Toc501821663)

[3.2. 算法设计 3](#_Toc501821664)

[3.3. 源代码 3](#_Toc501821665)

[3.4. 测试分析 3](#_Toc501821666)

[3.5. 技术总结 3](#_Toc501821667)

[4. Floyd Warshell求最短路径 4](#_Toc501821668)

[4.1. 题目描述 4](#_Toc501821669)

[4.2. 算法设计 4](#_Toc501821670)

[4.3. 源代码 4](#_Toc501821671)

[4.4. 测试分析 4](#_Toc501821672)

[4.5. 技术总结 4](#_Toc501821673)

[5. Wooden Sticks ( POJ 1065 ) 5](#_Toc501821674)

[5.1. 题目描述 5](#_Toc501821675)

[5.2. 算法设计 5](#_Toc501821676)

[5.3. 源代码 5](#_Toc501821677)

[5.4. 测试分析 5](#_Toc501821678)

[5.5. 技术总结 5](#_Toc501821679)

[6. Gone fishing ( POJ 1042 ) 6](#_Toc501821680)

[6.1. 题目描述 6](#_Toc501821681)

[6.2. 算法设计 6](#_Toc501821682)

[6.3. 源代码 6](#_Toc501821683)

[6.4. 测试分析 6](#_Toc501821684)

[6.5. 技术总结 6](#_Toc501821685)

[7. Corn Field ( POJ 3254 ) 7](#_Toc501821686)

[7.1. 题目描述 7](#_Toc501821687)

[7.2. 算法设计 7](#_Toc501821688)

[7.3. 源代码 7](#_Toc501821689)

[7.4. 测试分析 7](#_Toc501821690)

[7.5. 技术总结 7](#_Toc501821691)

[8. Paid Roads ( POJ 3411 ) 8](#_Toc501821692)

[8.1. 题目描述 8](#_Toc501821693)

[8.2. 算法设计 8](#_Toc501821694)

[8.3. 源代码 8](#_Toc501821695)

[8.4. 测试分析 8](#_Toc501821696)

[8.5. 技术总结 8](#_Toc501821697)

# 最近点对问题

## 题目描述

输入n个点的x，y坐标，计算n个点中距离最近的点对。

输入：第一行为一个整数，表示测试用例的组数，其后跟相应组数的测试用例。每组测试用例包括：首行一个整数，表示本组测试用例包含的点的个数，其后跟相应个点的数据，每行两个整数，表示该点的x、y坐标。

输出：每行输出一个测试用例的答案，即本测试用例中相距最近的两个点，用点的坐标表示：四个整数。前两个整数表示第一个点，后两个整数表示第二个点。若有多对相距最近的点，依次输出。

## 算法设计

创建一个Point类，用来存储点的信息，包括两个int类型的数据成员，x：点的横坐标，y：点的纵坐标。Point类的结构定义如图 1.1所示。

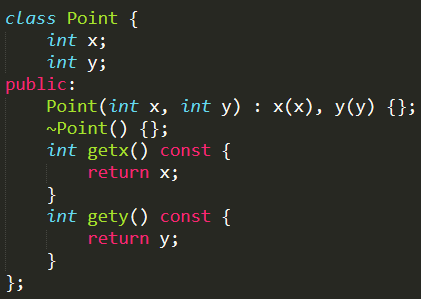


图 1.1 Point类结构定义

因为x、y是Point类的私有成员，外部不能直接访问，所以定义两个函数getx（）和gety（）函数来获取Point类的私有数据成员。

创建一个Distance类，用来存储点对，包括两个Point类型的数据成员，p1和p2，表示点对中的两个点。Distance类的结构定义如图 1.2所示。

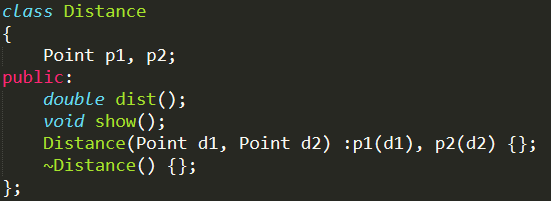


图 1.2 Distance类的结构定义

Distance类有两个成员函数，分别是dist（）和show（）。dist（）函数用来计算两个点的距离的值，函数实现如图 1.3所示。

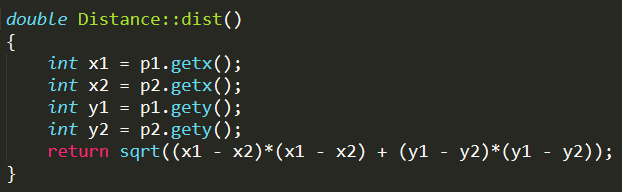


图 1.3 Distance类中dist函数的函数体

show（）函数是用来输出点对信息，当当前点对是最近的点对时，在输出结果时调用此函数即可，函数实现如图 1.4所示。

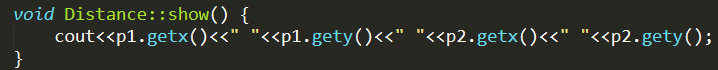


图 1.4 Distance类中show函数的函数体

存储点集合点对集的结构借助vector类来实现，首先包含头文件vector，然后再创建两个vector类的对象，如图 1.5所示。

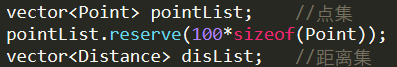


图 1.5 借助vector类来储存数据

其中，在vector后的< >中间填入向量元素的数据类型，如int、char等，此处分别创建两个以Point类和Distance类为基本数据元素的向量pointList和disList，分别用来存储点和点对。其中，调用pointList类的reserve（）函数预留出100个点的空间大小，目的是避免在加入新的点时类反复调用分配新的内存空间的函数，提高运行的效率，而disList中的数据量不多，所以没有进行预留。vector向量插入新元素的方法是调用vector类的push\_back（）函数，将一个新的元素加入到向量的尾部，如图 1.6所示。

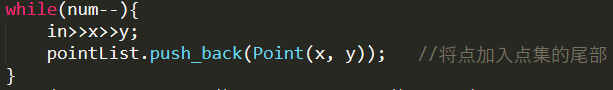


图 1.6 调用push\_back函数插入新的点

调用sort（）函数对点集进行排序，按照x坐标升序进行排序，如图 1.7所示。



图 1.7 对点集进行排序

sort（）函数的最后一个参数comp是一个函数指针，该函数的函数体如图 1.8所示。

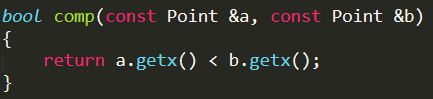


图 1.8 comp函数的函数体实现

排序完成后使用分治法的思想来得到最近点对：

如果点集中点的个数为0或1时，直接返回。

如果点集中点的个数为2，则计算两个点之间的距离之后与目前的最近点对之间的距离进行比较：如果目前最近点对集为空，则直接将当前两个点创建一个Distance类插入到距离集中；如果当前两个点的距离比目前最近点对的距离要小，则先将距离集清空，再将创建的Distance类插入到距离集中；如果当前两个点的距离和目前最近点对的距离相等，那么直接将创建的Distance类插入到距离集的队尾。

如果点集中点的个数大于2，那么将点集等分成左右两个子点集，如果是奇数个点，将最中间的点划分到左子点集中，然后对两个子点集递归调用分治函数分别计算两个子点集中各自的最近点对。在两个子点集返回之后，再计算跨过中线的点对之间的距离。

分治法的运行流程如图 1.9所示。

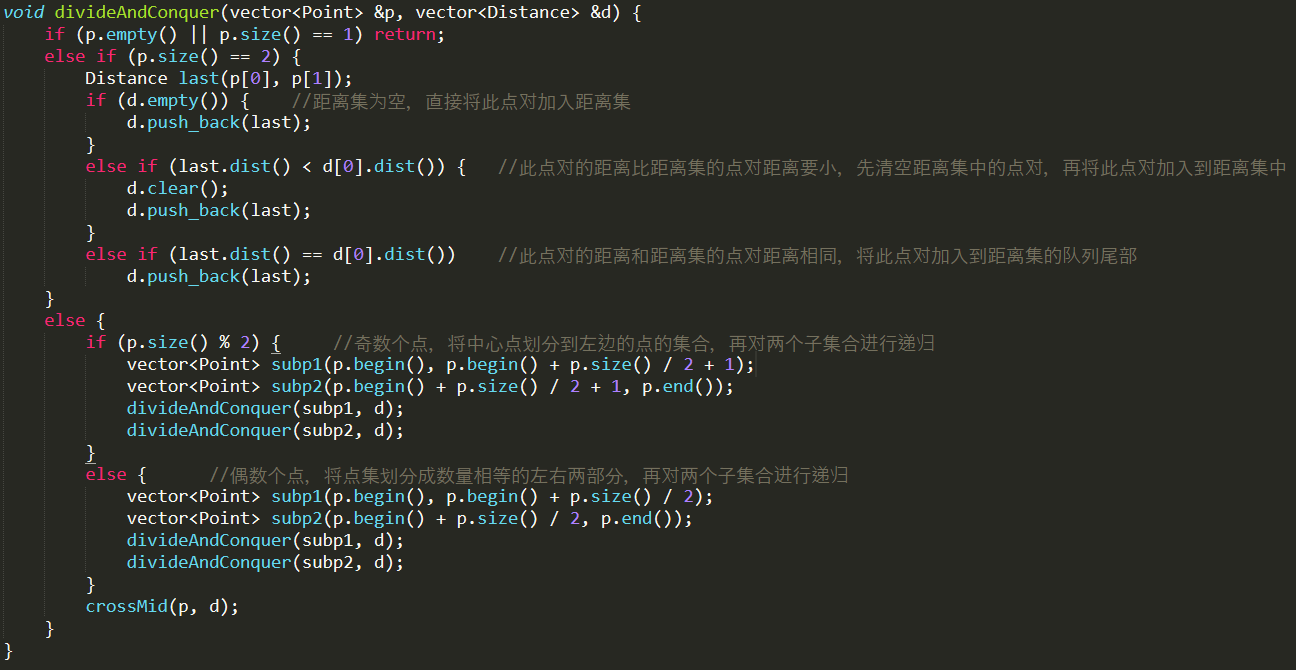


图 1.9 利用分治法思想计算最近点对

在crossMid（）函数中，计算跨过中间线的点对之间的距离。因为此时已经存在最近点对，所以要满足跨过中心线的点对的距离要比当前的最近点对的距离小，首先这些点到中线的距离肯定要比目前的最近点对的距离要小，可以根据这一条件排除一些明显不满足条件的点，如图 1.10所示。

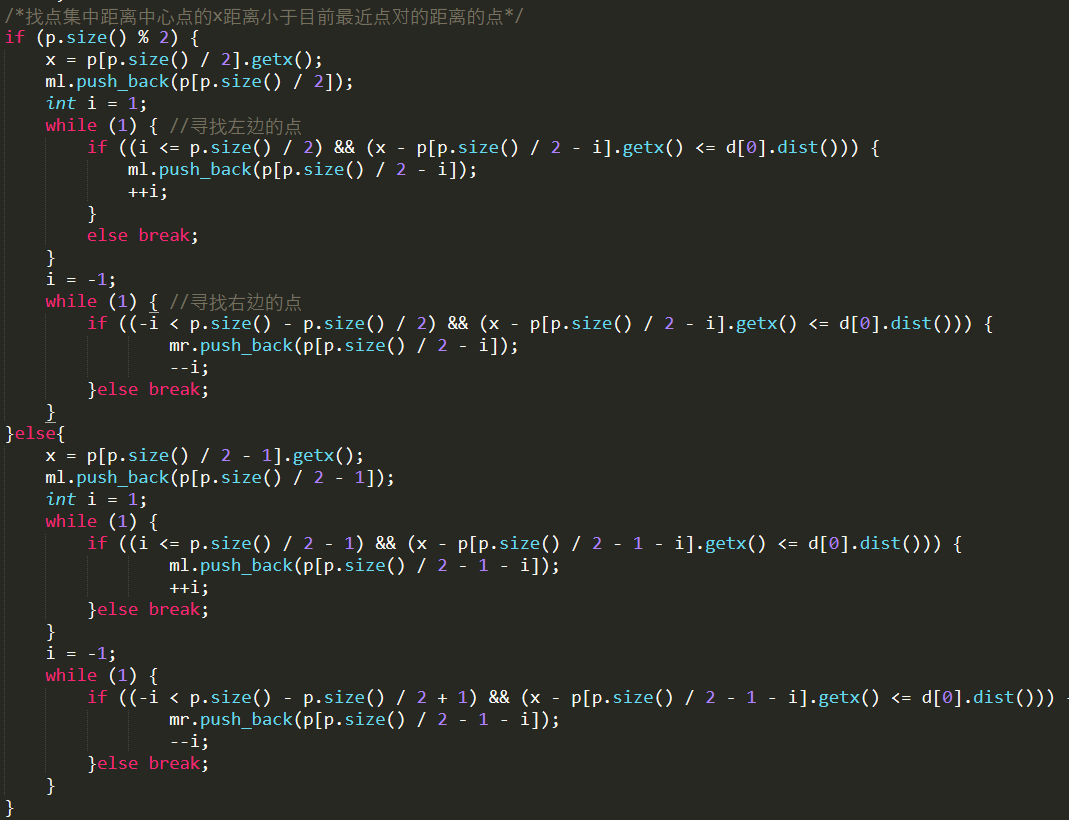


图 1.10 排除离中间线距离大与最近点对距离的点

在分别获取中心线左边和右边满足离中间线距离小于目前最近点对的点的集合后，对左边和右边的点之间的距离一一去计算，并按照分治法中两个点距离与当前最近点对的距离的处理方法进行处理，如图 1.11所示。

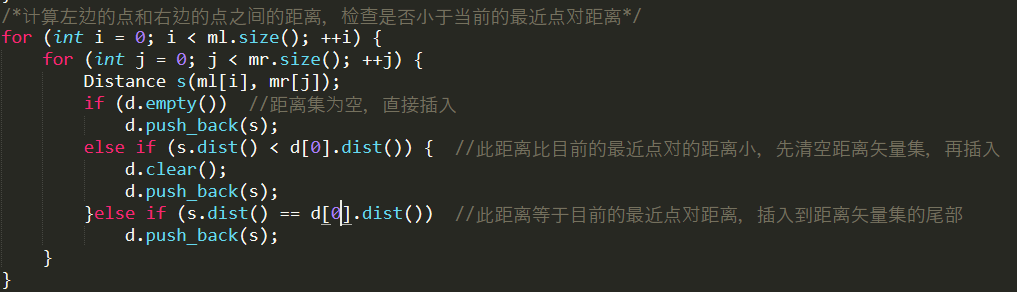


图 1.11 计算此点集中跨过中间线的点对距离

至此，左子点集中的最近点对、右子点集中的最近点对、跨过中间线的最近点对都已得到，因此此时得到的最近点对即为整个点集的最近点对。因此，将距离集中的点对输出即可，如图 1.12所示。

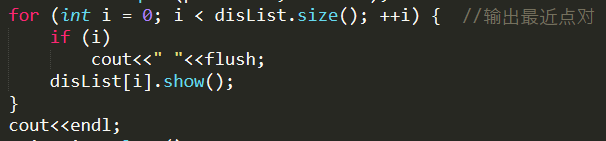


图 1.12 输出当前测试用例的结果

在输出结果之后，调用vector类的clear（）函数清空类中地数据，为下一次计算做准备，如图1.13所示。



图1.13 点集和距离集的清空

## 源代码

#include <iostream>

#include <algorithm>

#include <vector>

#include <math.h>

#include <fstream>

using namespace std;

class Point {

int x;

int y;

public:

Point(int x, int y) : x(x), y(y) {};

~Point() {};

int getx() const {

return x;

}

int gety() const {

return y;

}

};

class Distance

{

Point p1, p2;

public:

double dist();

void show();

Distance(Point d1, Point d2) :p1(d1), p2(d2) {};

~Distance() {};

};

double Distance::dist()

{

int x1 = p1.getx();

int x2 = p2.getx();

int y1 = p1.gety();

int y2 = p2.gety();

return sqrt((x1 - x2)\*(x1 - x2) + (y1 - y2)\*(y1 - y2));

}

void Distance::show() {

cout<<p1.getx()<<" "<<p1.gety()<<" "<<p2.getx()<<" "<<p2.gety();

}

bool comp(const Point &a, const Point &b)

{

return a.getx() < b.getx();

}

void crossMid(vector<Point> &p, vector<Distance> &d) {

vector<Point> ml;

vector<Point> mr;

int x;

/\*找点集中距离中心点的x距离小于目前最近点对的距离的点\*/

if (p.size() % 2) {

x = p[p.size() / 2].getx();

ml.push\_back(p[p.size() / 2]);

int i = 1;

while (1) { //寻找左边的点

if ((i <= p.size() / 2) && (x - p[p.size() / 2 - i].getx() <= d[0].dist())) {

ml.push\_back(p[p.size() / 2 - i]);

++i;

}

else break;

}

i = -1;

while (1) { //寻找右边的点

if ((-i < p.size() - p.size() / 2) && (x - p[p.size() / 2 - i].getx() <= d[0].dist())) {

mr.push\_back(p[p.size() / 2 - i]);

--i;

}else break;

}

}else{

x = p[p.size() / 2 - 1].getx();

ml.push\_back(p[p.size() / 2 - 1]);

int i = 1;

while (1) {

if ((i <= p.size() / 2 - 1) && (x - p[p.size() / 2 - 1 - i].getx() <= d[0].dist())) {

ml.push\_back(p[p.size() / 2 - 1 - i]);

++i;

}else break;

}

i = -1;

while (1) {

if ((-i < p.size() - p.size() / 2 + 1) && (x - p[p.size() / 2 - 1 - i].getx() <= d[0].dist())) {

mr.push\_back(p[p.size() / 2 - 1 - i]);

--i;

}else break;

}

}

/\*计算左边的点和右边的点之间的距离，检查是否小于当前的最近点对距离\*/

for (int i = 0; i < ml.size(); ++i) {

for (int j = 0; j < mr.size(); ++j) {

Distance s(ml[i], mr[j]);

if (d.empty()) //距离集为空，直接插入

d.push\_back(s);

else if (s.dist() < d[0].dist()) { //此距离比目前的最近点对的距离小，先清空距离矢量集，再插入

d.clear();

d.push\_back(s);

}else if (s.dist() == d[0].dist()) //此距离等于目前的最近点对距离，插入到距离矢量集的尾部

d.push\_back(s);

}

}

}

void divideAndConquer(vector<Point> &p, vector<Distance> &d) {

if (p.empty() || p.size() == 1) return;

else if (p.size() == 2) {

Distance last(p[0], p[1]);

if (d.empty()) { //距离集为空，直接将此点对加入距离集

d.push\_back(last);

}

else if (last.dist() < d[0].dist()) { //此点对的距离比距离集的点对距离要小，先清空距离集中的点对，再将此点对加入到距离集中

d.clear();

d.push\_back(last);

}

else if (last.dist() == d[0].dist()) //此点对的距离和距离集的点对距离相同，将此点对加入到距离集的队列尾部

d.push\_back(last);

}

else {

if (p.size() % 2) { //奇数个点，将中心点划分到左边的点的集合，再对两个子集合进行递归

vector<Point> subp1(p.begin(), p.begin() + p.size() / 2 + 1);

vector<Point> subp2(p.begin() + p.size() / 2 + 1, p.end());

divideAndConquer(subp1, d);

divideAndConquer(subp2, d);

}

else { //偶数个点，将点集划分成数量相等的左右两部分，再对两个子集合进行递归

vector<Point> subp1(p.begin(), p.begin() + p.size() / 2);

vector<Point> subp2(p.begin() + p.size() / 2, p.end());

divideAndConquer(subp1, d);

divideAndConquer(subp2, d);

}

crossMid(p, d);

}

}

int main() {

int n; //测试样例个数

int num; //当前测试样例点的个数

int x, y; //用来接收点的坐标

ifstream in("in.dat"); //从文件中读入数据

if (!in.is\_open()){

cout<<"fail to open file : in.dat"<<endl;

return -1;

}

vector<Point> pointList; //点集

pointList.reserve(100\*sizeof(Point));

vector<Distance> disList; //距离集

in>>n;

while(n--){

in>>num;

while(num--){

in>>x>>y;

pointList.push\_back(Point(x, y)); //将点加入点集的尾部

}

sort(pointList.begin(), pointList.end(), comp); //根据x坐标排序

divideAndConquer(pointList, disList);

for (int i = 0; i < disList.size(); ++i) { //输出最近点对

if (i)

cout<<" "<<flush;

disList[i].show();

}

cout<<endl;

pointList.clear();

disList.clear();

}

system("pause");

return 0;

}

## 测试分析

自测数据输入如下所示：

|  |
| --- |
| 2  4  1 9  3 8  4 6  8 1  5  10 8  23 98  18 32  78 43  76 25 |

其中，第一行的2表示有两组测试用例。第二行的4表示第一组测试用例有四个点，其后跟四行，依次为四个点的坐标（1,9）、（3,8）、（4,6）、（8,1）。第七行的5代表第二组测试用例有五个点，其后跟五行，依次为五个点的坐标（10,8），（23,98）、（18,32）、（78,43）、（76,25）。程序运行结果如图 1.13所示。

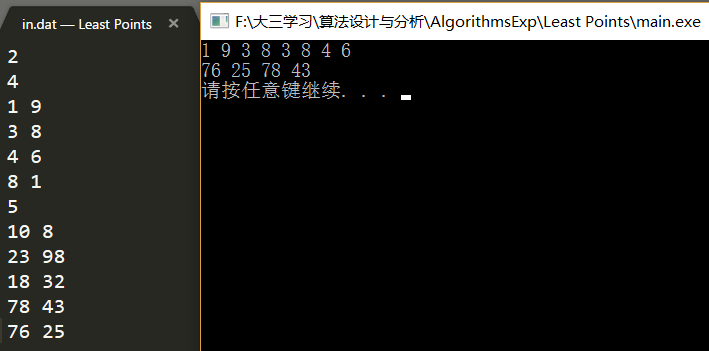


图 1.13 自测数据测试结果

验算可知，在第一组的四个节点中，点（1,9）与点（3,8）、点（3,8）与点（4,6）两组点对的距离是，其他的点对之间的距离都比这个值要大；在第二组中的五个结点中，点（76,25）与点（78,43）之间的距离是，同样比其他的点对之间的距离要小。因此，程序运行两个测试用例的输出结果都与正确答案相符，可以得到对自测数据的运行是正确。

统测数据如下所示：

|  |
| --- |
| 2  4  1 8  9 5  10 7  -1 7  2  1 8  9 4 |

其中，第一行的2表示有两组测试用例。第二行的4表示第一组测试用例有四个点，其后跟四行，依次为四个点的坐标（1,8）、（9,5）、（10,7）、（-1,7）。第七行的2代表第二组测试用例有两个点，其后跟两行，依次为两个点的坐标（1,8），（9,4）。程序运行结果如图 1.14所示。

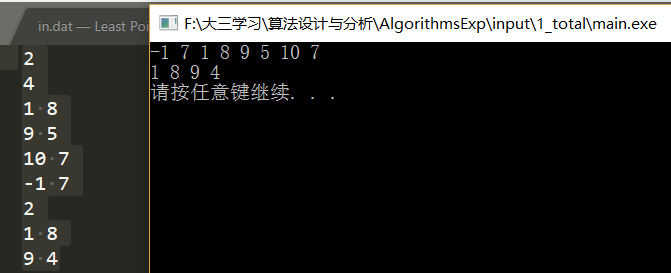


图 1.14 统测数据测试结果

验算可知，在第一组的四个节点中，点（-1,7）与点（1,8）、点（9,5）与点（10,7）两组点对的距离是，其他的点对之间的距离都比这个值要大；在第二组中的五个结点中，点（1,8）与点（9,4）之间的距离是，同样比其他的点对之间的距离要小。因此，程序运行两个测试用例的输出结果都与正确答案相符，可以得到对统测数据的运行是正确。

可能出现点集中的点的个数为0以及有两个点重合的情况，测试数据如下所示：

|  |
| --- |
| 2  0  3  1 8  2 9  1 8 |

其中，第一行的2表示有两组测试用例。第二行的0表示第一组测试用例中没有点。第三行的3代表第二组测试用例有三个点，其后跟三行，依次为三个点的坐标（1,8），（2,9），（1,8）。程序运行结果如图 1.15所示。

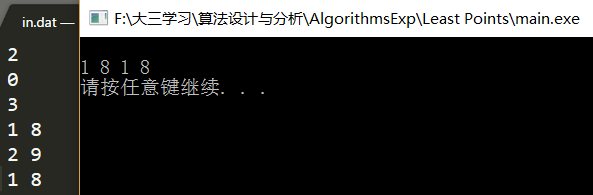


图 1.15 点集没有结点以及结点重合情况测试结果

在第一组测试用例中，因为没有结点，所以输出为一个空行。在第二组测试用例中，有两个点处在（1,8）位置，他们之间的距离为0，所以两个点组成最近点对。因此，对于点集中没有点和点集中结点重合两种情况都能正确输出结果。

## 技术总结

本题是一个在包含n个点的集合Q中寻找最近点对的问题。“最近”指的是通常意义下的欧几里得距离：点之间的欧几里得距离为。利用分治法的思想解决最近点对的问题，算法的每一次递归调用的输入为已经按照横左边x排好序的点集P和目前最近点对的距离集D。

如果本次递归的点集中的点的个数为1，则直接返回；若个数为2，则直接结算两个点之间的距离，并判断是否需要更新当前的最近点对的集合；如果大于等于3，则进入分治的操作。

首先在点集P中找到一根中间线，将整个点集分成两个子集合。因为点集P是已经排好序的，所以可以直接通过点集P中点的个数来将点集对半分成两半。

将点集划分成两个子集合后，分别对两个子集合进行递归，分别得到中的最近点对的距离和中的最近点对的距离，此时置目前的最近点对的距离。此时已经得到了中间线两侧的点的最小值了，接下来要计算的是跨过中间线（或者是处在中间线上）的两个点之间的距离。如果一个点到中间线的距离大于，那么不存在中间线上的点或者是中间线另外一边的点和这个点之间的距离小于，所以可以根据这个原理从两侧的点集中筛选出距离中间线的距离小于或者是等于的点组成的集合。对中的点一一配对计算两点之间的距离并与比较，将更新为两者中较小的值。在一一配对结束后，此点集中两个子点集中的最近点对的距离、跨越中间线的点对的最短距离都已经覆盖到了，此时的代表的便是本点集中最近点对的距离，返回即可。

# 大数乘法

## 题目描述

利用分治法设计一个计算两个n位的大整数相乘的算法，要去时间复杂度低于。

大整数（big interger）：位数很多的整数，普通的计算机不能直接处理，如：9834975972130802345791023498570345，对大整数的算数运算，显然常规程序语言是无法直接表示的。编程实现大整数的加、减、乘运算需考虑操作数为0、负数、任意位等各种情况。

输入：第一行为一个整数，表示测试用例的组数，其后跟相应组数的测试用例，每个测试用例一行，包含3个整数（长整数数字串），前两个是待测试的操作数，第3个整数表示操作类型（1：加法，2：减法，3：乘法）

输出：每组测试用例输出一行，最后不要加空行。

## 算法设计

由于参与运算的两个数据的位数超过了一般的整数的存储范围，所以考虑借助字符串来存储两个数据，并采取字符串的操作进行两个数据的运算。首先先把两个数据读入到string类型的字符串num1和num2中，如图 2.1所示。

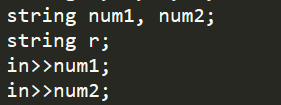


图 2.1 读取两个大整数

利用int型的变量读入第三个参数，如果当前操作为乘法，则先获取两个数据的符号，可以直接访问字符串类型的第一个元素并与’-’进行比较即可，如果是负数的话调用substr函数去掉第一位的符号位，最后的结果的符号位两个数据符号位的异或，获取符号的操作如图所示。

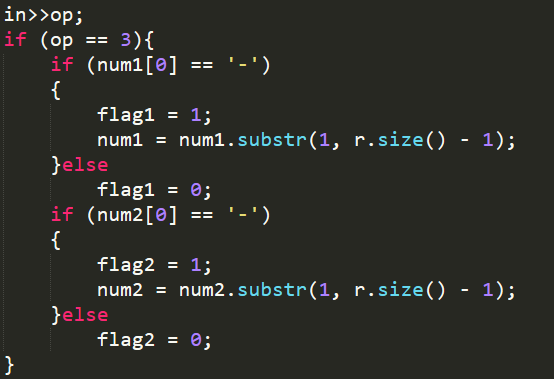


图 2.2 提取数据的符号

根据操作类型op的不同，对数据执行不同的算法，如图 2.3所示。

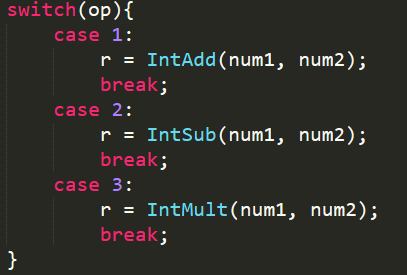


图 2.3 选择进行的操作

再得到r的数据后，因为在字符串的操作中可能为了方便计算使两个数据保持相同的位数会在数据的高位补上0，所以要先对结果的高位的0进行修剪，修剪后再进行输出，如果是乘法还要加上符号（加减法的符号可以通过相互调用来解决符号的问题），如图 2.4所示。

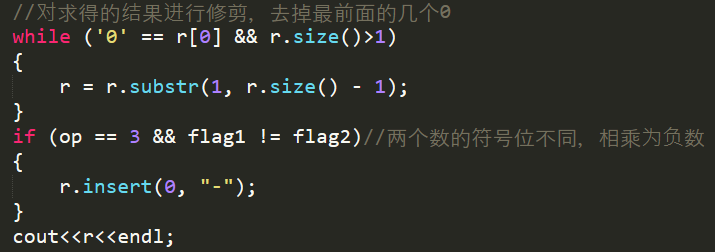


图 2.4 运算结果的输出

对于大整数的加法，首先对数据的符号进行判断，如果两个数都为负，则将两个数据的符号位去掉，按照两个正数来相加，在加完后填上一个负号；如果两者异号，则调用减法来计算。如图 2.5所示。

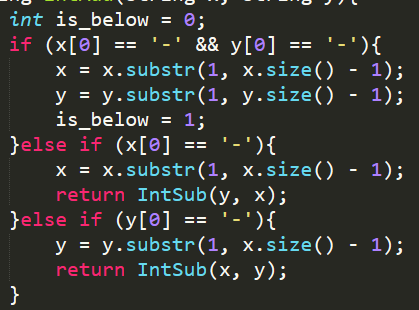


图 2.5 加法中数据的处理

从两个字符串的低位开始逐位相加，需要注意的是，通过对字符串类型去下标，如num1[10]返回的是对应数字的ASCLL码，所以在两个数字相加后要减去一个’0’。如果上述结果大于10，要将结果减去10，并向上一位进位，按照这个策略逐位的相加，直到某一个字符串访问结束，如图 2.6所示。

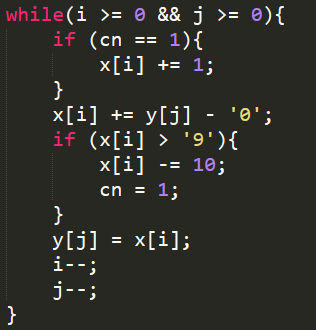


图 2.6 按位进行加法操作

执行完按位的加法后，要判断是否有进位没有加上和符号的判断，如图所示。

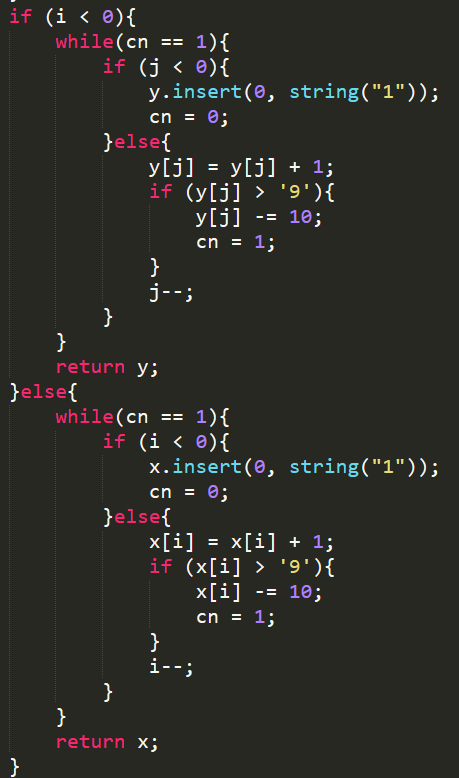


图 2.7 执行按位加法后对进位和符号位的修正

减法的运行流程与加法类似，但是在进行运算之前要先将两个数统一成相同的位数，如果两个数的位数不相同，则通过在数据前补0来补齐，如图 2.8所示。

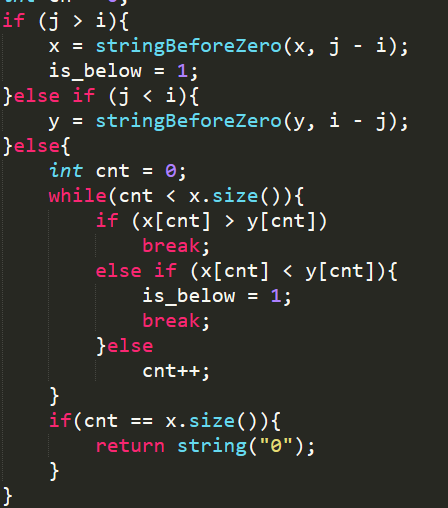


图 2.8 补齐数据和判断结果符号

从低位开始按位做减法，如果减出来的结果小于0，则向上一位借1，再做减法。在做减法的时候，为了保证运算出来后的数据仍然是一个数字的ASCLL码，要对减法得到的结果加上一个’0’，如图 2.9所示。

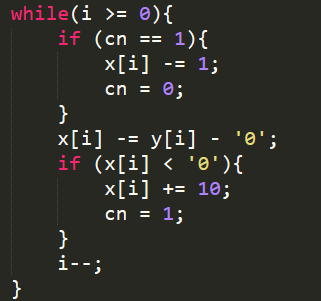


图 2.9 减法的按位减操作

大整数的乘法首先判断两个数的高位是否有多余的0，如果有则通过substr函数去掉，如图 2.10所示。

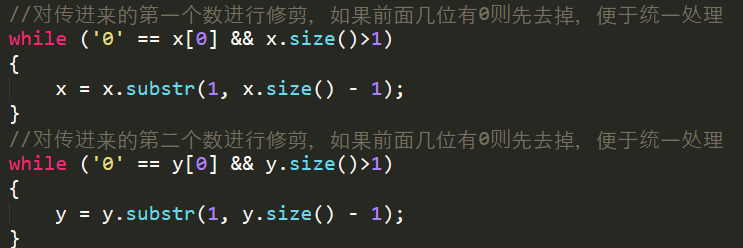


图 2.10 修剪两个数据

为了使用递归的方法来进行大整数乘法，首先把两个整数统一成位的数据。有很多个满足条件的n，此处选择其中最小的n。操作流程如图 2.11所示。

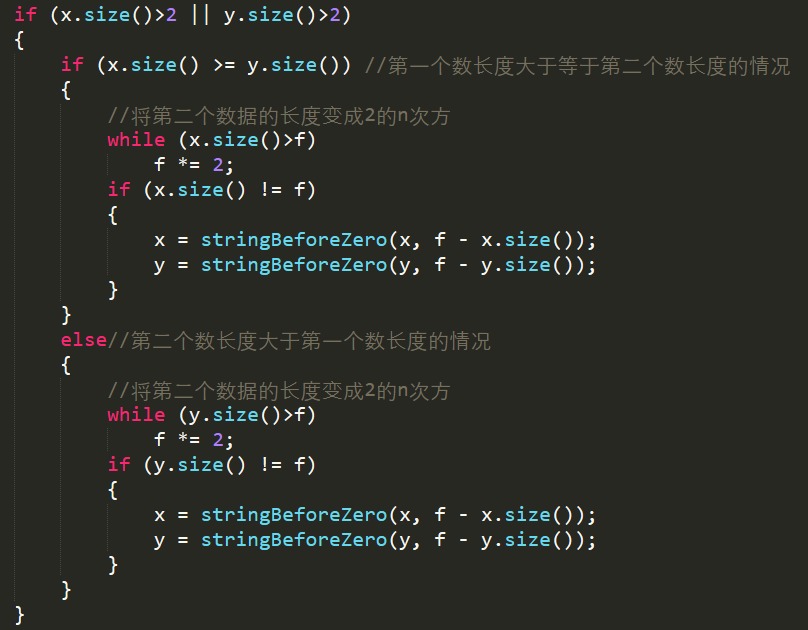


图 2.11 将两个数字统一位数

因为在进入函数时对数据进行了修剪所以有可能数据的位数为1，此时为了能够递归结束，在数据前补上一个0，如图 2.12所示。

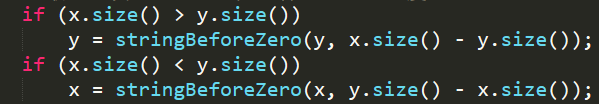


图 2.12 对一位的数据前插入一个0

对于两个大整数的乘法，如c=a\*b，可以根据以下公式对乘法进行分治：

使用分治法对乘法进行分治的流程如图所示。

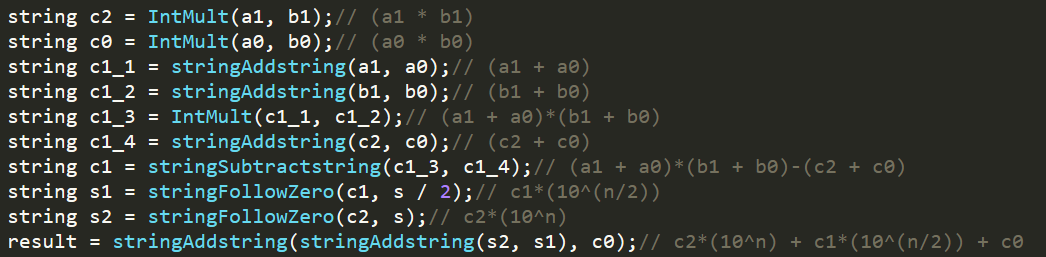


图 2.13 分治法实现大整数乘法

## 源代码

#include <iostream>

#include <sstream>

#include <fstream>

#include <string>

#include <stdlib.h>

using namespace std;

string IntAdd(string x, string y);

//string类型转换成int类型

int string\_to\_num(string k)//string字符串变整数型例如str="1234"，转换为整数的1234.

{

int back;

stringstream stream;

stream << k;

stream >> back;

return back;

}

//整形数转换为string类型

string num\_to\_string(int intValue)

{

string result;

stringstream stream;

stream << intValue;//将int输入流

stream >> result;//从stream中抽取前面放入的int值

return result;

}

//在字符串str前添加s个零

string stringBeforeZero(string str, int s)

{

for (int i = 0;i<s;i++)

{

str.insert(0, "0");

}

return str;

}

//两个大整数字符串相加，超出计算机表示范围的数也能实现相加(本函数可以实现大整数加法运算)

string stringAddstring(string str1, string str2)

{

//假定str1和str2是相等的长度，不相等时在前面自动补零，使两个字符串长度相等

if (str1.size() > str2.size())

{

str2 = stringBeforeZero(str2, str1.size() - str2.size());

}

else if (str1.size() < str2.size())

{

str1 = stringBeforeZero(str1, str2.size() - str1.size());

}

string result;

int flag = 0;//前一进位是否有标志,0代表无进位，1代表有进位

for (int i = str1.size() - 1;i >= 0;i--)

{

int c = (str1[i] - '0') + (str2[i] - '0') + flag;//利用ASCII码对字符进行运算,这里加上flag代表的是:当前一位有进位时加1，无进位时加0

flag = c / 10;//c大于10时，flag置为1，否则为0

c %= 10;//c大于10时取模，否则为其本身

result.insert(0, num\_to\_string(c));//在result字符串最前端插入新生成的单个字符

}

if (0 != flag) //最后一为(最高位)判断，如果有进位则再添一位

{

result.insert(0, num\_to\_string(flag));

}

return result;

}

/\*两个大整数字符串相减，超出计算机表示范围的数也能实现相减(在这里比较特殊，第一个参数一定大于第二个参数,

因为：a1\*b0+a0\*b1=(a1+a0)\*(b1+b0)-(a1\*b1+a0\*b0) > 0 ,所以(a1+a0)\*(b1+b0) > (a1\*b1+a0\*b0)

这个函数的两个参数，第一个代表的其实就是(a1+a0)\*(b1+b0)，第二个代表的其实就是(a1\*b1+a0\*b0)

所以本函数里不用考虑最终得到结果为负数的情况，至于计算有关大整数负数相乘的问题可以通过其他途径判断

\*/

string stringSubtractstring(string str1, string str2)

{

//对传进来的两个数进行修剪，如果前面几位有0则先去掉，便于统一处理

while ('0' == str1[0] && str1.size()>1)

{

str1 = str1.substr(1, str1.size() - 1);

}

while ('0' == str2[0] && str2.size()>1)

{

str2 = str2.substr(1, str2.size() - 1);

}

//使两个字符串长度相等

if (str1.size() > str2.size())

{

str2 = stringBeforeZero(str2, str1.size() - str2.size());

}

string result;

for (int i = str1.size() - 1;i >= 0;i--)

{

int c = (str1[i] - '0') - (str2[i] - '0');//利用ASCII码进行各位减法运算

if (c < 0) //当不够减时向前一位借位，前一位也不够位时再向前一位借位，依次如下

{

c += 10;

int prePos = i - 1;

char preChar = str1[prePos];

while ('0' == preChar)

{

str1[prePos] = '9';

prePos -= 1;

preChar = str1[prePos];

}

str1[prePos] -= 1;

}

result.insert(0, num\_to\_string(c));//在result字符串最前端插入新生成的单个字符

}

return result;

}

//在字符串str后跟随s个零

string stringFollowZero(string str, int s)

{

for (int i = 0;i<s;i++)

{

str.insert(str.size(), "0");

}

return str;

}

string IntSub(string x, string y){

int is\_below = 0;

int change = 0;

if (x[0] == '-' && y[0] == '-'){

x = x.substr(1, x.size() - 1);

y = y.substr(1, y.size() - 1);

change = 1;

}else if (x[0] == '-'){

x = x.substr(1, x.size() - 1);

string r = IntAdd(x, y);

r.insert(0, string("-"));

return r;

}else if (y[0] == '-'){

y = y.substr(1, y.size() - 1);

string r = IntAdd(x, y);

r.insert(0, string("-"));

return r;

}

//对传进来的第一个数进行修剪，如果前面几位有0则先去掉，便于统一处理

while ('0' == x[0] && x.size()>1)

{

x = x.substr(1, x.size() - 1);

}

//对传进来的第二个数进行修剪，如果前面几位有0则先去掉，便于统一处理

while ('0' == y[0] && y.size()>1)

{

y = y.substr(1, y.size() - 1);

}

int i = x.size() - 1, j = y.size() - 1;

int cn = 0;

if (j > i){

x = stringBeforeZero(x, j - i);

is\_below = 1;

}else if (j < i){

y = stringBeforeZero(y, i - j);

}else{

int cnt = 0;

while(cnt < x.size()){

if (x[cnt] > y[cnt])

break;

else if (x[cnt] < y[cnt]){

is\_below = 1;

break;

}else

cnt++;

}

if(cnt == x.size()){

return string("0");

}

}

if (is\_below){

string r = x;

x = y;

y = r;

}

i = x.size();

while(i >= 0){

if (cn == 1){

x[i] -= 1;

cn = 0;

}

x[i] -= y[i] - '0';

if (x[i] < '0'){

x[i] += 10;

cn = 1;

}

i--;

}

if ((is\_below && !change) || (!is\_below && change))

x.insert(0, "-");

return x;

}

string IntAdd(string x, string y){

int is\_below = 0;

if (x[0] == '-' && y[0] == '-'){

x = x.substr(1, x.size() - 1);

y = y.substr(1, y.size() - 1);

is\_below = 1;

}else if (x[0] == '-'){

x = x.substr(1, x.size() - 1);

return IntSub(y, x);

}else if (y[0] == '-'){

y = y.substr(1, x.size() - 1);

return IntSub(x, y);

}

//对传进来的第一个数进行修剪，如果前面几位有0则先去掉，便于统一处理

while ('0' == x[0] && x.size()>1)

{

x = x.substr(1, x.size() - 1);

}

//对传进来的第二个数进行修剪，如果前面几位有0则先去掉，便于统一处理

while ('0' == y[0] && y.size()>1)

{

y = y.substr(1, y.size() - 1);

}

int i = x.size() - 1, j = y.size() - 1;

int cn = 0;

while(i >= 0 && j >= 0){

if (cn == 1){

x[i] += 1;

}

x[i] += y[j] - '0';

if (x[i] > '9'){

x[i] -= 10;

cn = 1;

}

y[j] = x[i];

i--;

j--;

}

if (i < 0){

while(cn == 1){

if (j < 0){

y.insert(0, string("1"));

cn = 0;

}else{

y[j] = y[j] + 1;

if (y[j] > '9'){

y[j] -= 10;

cn = 1;

}

j--;

}

}

return y;

}else{

while(cn == 1){

if (i < 0){

x.insert(0, string("1"));

cn = 0;

}else{

x[i] = x[i] + 1;

if (x[i] > '9'){

x[i] -= 10;

cn = 1;

}

i--;

}

}

return x;

}

}

//分治法大整数乘法实现函数

string IntMult(string x, string y)//递归函数

{

//对传进来的第一个数进行修剪，如果前面几位有0则先去掉，便于统一处理

while ('0' == x[0] && x.size()>1)

{

x = x.substr(1, x.size() - 1);

}

//对传进来的第二个数进行修剪，如果前面几位有0则先去掉，便于统一处理

while ('0' == y[0] && y.size()>1)

{

y = y.substr(1, y.size() - 1);

}

/\*这里的f变量代表在两个数据字符串长度不相等或者不是2的指数倍的情况下，所要统一的长度，这样做是为了让数据长度为2的n次方

的情况下便于利用分治法处理

\*/

int f = 4;

/\*当两字符串中有任意一个字符串长度大于2时都要通过与上面定义的f值进行比较，使其达到数据长度为2的n次方，实现方式是在前面

补0，这样可以保证数据值大小不变

\*/

if (x.size()>2 || y.size()>2)

{

if (x.size() >= y.size()) //第一个数长度大于等于第二个数长度的情况

{

//将第二个数据的长度变成2的n次方

while (x.size()>f)

f \*= 2;

if (x.size() != f)

{

x = stringBeforeZero(x, f - x.size());

y = stringBeforeZero(y, f - y.size());

}

}

else//第二个数长度大于第一个数长度的情况

{

//将第二个数据的长度变成2的n次方

while (y.size()>f)

f \*= 2;

if (y.size() != f)

{

x = stringBeforeZero(x, f - x.size());

y = stringBeforeZero(y, f - y.size());

}

}

}

//数据长度为1时,在前面补一个0(这里之所以会出现长度为1的数据是因为前面对数据修剪过)

if (1 == x.size())

x = stringBeforeZero(x, 1);

if (1 == y.size())

y = stringBeforeZero(y, 1);

//最后一次对数据校正，确保两个数据长度统一

if (x.size() > y.size())

y = stringBeforeZero(y, x.size() - y.size());

if (x.size() < y.size())

x = stringBeforeZero(x, y.size() - x.size());

int s = x.size(); //两个数据的长度

string a1, a0, b1, b0;

if (s > 1)

{

a1 = x.substr(0, s / 2);

a0 = x.substr(s / 2, s - 1);

b1 = y.substr(0, s / 2);

b0 = y.substr(s / 2, s - 1);

}

string result;

if (s == 2) //长度为2时代表着递归的结束条件

{

int na = string\_to\_num(a1);

int nb = string\_to\_num(a0);

int nc = string\_to\_num(b1);

int nd = string\_to\_num(b0);

result = num\_to\_string((na \* 10 + nb) \* (nc \* 10 + nd));

}

else

{ //大于不为2时利用分治法进行递归运算

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

c = a\*b = c2\*(10^n) + c1\*(10^(n/2)) + c0;

a = a1a0 = a1\*(10^(n/2)) + a0;

b = b1b0 = b1\*(10^(n/2)) + b0;

c2 = a1\*b1; c0 = a0\*b0;

c1 = (a1 + a0)\*(b1 + b0)-(c2 + c0);

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

string c2 = IntMult(a1, b1);// (a1 \* b1)

string c0 = IntMult(a0, b0);// (a0 \* b0)

string c1\_1 = stringAddstring(a1, a0);// (a1 + a0)

string c1\_2 = stringAddstring(b1, b0);// (b1 + b0)

string c1\_3 = IntMult(c1\_1, c1\_2);// (a1 + a0)\*(b1 + b0)

string c1\_4 = stringAddstring(c2, c0);// (c2 + c0)

string c1 = stringSubtractstring(c1\_3, c1\_4);// (a1 + a0)\*(b1 + b0)-(c2 + c0)

string s1 = stringFollowZero(c1, s / 2);// c1\*(10^(n/2))

string s2 = stringFollowZero(c2, s);// c2\*(10^n)

result = stringAddstring(stringAddstring(s2, s1), c0);// c2\*(10^n) + c1\*(10^(n/2)) + c0

}

return result;

}

int main()

{

int f = 1;

int n = 0;

int op = 0;

ifstream in("in.dat");

if(!in.is\_open())

{

cout<<"fail to open file : in.dat"<<endl;

return -1;

}

in>>n;

while (n--)

{

int flag1 = 0, flag2 = 0;//记录两个大数的符号，为1是为负数

string A, B, C, D;

string num1, num2;

string r;

in>>num1;

in>>num2;

int i = 0;

in>>op;

if (op == 3){

if (num1[0] == '-')

{

flag1 = 1;

num1 = num1.substr(1, r.size() - 1);

}else

flag1 = 0;

if (num2[0] == '-')

{

flag2 = 1;

num2 = num2.substr(1, r.size() - 1);

}else

flag2 = 0;

}

switch(op){

case 1:

if ((flag1 | flag2) && !(flag1 & flag2))

r = stringSubtractstring(num1, num2);

else

r = stringAddstring(num1, num2);

break;

case 2:

r = IntSub(num1, num2);

break;

case 3:

r = IntMult(num1, num2);

break;

}

//对求得的结果进行修剪，去掉最前面的几个0

while ('0' == r[0] && r.size()>1)

{

r = r.substr(1, r.size() - 1);

}

if (op == 3 && flag1 != flag2)//两个数的符号位不同，相乘为负数

{

r.insert(0, "-");

}

cout<<r<<endl;

}

system("pause");

}

## 测试分析

自测数据如下所示：

|  |
| --- |
| 5  89898989898989898989 10101010101010101010 1  342617845187641875431 342617845187641875431 2  4179847814732141471 14715467143551784 2  378137129312738912371237123172931 462317843256872158274151521 3  -427184478321497423 421482142814124 3 |

第一行表示有5组测试样例，根据第三个参数可知其中第一组进行加法，第二组和第三组进行加法，第四组和第五组进行乘法。程序的运行结果如图 2.14所示。

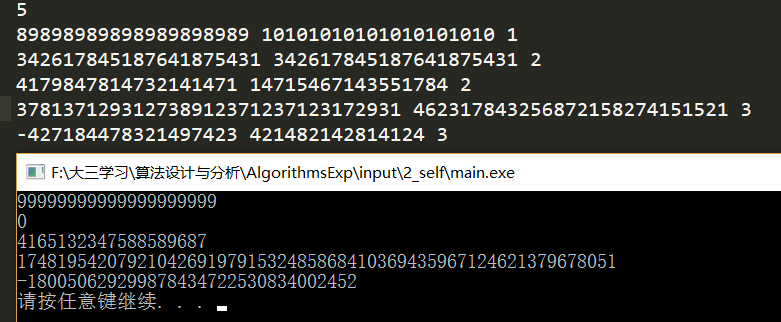


图 2.14 自测数据运行结果

统测数据如下所示：

|  |
| --- |
| 5  -12132134374136478324623 12132134374136478324623 1  0 312321312314324223 1  13467153478115678 143298165941654751756156284 2  -3241412347812341 -125748543715431952 3  1213123129312932183 0 3 |

统测的程序运行结果如图 2.15所示。

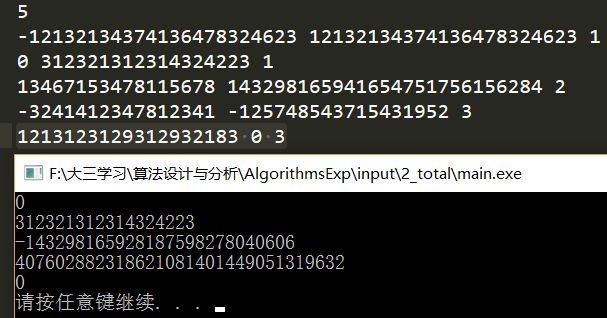


图 2.15统测数据运行结果

将两次测试的结果与期望的结果比较可知，程序运行输出的结果与预期一致，可知程序的运行正确。

## 技术总结

大整数的存储不能简单的使用int型或者long型这些常用的数据类型来存储，考虑到可以把一串大整数看成一个字符串，所以采取了string类型来存储。加法和减法可以之间对字符串的从低位到高位的每一个按位操作，只需要注意加法要减去一个’0’而减法要加上一个’0’而且要注意加法进位和减法借位时的操作。

大整数的乘法主要是依托以下公式进行的：

所以在将两个数据补齐到位后可以采取分治法来把两个乘数进行分解计算。

# 最优二分查找树

## 题目描述

给定一个n个不同关键词的已按升序排好序的序列K=，我们希望用这些关键词构造一棵二叉搜索树。对每个关键词，都有一个概率表示其搜索频率。有些要搜索的值可能不在K中，因此我们还有n+1个“伪关键词”表示不在K中的值。表示所有小于的值，表示所有大于的值，对i=1，2，…，n-1，表示所有在和之间的值，对于每一个伪关键词，也有一个与之对应的搜索概率。对关键词的集合构造一棵最优的二叉搜索树，使得期望搜索代价最小。

输入：最优二叉搜索树的root表。

输出：对于每个关键词，如果是根，则输出“为根”，否则输出此关键词与其父结点的关系，如“为的左（右）孩子”；若为“伪关键词”，则输出其与父结点的关系，如“为的左（右）孩子”。

## 算法设计

因为已经有最优二叉搜索树的root表，所以要建立一棵二叉搜索树只需要在表上找到需要的数据即可。Root表中的数据代表着一段区间的最优划分点，比如root[2][7] = 5代表一个包含2~7号节点的子树的最优二叉搜索树是以5号节点为根节点，2~4号结点为左子树中的节点，6~7号节点为右子树中的节点的一棵二叉树。

首先先读入root表格的数据，采取vector类来进行存储，利用push\_back函数将新的一行存储到尾部，如图 3.1所示。

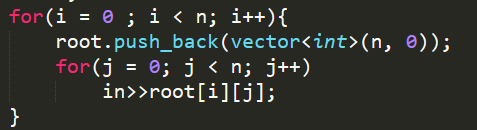


图 3.1 读入root表格数据

按照二叉搜索树的建造策略，最优的二叉搜索树是以root[0][n-1]为根节点，所以首先输出最优二叉搜索树的根节点信息，如图 3.2所示。



图 3.2 输出最优二叉搜索树的根节点信息

假设root[0][n-1]=k，则第k个结点将所有的节点分成了两部分：{0,1，…，k-1}和{k+1,k+2，…，n-1}。分别对左右两部分创建一个最优的二叉搜索树，如图 3.3所示。



图 3.3 对左右两部分的节点递归建树

因为递归创建左最优二叉搜索子树和递归创建右最优二叉搜索子树的方法比较相似，此处以创建左最优二叉搜索子树为例。

如果i < j，说明节点集为空，直接返回即可，否则输出当前节点是上一个节点（函数的第三个参数+1）的左孩子。如果i = j，说明当前的节点集中只有一个节点，则输出此节点的两个叶子节点的信息，如图所示。

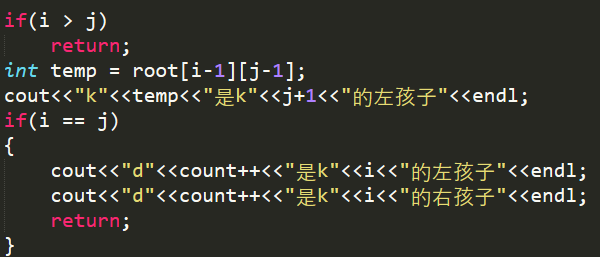


图 3.4 输出当前节点的信息

上述的temp变量是下一个划分点的位置，如果下一个节点的划分位置等于当前的节点集的左边界，说明此节点界的最小序号的节点的左孩子是一个叶子节点，否则就根据temp的值对左半边的子节点集进行进一步的划分；对右边的判断相同，如图 3.5所示。

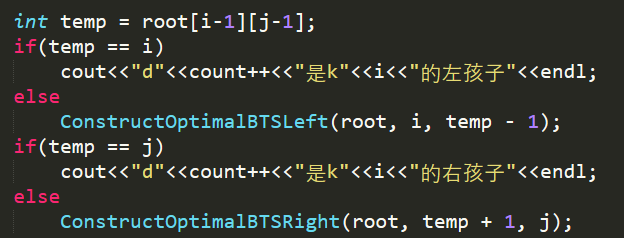


图 3.5 递归建左右子最优搜索子树

创建右最优搜索子树的流程如图所示。

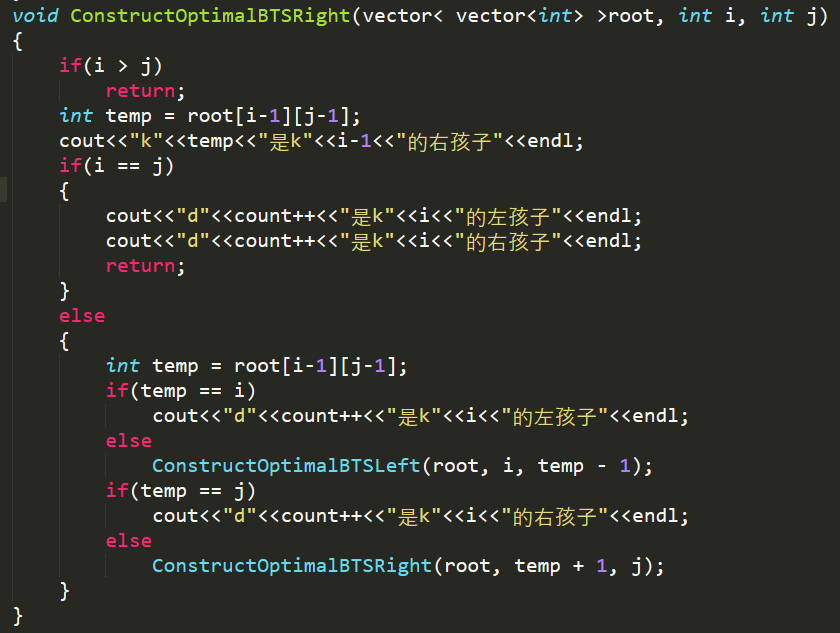


图 3.6 递归创建最优二叉搜索右子树函数体

## 源代码

#include <cstdio>

#include <cstdlib>

#include <vector>

#include <fstream>

#include <iostream>

using namespace std;

void ConstructOptimalBTSLeft(vector< vector<int> >root, int i, int j);

void ConstructOptimalBTSRight(vector< vector<int> >root, int i, int j);

int n = 0;

int count = 0;

int main()

{

int i = 0, j = 0;

ifstream in("in.dat");

if(!in.is\_open())

{

cout<<"fail to open file : in.dat"<<endl;

return -1;

}

vector< vector<int> > root;

in>>n;

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

root.push\_back(vector<int>(n, 0));

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

in>>root[i][j];

}

cout<<"k"<<root[0][n-1]<<"是根"<<endl;

ConstructOptimalBTSLeft(root, 1, root[0][n-1] - 1);

ConstructOptimalBTSRight(root, root[0][n-1] + 1, n);

system("pause");

return 0;

}

void ConstructOptimalBTSLeft(vector< vector<int> >root, int i, int j)

{

if(i > j)

return;

int temp = root[i-1][j-1];

cout<<"k"<<temp<<"是k"<<j+1<<"的左孩子"<<endl;

if(i == j)

{

cout<<"d"<<count++<<"是k"<<i<<"的左孩子"<<endl;

cout<<"d"<<count++<<"是k"<<i<<"的右孩子"<<endl;

return;

}

else

{

int temp = root[i-1][j-1];

if(temp == i)

cout<<"d"<<count++<<"是k"<<i<<"的左孩子"<<endl;

else

ConstructOptimalBTSLeft(root, i, temp - 1);

if(temp == j)

cout<<"d"<<count++<<"是k"<<i<<"的右孩子"<<endl;

else

ConstructOptimalBTSRight(root, temp + 1, j);

}

}

void ConstructOptimalBTSRight(vector< vector<int> >root, int i, int j)

{

if(i > j)

return;

int temp = root[i-1][j-1];

cout<<"k"<<temp<<"是k"<<i-1<<"的右孩子"<<endl;

if(i == j)

{

cout<<"d"<<count++<<"是k"<<i<<"的左孩子"<<endl;

cout<<"d"<<count++<<"是k"<<i<<"的右孩子"<<endl;

return;

}

else

{

int temp = root[i-1][j-1];

if(temp == i)

cout<<"d"<<count++<<"是k"<<i<<"的左孩子"<<endl;

else

ConstructOptimalBTSLeft(root, i, temp - 1);

if(temp == j)

cout<<"d"<<count++<<"是k"<<i<<"的右孩子"<<endl;

else

ConstructOptimalBTSRight(root, temp + 1, j);

}

}

## 测试分析

自测数据输入如下所示：

|  |
| --- |
| 7  1 2 2 2 3 3 5  0 2 3 3 3 5 5  0 0 3 3 4 5 5  0 0 0 4 5 5 6  0 0 0 0 5 6 6  0 0 0 0 0 6 7  0 0 0 0 0 0 7 |

第一行表示有7个结点，后跟七行，为root表的数据。因为root表只占一个大小的一半多，所以对于n=7时有1+2+..+7 = 28个0。根据自测数据生成的最优二叉搜索树如图 3.7所示。

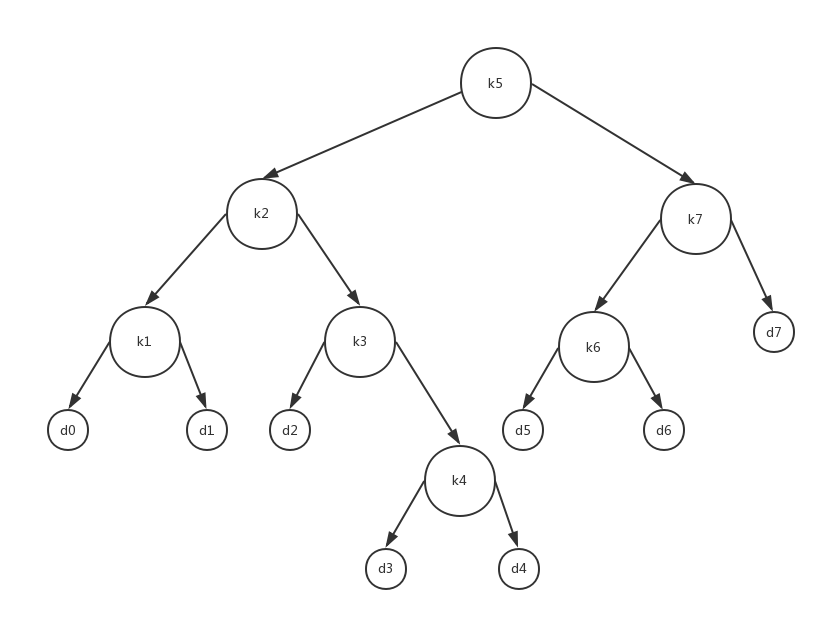


图 3.7 根据自测数据生成的最优二叉搜索树

程序的运行结果如图 3.7所示。

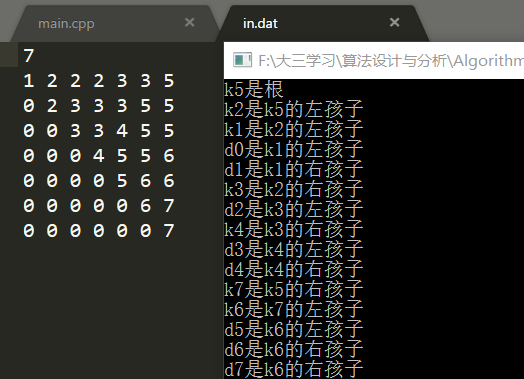


图 3.8 自测数据的程序输出

统测数据如下所示：

|  |
| --- |
| 5  1 1 2 2 2  0 2 2 2 4  0 0 3 4 5  0 0 0 4 5  0 0 0 0 5 |

根据统测数据生成的最优二叉搜索树如图所示。

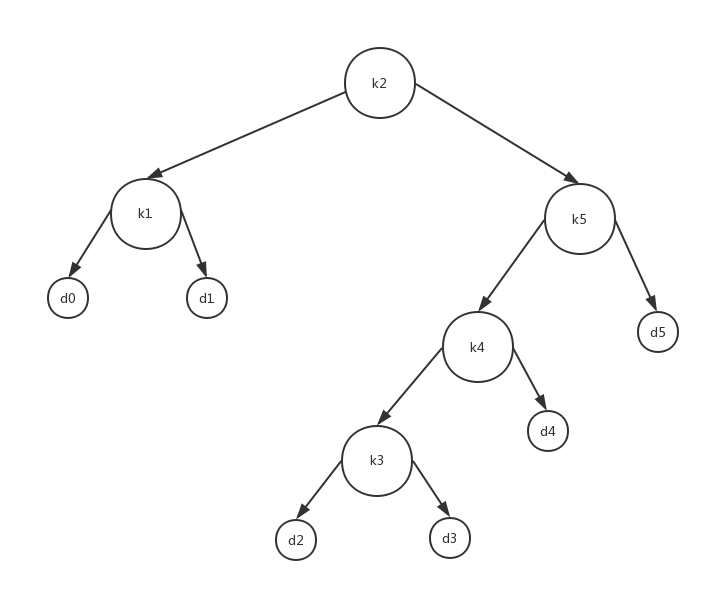


图 3.9 根据统测数据生成的最优二叉搜索树

程序的运行结果如图所示：

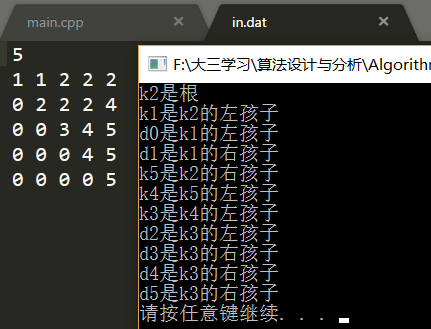


图 3.10 统测数据的程序输出

## 技术总结

最优二叉搜索树是利用动态规划策略自底向上不断更新三个矩阵：最小期望搜索代价表e、搜索概率表w和子树根节点选择表root。因为本题直接给出了创建最优二叉搜索树唯一需要的root表，所以此处不涉及三个表的更新思路。Root表中的数据代表的是一个区间段内的最优划分点的位置，如root[i][j] = k表示对于节点集{i，i+1，…，j-1，j}，最优二叉搜索树是节点k为根节点的一棵二叉搜索树。在选择完根节点后，剩余的节点被分成了两个子节点集：左子节点集{i，i+1，…，k-2，k-1}和右子节点集{k+1，k+2，…，j-1，j}，然后再对左右两个子节点集递归创建两棵子搜索树。

# Floyd Warshell求最短路径

## 题目描述

补充ALL-PATH算法，增加前驱矩阵，是的在求出节点间的最短路径长度矩阵A后，能够推导出每对节点间的最短路径。

输入：第一行为一个整数，表示测试用例的组数，其后跟相应组数的测试用例，每组用例包括：

·首行：一个整数，表示本组测试用例包含的结点数n，其后跟n行

·其后：每行n个整数，表示结点间邻接关系及边的长度（邻接成本矩阵），边的长度小于100,100即表示结点间没有边

输出：第一行为一个整数，表示测试用例的组数，其后跟相应组数的测试用例，每组测试用例包括：

·首行：一个整数，表示本组测试用例包含的结点数n，其后跟行

·其后：开始的n行，每行n个整数，表示结点间最短路径的长度（A矩阵），路径的程度＜32767,32767即表示结点间没有可达的路径。其后行，顺次输出结点对（1,1）、（1,2）、…、（1，n），（2,1）、（2,2）、…、（2,n）、…（n,1）、（n,n）之间的最短路径结点序列，结点间用空格隔开。（i，i）输出i，若（i，j）之间没有路径，输出NULL。

## 算法设计

首先写构造两个大小为个整形数据的数组d和p，分别用来存储Floyd-Warshall算法中的两个数组。接着对两个数组赋初值：对于d数组，逐个读取输入中的数据输入即可，但是需要注意的是，在输入中以100代表两点之间没有相连的边，为了避免这个100的值与计算过程中可能出现的100出现冲突，所以将100替换成了路径长度的上限32767；对于p数组中的元素p[i][j] = j。两个数组的初始化如图 4.1所示。

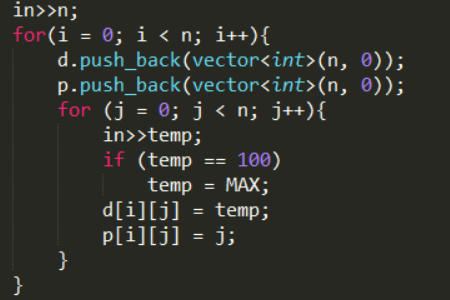


图 4.1 数组d和数组p的初始化

数组d和p使用的是vector库封装的数据结构vector来存储数据的，具体的使用方法此处不再赘述。通过push\_back函数将函数的参数加入到vecotr容器的尾部，此处插入的是另一个包括n个整数，初值为0的vector容器。在循环结束后，得到的是一个容纳n个vector容器的vector容器d，它所容纳的每个vector容器中存储了n个整型数据。

在数据初始化后，对所有节点进行遍历，在遍历的过程中，以当前遍历到的节点i作为中间结点，判断d（u，v）是否大于d（u，i）+ d（i，v），如果小于的话说明从u结点去往v结点经过i结点的中转要比直接去的路程要短。此时将点u和点v之间的距离d[u][v]修改为d[u][i] + d[i][v]的值，并将p[u][v]修改为p[u][i]，注意，从u结点经由i点去往v结点不代表直接将p[u][v]修改成i，而是要修改成p[u][i]，这是因为经由i点中转前往v结点不代表直接前往i点，可能还会经过其他结点的中转。循环更新两个矩阵的流程如图 4.2所示。

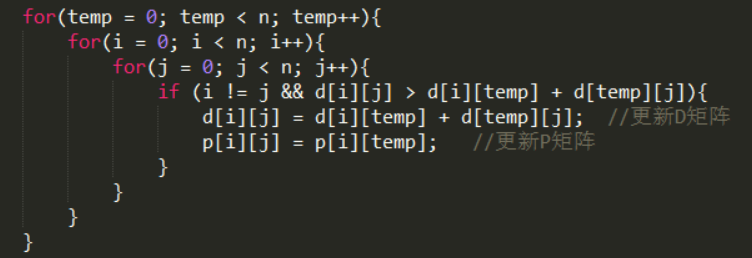


图 4.2 Floyd-Warshall算法更新两个矩阵

输出运算结果时首先输出运算结束后的d矩阵，输出方式很简单，此处不过多描述，如图 4.3所示。

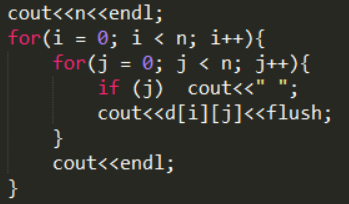


图 4.3 d矩阵的输出

输出完d矩阵后，分别输出两两结点之间的最短路径，其中，如果起始结点和目标结点为同一个结点，只需输出对应结点的编号。获取最短路径的方法是根据p矩阵一步一步跳，直到到达目标节点，获取最短路径的实现如图 4.4所示。

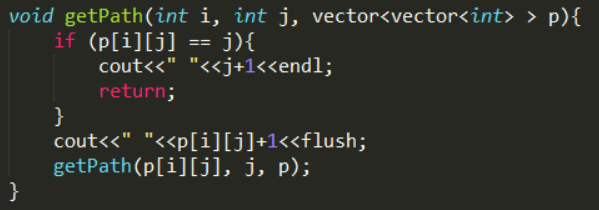


图 4.4 获取最短路径函数getPath的函数体

在获取完路径之后，将路径输出即可，如图 4.5所示。在结果输出后，为了方便下一次循环中继续使用d矩阵和p矩阵来存储中间的状态及最终的结果，要对此次测试用例中的数据进行清空操作。Vector对象的清空操作直接调用对象的clear函数即可，如图 4.5的最后两行。如果没有清空这一步，会导致每次访问数组元素时都访问的是第一次测试用例中的数据，导致程序运算出错误的结果。

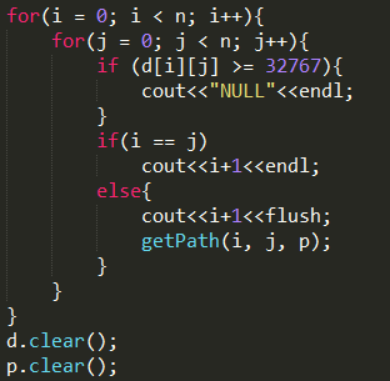


图 4.5 结果的输出及矩阵的清空

## 源代码

#include <iostream>

#include <fstream>

#include <vector>

#include <cstdlib>

#define MAX 32767

using namespace std;

void getPath(int i, int j, vector<vector<int> > p){

if (p[i][j] == j){

cout<<" "<<j+1<<endl;

return;

}

cout<<" "<<p[i][j]+1<<flush;

getPath(p[i][j], j, p);

}

int main(){

int t = 0; //测试样例个数

int n = 0; //测试样例结点个数

int i = 0, j = 0, temp = 0;

vector< vector<int> > d;

vector< vector<int> > p;

ifstream in("in.dat");

if (!in.is\_open()){

cout<<"fail to open file : in.dat"<<endl;

return -1;

}

in>>t;

cout<<t;

while(t--){

in>>n;

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

d.push\_back(vector<int>(n, 0));

p.push\_back(vector<int>(n, 0));

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

in>>temp;

if (temp == 100)

temp = MAX;

d[i][j] = temp;

p[i][j] = j;

}

}

for(temp = 0; temp < n; temp++){

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

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

if (i != j && d[i][j] > d[i][temp] + d[temp][j]){

d[i][j] = d[i][temp] + d[temp][j]; //更新D矩阵

p[i][j] = p[i][temp]; //更新P矩阵

}

}

}

}

cout<<n<<endl;

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

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

if (j) cout<<" ";

cout<<d[i][j]<<flush;

}

cout<<endl;

}

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

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

if (d[i][j] >= 32767){

cout<<"NULL"<<endl;

}

if(i == j)

cout<<i+1<<endl;

else{

cout<<i+1<<flush;

getPath(i, j, p);

}

}

}

d.clear();

p.clear();

}

system("pause");

return 0;

}

## 测试分析

自测数据输入如下：

|  |
| --- |
| 1  9   1. 1 5 100 100 100 100 100 100 100 2. 0 3 7 5 100 100 100 100   5 3 0 100 1 7 100 100 100  100 7 100 0 2 100 3 100 100  100 5 1 2 0 3 6 9 100  100 100 7 100 3 0 100 5 100  100 100 100 3 6 100 0 2 7  100 100 100 100 9 5 2 0 4  100 100 100 100 100 100 7 4 0 |

其中第一行表示有1组测试用例，第二行表示这一组测试用例中有9个结点，后跟9行，分别为9个结点到其他结点的距离，其中的100表示两个结点没有直接相连的边。

由于程序运行输出的结果较长，此处分几部分展示。程序运行结束后的距离矩阵如图 4.6所示。

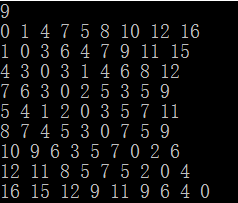


图 4.6 自测数据下各结点之间的最短路径长度

第二行的9个数据分别表示第一个结点到九个结点的最短路径的长度，其中，第一个结点到第一个结点的路径长度为0。将各结点间的路径长度自测数据的标准输出对比可知，程序运行结果正确。

在最短路径长度的矩阵后有9×9行，每九行代表一个结点去往另一个结点的最短路径上经过的结点编号，以第一个结点的输出为例，如图 4.7所示。

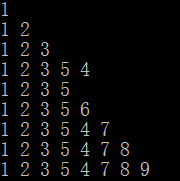


图 4.7 第一个结点到各个结点的路径

与标准输出对比可知程序运行结果正确，其他八个结点的输出相似，此处不一一列举。

统测数据输入如下所示：

|  |
| --- |
| 1  4  0 6 100 3  5 0 1 100  3 100 0 2  8 2 100 0 |

统测数据的输出结果如图所示， 与预期的输出结果一致。

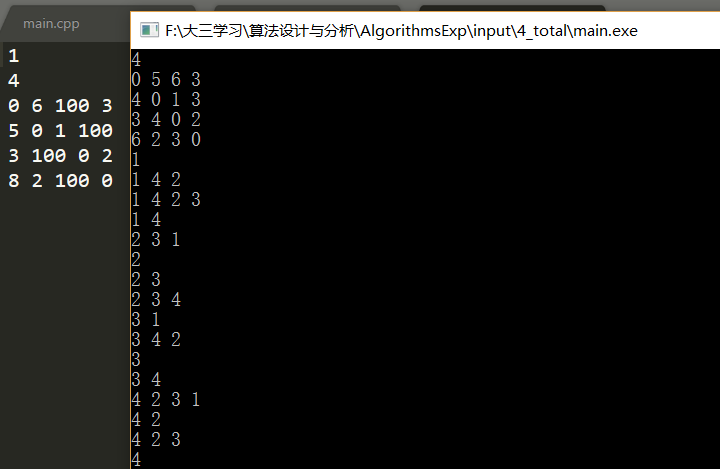


图 4.8 统测数据输出结果

## 技术总结

本题的主要思路是按照Floyd-Warshall算法的思想对两个矩阵进行维护与更新。Floyd-Warshall算法考虑的是从一条最短路径上的中间结点出发，利用结点i与结点j之间权重最小的简单路径p和从i到j之间的中间节点均取自集合{1,2，…，k-1}的最短路径之间的关系。主要基于图的松弛思想对边进行松弛：

在计算最短路径权重矩阵D的同时计算前驱矩阵P，定义p[i][j]为从结点i到结点j的一条所用中间节点都取自集合{1,2，…，k-1}的最短路径上的j的前驱结点。

# Wooden Sticks ( POJ 1065 )

## 题目描述

·Description

|  |
| --- |
| here is a pile of n wooden sticks. The length and weight of each stick are known in advance. The sticks are to be processed by a woodworking machine in one by one fashion. It needs some time, called setup time, for the machine to prepare processing a stick. The setup times are associated with cleaning operations and changing tools and shapes in the machine. The setup times of the woodworking machine are given as follows:  (a) The setup time for the first wooden stick is 1 minute.  (b) Right after processing a stick of length l and weight w , the machine will need no setup time for a stick of length l' and weight w' if l <= l' and w <= w'. Otherwise, it will need 1 minute for setup.  You are to find the minimum setup time to process a given pile of n wooden sticks. For example, if you have five sticks whose pairs of length and weight are ( 9 , 4 ) , ( 2 , 5 ) , ( 1 , 2 ) , ( 5 , 3 ) , and ( 4 , 1 ) , then the minimum setup time should be 2 minutes since there is a sequence of pairs ( 4 , 1 ) , ( 5 , 3 ) , ( 9 , 4 ) , ( 1 , 2 ) , ( 2 , 5 ) . |

·Input

|  |
| --- |
| The input consists of T test cases. The number of test cases (T) is given in the first line of the input file. Each test case consists of two lines: The first line has an integer n , 1 <= n <= 5000 , that represents the number of wooden sticks in the test case, and the second line contains 2n positive integers l1 , w1 , l2 , w2 ,..., ln , wn , each of magnitude at most 10000 , where li and wi are the length and weight of the i th wooden stick, respectively. The 2n integers are delimited by one or more spaces. |

·Output

|  |
| --- |
| The output should contain the minimum setup time in minutes, one per line. |

·Sample Input

|  |
| --- |
| 3  5  4 9 5 2 2 1 3 5 1 4  3  2 2 1 1 2 2  3  1 3 2 2 3 1 |

·Sample Output

|  |
| --- |
| 2  1  3 |

## 算法设计

这个题目的思路有点类似于俄罗斯套娃：如果一个木棍的长度和重量比之前的某根木棍的长度和重量都要大，那么这两个木棍的加工只需要花费一个单位的时间。

首先，定义一个用来存储木棍属性的结构my\_Stick，这个结构中包括两个数据元素：木棍的长度和重量。结构定义如图 5.1所示。

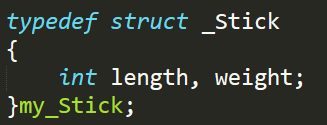


图 5.1 木棍属性结构定义

因为题目说明了测试用例的木棍个数是在[1，5000]范围内，所以定义一个以my\_Stick结构为基本元素的数组s用来存储木棍的属性，数组的大小为5001个my\_Stick结构的大小，再定义一个大小为5001个整型数据的数组flag用来表示每一根木棍是否已被加工。首先读入木棍的信息存入到数组s中，如图 5.2所示。

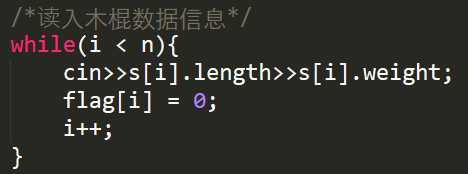


图 5.2 读入木棍数据

在读入数据后，首先对木棍进行排序，排序的策略是按照木棍的长度降序排列，如果两根木棍的长度一致，则按照木棍的重量降序排列。在C++中有一个封装好的sort函数，只需要自定义比较的策略即可，调用sort函数对木棍数组进行降序排列如图 5.3所示，其中的n为木棍的个数，cmp未定义的函数。



图 5.3 木棍降序排列

自定义的cmp函数的返回值是一个bool类型，在函数体中对两个木棍的长度和重量进行比较，函数体的实现如图 5.4所示。

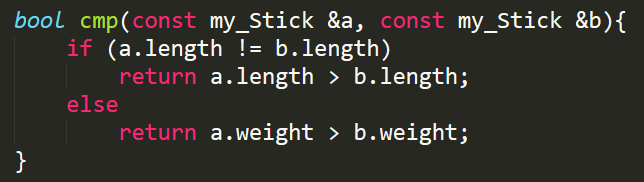


图 5.4 自定义cmp函数的函数体

在木棍降序排列完成后，采用贪心策略来对木棍进行加工。对木棍从第一根到最后一根进行枚举，如果这一根木棍已被加工过，则枚举下一根；如果没有加工过，则依次遍历其后的所有未加工的木棍，因为木棍是按照长度的降序排列的，所以后续的木棍的长度肯定不会大于此时的木棍，所以只需判断后续的木棍的重量是否小于此时的木棍的重量，如果小于则说明该木棍可以与此木棍一同加工，将该木棍的flag置为1，同时更新此次加工的最小重量，并继续寻找下一根可以一并加工的木棍。贪心策略的循环流程图 5.5所示。

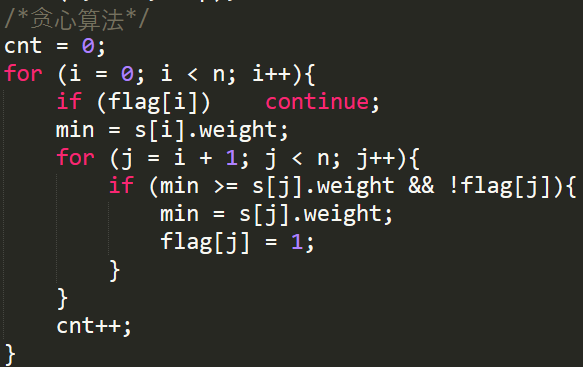


图 5.5 贪心策略找到最小加工次数

## 源代码

#include<iostream>

#include<algorithm>

using namespace std;

typedef struct \_Stick

{

int length, weight;

}my\_Stick;

bool cmp(const my\_Stick &a, const my\_Stick &b){

if (a.length != b.length)

return a.length > b.length;

else

return a.weight > b.weight;

}

int main(){

/\*测试样例组数\*/

my\_Stick s[5001];

int flag[5001];

int t = 0, i = 0, j = 0, cnt = 1, min = 0;

cin>>t;

while(t--){

int n = 0; /\*木棍个数\*/

cin>>n;

i = 0;

/\*读入木棍数据信息\*/

while(i < n){

cin>>s[i].length>>s[i].weight;

flag[i] = 0;

i++;

}

/\*对木棍数据进行降序排序\*/

sort(s, s+n, cmp);

/\*贪心算法\*/

cnt = 0;

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

if (flag[i]) continue;

min = s[i].weight;

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

if (min >= s[j].weight && !flag[j]){

min = s[j].weight;

flag[j] = 1;

}

}

cnt++;

}

cout<<cnt<<endl;

}

return 0;

}

## 测试分析

样例输入如下所示：

|  |
| --- |
| 3  5  4 9 5 2 2 1 3 5 1 4  3  2 2 1 1 2 2  3  1 3 2 2 3 1 |

其中第一行表示有三组测试用例，第二行、第四行、第六行分别表示对应测试用例中的木棍个数n，其后跟一行，包含2n个整数，分别是第一根到第n跟木棍的长度和重量的值。

对第一个测试用例中的五根木棍按照长度和重量的降序排列后可以得到以下顺序：

（5,2）->（4,9）->（3,5）->（2,1）->（1,4）

在第一根木棍（5,2）加工完后，检查后续的木棍发现第四根木棍（2,1）可以一并加工，所以第一个加工的单位时间内加工了两根木棍；在加工第二根木棍（4,9）后检查后续的木棍可知，（3,5）可以一并加工，此时将（3,5）对应的flag置为1，同时将此次加工的最小重量更新为5（更新前为9），继续检查后续木棍发现（1,4）可以一并加工，所以在第二个加工的单位时间内加工了三根木棍。至此，所有的五根木棍均已加工完成，总共花费了两个单位的加工时间。按照同样的思路可以得知第二组测试用例所需要的加工时间为1个单位时间，第三组测试用例所需要的加工时间为3个单位时间。

程序执行样例输入的运行结果如图 5.6所示，可以得知，运行结果与实际结果一致。

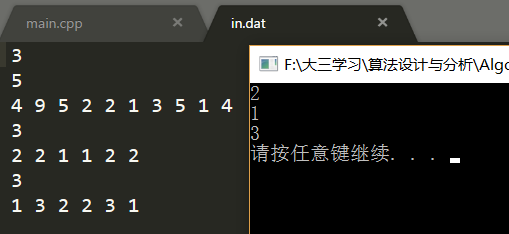


图 5.6 样例输入测试结果

提交到系统中进行黑箱测试可以得到测试结果如图 5.7所示。可以看出，程序运行结果符合题目要求。

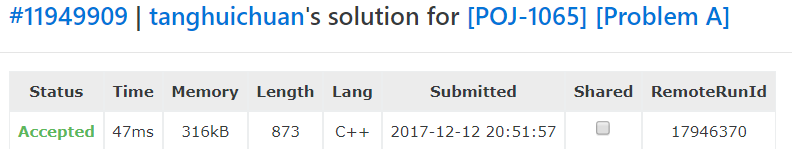


图 5.7 POJ-1065黑箱测试结果图

## 技术总结

本题的主要思想是按照两个关键词排序以及贪心策略求最小加工时间。

对木棍按照关键词排序是直接调用的C++封装好的sort排序函数，以木棍的长度作为第一关键词，重量作为第二关键词来降序排列。

贪心策略用在计算最小的加工时间处，每次加工一根木棍后，按照贪心策略的原则找到剩余的木棍中可以一并加工的长度最大的木棍（长度相同则选择重量相同）这样，可以在单位加工时间内尽可能多的加工最多的木棍。

# Gone fishing ( POJ 1042 )

## 题目描述

·Description

|  |
| --- |
| John is going on a fishing trip. He has h hours available (1 <= h <= 16), and there are n lakes in the area (2 <= n <= 25) all reachable along a single, one-way road. John starts at lake 1, but he can finish at any lake he wants. He can only travel from one lake to the next one, but he does not have to stop at any lake unless he wishes to. For each i = 1,...,n - 1, the number of 5-minute intervals it takes to travel from lake i to lake i + 1 is denoted ti (0 < ti <=192). For example, t3 = 4 means that it takes 20 minutes to travel from lake 3 to lake 4. To help plan his fishing trip, John has gathered some information about the lakes. For each lake i, the number of fish expected to be caught in the initial 5 minutes, denoted fi( fi >= 0 ), is known. Each 5 minutes of fishing decreases the number of fish expected to be caught in the next 5-minute interval by a constant rate of di (di >= 0). If the number of fish expected to be caught in an interval is less than or equal to di , there will be no more fish left in the lake in the next interval. To simplify the planning, John assumes that no one else will be fishing at the lakes to affect the number of fish he expects to catch.  Write a program to help John plan his fishing trip to maximize the number of fish expected to be caught. The number of minutes spent at each lake must be a multiple of 5. |

·Input

|  |
| --- |
| You will be given a number of cases in the input. Each case starts with a line containing n. This is followed by a line containing h. Next, there is a line of n integers specifying fi (1 <= i <=n), then a line of n integers di (1 <=i <=n), and finally, a line of n - 1 integers ti (1 <=i <=n - 1). Input is terminated by a case in which n = 0. |

·Output

|  |
| --- |
| For each test case, print the number of minutes spent at each lake, separated by commas, for the plan achieving the maximum number of fish expected to be caught (you should print the entire plan on one line even if it exceeds 80 characters). This is followed by a line containing the number of fish expected.  If multiple plans exist, choose the one that spends as long as possible at lake 1, even if no fish are expected to be caught in some intervals. If there is still a tie, choose the one that spends as long as possible at lake 2, and so on. Insert a blank line between cases. |

·Sample Input

|  |
| --- |
| 2  1  10 1  2 5  2  4  4  10 15 20 17  0 3 4 3  1 2 3  4  4  10 15 50 30  0 3 4 3  1 2 3  0 |

·Sample Output

|  |
| --- |
| 45， 5  Number of fish expected: 31  240，0，0，0  Number of fish expected: 480  115，10，50，35  Number of fish expected: 724 |

## 算法设计

根据题意，因为John只能从编号较小的湖走向编号较大的湖，因为返回的话只会增加John在湖之间走花费的时间，从而减少了他钓鱼过程的时间。对John钓鱼过程中最后一个钓鱼的湖（即所有钓过鱼的湖中编号最大的那一个）进行枚举，假如John最后是在第i个湖钓鱼，那么需要先减去从最开始的湖泊到第i个湖之间路上所花费的时间，剩下的时间即为John所拥有的花费在钓鱼上的时间。因为从第一个湖泊到第i个湖泊之间的路程上所花费的时间已经捡去了，所以可以看做这i个湖之间是没有距离的，也就是可以不花费任何代价地从这i个湖泊中选择一个当前鱼量最多的湖进行钓鱼。

首先使用整型数组来分别储存每个鱼塘中的初始鱼量fi[i]、每钓鱼五分钟后对应鱼塘减少的鱼量di[i]和从第i个湖去第i+1个湖路上所花费的时间ti[i]。因为要对每种情况进行枚举，在每次枚举的过程中会对鱼塘的鱼量进行修改，所以额外增加一个缓存数组fi\_temp[i]，每次枚举之前将fi[i]的值复制到fi\_temp[i]中，在枚举的过程中只对fi\_temp数组中的数据进行操作，避免了修改fi中的值，对后续的操作造成影响。本次实验中定义的变量如图 6.1所示。

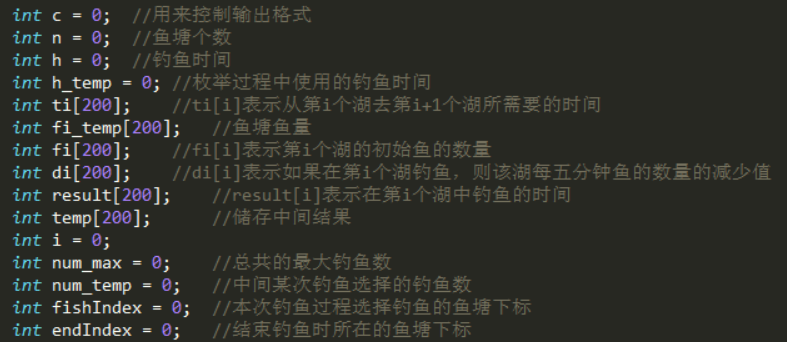


图 6.1 程序变量的声明及对应含义

首先读入湖的个数n并判断是否满足终止条件：n = 0。如果n大于0，则开始进行本轮的计算。利用cin读入各项数据，如图 6.2所示。其中需要注意的是，输入的时间h是以小时为单位，而后面的钓鱼和步行的时间都是以五分钟为单位，所以通过将h乘上12来将小时转换成以五分钟为单位的数，便于后续的计算。

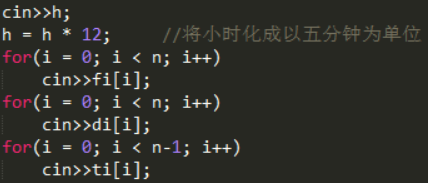


图 6.2 读入本轮计算参数示意图

以结束钓鱼的鱼塘下标endIndex作为枚举的参数，对每一种情况进行枚举。根据之前的分析，因为不能直接在fi数组中对鱼塘的鱼量进行操作，所以在每一次枚举的过程的最开始，要将fi数组的值复制到fi\_temp数组中，在枚举的过程中对fi\_temp进行操作，同时还要将钓鱼的总时间和存储在每个湖钓鱼的时间的数组重置，对应的操作如图 6.3所示。

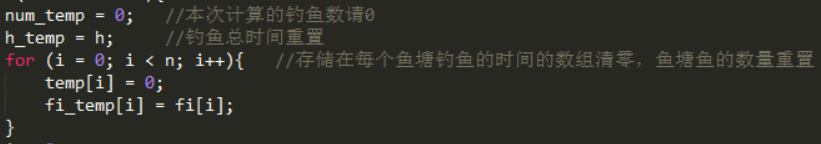


图 6.3 枚举过程中变量的初始化

在对本次枚举过程中所需要用到的变量初始化后，首先减去此次枚举过程中在路上花费的时间。因为只涉及到对ti数组各个数的读取，而不涉及修改ti数组中某些项的值，所以可以直接使用ti数组，而不用重新洗定义一个ti\_temp数组。本次枚举的终点是第endIndex个湖，所以利用一个循环减去从第一个湖到第endIndex个湖中间花费的时间，如图 6.4所示。



图 6.4 减去路上步行花费的时间

减去步行花费的时间后，剩下的时间即为钓鱼的时间，当钓鱼的时间大于0时，表示还有时间可以钓鱼，此时遍历第一个湖到第endIndex个湖中所有的湖的鱼量，选中到一个鱼量最大的湖，在其中钓鱼，如图 6.5所示。

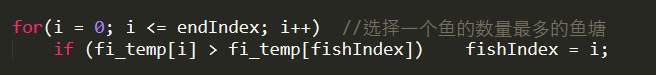


图 6.5 贪心算法选择一个鱼量最多的湖

如果选中的湖的鱼量为0，表示第一个湖到第endIndex个湖中都没有鱼了，此时将剩余的时间加到第一个湖中钓鱼的时间中，并退出循环，如图 6.6所示。

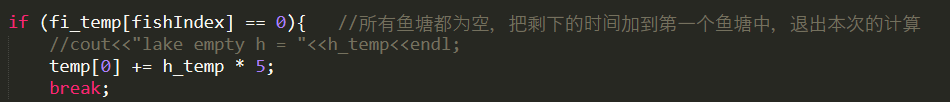


图 6.6 所有的湖都没有鱼时退出循环

如果选中的湖中还有鱼，则在选中的湖中钓鱼。将在选中的湖钓到的鱼量加入到本次钓鱼过程的总钓鱼量中，接着用选中湖的鱼量fi\_temp[fishIndex]减去选中的湖每钓鱼依次后减少的鱼量di[fishIndex]。同样因为对di数组只涉及到读，而不需要修改其中的数据，所以不需要额外分配一个缓存数组。需要注意的是，如果一个湖的鱼量在做完减法后小于0，应该将这个湖的鱼量修正为0。在对鱼量进行完修改之后要对时间进行修改，因为每一次钓鱼的时长是五分钟，所以要将temp[fishIndex]加上5分钟，同时要将钓鱼的时间h\_temp减去1（这个1 表示的是一个五分钟）。如图 6.7所示。

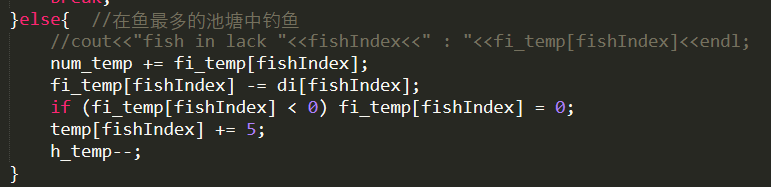


图 6.7 在选中的湖中钓鱼

到这里，本次枚举的最大钓鱼量已经计算出来了，将本次的钓鱼量和所以已经枚举过的情况中的最大钓鱼量进行对比，如果本次的钓鱼量更大，则将本次钓鱼过程的鱼量和在各个湖钓鱼的时长存入到存放结果的变量中，如图 6.8所示。

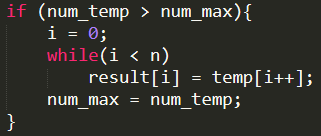


图 6.8 与之前的最大钓鱼量进行比较

比较完后将endIndex加一，开始计算在下一个湖终止的情况下的最大钓鱼量，直到枚举完在最后一个湖结束钓鱼，此时存放在num\_max中的即为所有情况中的最大钓鱼量，在result数组中的数据即为在得到最大钓鱼量的情况下在各个湖中钓鱼的时间，将其输出即可，如图 6.9所示。

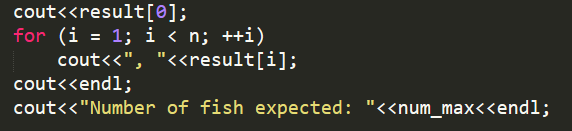


图 6.9 输出本次测试用例结果

## 源代码

#include<iostream>

using namespace std;

int main(int argc, char const \*argv[])

{

int c = 0; //用来控制输出格式

int n = 0; //鱼塘个数

int h = 0; //钓鱼时间

int h\_temp = 0; //枚举过程中使用的钓鱼时间

int ti[200]; //ti[i]表示从第i个湖去第i+1个湖所需要的时间

int fi\_temp[200]; //鱼塘鱼量

int fi[200]; //fi[i]表示第i个湖的初始鱼的数量

int di[200]; //di[i]表示如果在第i个湖钓鱼，则该湖每五分钟鱼的数量的减少值

int result[200]; //result[i]表示在第i个湖中钓鱼的时间

int temp[200]; //储存中间结果

int i = 0;

int num\_max = 0; //总共的最大钓鱼数

int num\_temp = 0; //中间某次钓鱼选择的钓鱼数

int fishIndex = 0; //本次钓鱼过程选择钓鱼的鱼塘下标

int endIndex = 0; //结束钓鱼时所在的鱼塘下标

while(i < 200){

ti[i] = 0;

fi[i] = 0;

di[i] = 0;

result[i] = 0;

temp[i++] = 0;

}

while(cin>>n && n > 0){

if (c != 0)

cout<<endl;

c = 1;

cin>>h;

h = h \* 12; //将小时化成以五分钟为单位

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

cin>>fi[i];

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

cin>>di[i];

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

cin>>ti[i];

i = 0;

endIndex = 0;

num\_max = -1;

while(endIndex < n){

num\_temp = 0; //本次计算的钓鱼数请0

h\_temp = h; //钓鱼总时间重置

for (i = 0; i < n; i++){ //存储在每个鱼塘钓鱼的时间的数组清零，鱼塘鱼的数量重置

temp[i] = 0;

fi\_temp[i] = fi[i];

}

i = 0;

while(i < endIndex) h\_temp -= ti[i++]; //先减去在路上花费的时间

while(h\_temp > 0){

fishIndex = 0;

for(i = 0; i <= endIndex; i++) //选择一个鱼的数量最多的鱼塘

if (fi\_temp[i] > fi\_temp[fishIndex]) fishIndex = i;

if (fi\_temp[fishIndex] == 0){ //所有鱼塘都为空，把剩下的时间加到第一个鱼塘中，退出本次的计算

//cout<<"lake empty h = "<<h\_temp<<endl;

temp[0] += h\_temp \* 5;

break;

}else{ //在鱼最多的池塘中钓鱼

//cout<<"fish in lack "<<fishIndex<<" : "<<fi\_temp[fishIndex]<<endl;

num\_temp += fi\_temp[fishIndex];

fi\_temp[fishIndex] -= di[fishIndex];

if (fi\_temp[fishIndex] < 0) fi\_temp[fishIndex] = 0;

temp[fishIndex] += 5;

h\_temp--;

}

}

if (num\_temp > num\_max){

i = 0;

while(i < n)

result[i] = temp[i++];

num\_max = num\_temp;

}

endIndex++;

}

cout<<result[0];

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

cout<<", "<<result[i];

cout<<endl;

cout<<"Number of fish expected: "<<num\_max<<endl;

}

return 0;

}

## 测试分析

样例输入如下所示：

|  |
| --- |
| 2  1  10 1  2 5  2  4  4  10 15 20 17  0 3 4 3  1 2 3  4  4  10 15 50 30  0 3 4 3  1 2 3  0 |

第一行表示第一个测试有两个湖，第二行表示John总共有1个小时的时间，第三行是两个湖的初始鱼量，分别为10和1，第四行是两个湖钓鱼五分钟后减少的鱼量，第五行是从第一个湖到第二个湖步行所需要的时间，2表示需要两个五分钟，即10分钟的步行时间。后续的数据类似，此处不再赘述。最后一行表示本次测试用例没有湖，这是测试结束的标志。

程序的运行结果如图 6.10所示。



图 6.10 样例输入测试结果

与样例输出对比可知，程序运行的结果正确。提交到系统中进行黑箱测试可以得到测试结果如图 6.11所示。可以看出，程序运行结果符合题目要求。

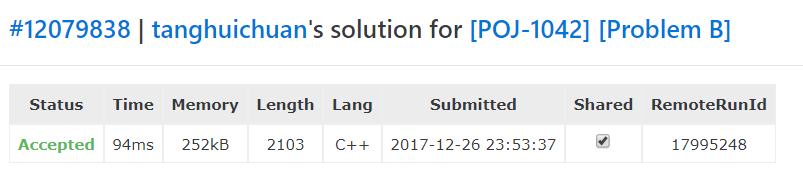


图 6.11 POJ-1042黑箱测试结果图

## 技术总结

本题的主要思想是枚举加贪心算法，难度并不大。首先对结束钓鱼的湖的位置进行枚举，从第一个湖开始一直到最后一个湖。每次先将在湖之间步行花费的时间减去，剩下的时间就是John所拥有的能用在钓鱼上的时间。接着根据贪心算法的原则，每次有时间剩余的时候都选择当前鱼量最大的湖处钓鱼，直到时间耗尽。在所有的枚举情况中选择钓鱼量最多的情况，即为本次测试用例的最优解。

在执行算法的过程中需要注意以下几点：

·如果本次测试用例的解不唯一，即有多个枚举情况中钓鱼数都为最大值，则选择在第一个湖耗时最多的解；如果解仍然不唯一，则选择在第二个湖耗时最多的解，以此类推。

·如果某次在某个池塘钓鱼结束后存在鱼量减去减少量后鱼量为负的情况，则需要将该池塘的鱼量修改为0，而不应该保留为负数。

·在测试数据中可能存在所以的湖中的鱼数都为0的情况。

# Corn Field (POJ 3254 )

## 题目描述

·Description

|  |
| --- |
| Farmer John has purchased a lush new rectangular pasture composed of *M* by *N* (1 ≤ *M* ≤ 12; 1 ≤ *N*≤ 12) square parcels. He wants to grow some yummy corn for the cows on a number of squares. Regrettably, some of the squares are infertile and can't be planted. Canny FJ knows that the cows dislike eating close to each other, so when choosing which squares to plant, he avoids choosing squares that are adjacent; no two chosen squares share an edge. He has not yet made the final choice as to which squares to plant.  Being a very open-minded man, Farmer John wants to consider all possible options for how to choose the squares for planting. He is so open-minded that he considers choosing no squares as a valid option! Please help Farmer John determine the number of ways he can choose the squares to plant. |

·Input

|  |
| --- |
| Line 1: Two space-separated integers：M and N  Lines 2..M+1: Line i+1 describes row I of the pasture with N space-separated integers indicating whether a square is fertile ( 1 for fertile, 0 for infertile) |

·Output

|  |
| --- |
| Line 1: One integer：the number of ways that FJ can choose the squares modulo 100,000,000. |

·Sample Input

|  |
| --- |
| 2 3   1. 1 1   0 1 0 |

·Sample Output

|  |
| --- |
| 9 |

·Hint

|  |
| --- |
| Number the squares as follows：  1 2 3  4  There are four ways to plant only one squares ( 1, 2, 3 or 4), three ways to plant two squares ( 13, 14 or 34), 1 way to plant on three squares (134), and one way to plant on no squares. So the result is : 4+3+1+1 = 9. |

## 算法设计

对于每一行的N块地，在不考虑地是否可以种植的情况下，即假设这N块地为全1的情况下枚举所有可以在这N块地上可以放牛的情况。为了存储放牛的状态，可以将这一行的放牛状态压缩到一个int型的变量中（题目中限定了N的范围是[1，12]，所以一个int 型的变量可以容纳这最多12位数）。比如说，一种放法是每一块地上都不放牛，则可以用12个0来表示，即存入一个0。

对于每一行的N块地，上述的变量的范围是[0，]。而根据题目要求，需要满足相邻的两块地不能同时放牛，这一条件可以用位运算来判断：如果两块相邻的地上同时放了牛，那么将这个状态对应的数左移（或右移均可，此处采取左移的策略）一位后与原数做按位与操作。因为存在相邻位的1，所以得到的结果会不为全零，这就表示在这种放牛的策略下，同一行中有两块相邻的地同时放牛了。反之，如果按位与运算的结果为零，表示相邻的两块地是没有同时放牛的，即满足题目的要求，此时将这种放牛的状态存放到一个用来存放所以可能的状态的数组v中，再继续寻找下一个可行的状态。算法的实现如图 7.1所示。

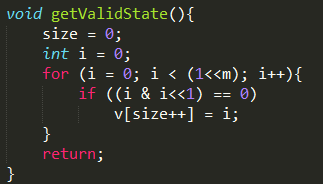


图 7.1 寻找可行状态

在输入每一行的地的状态时，同样采用状态压缩的方式对每一行的状态进行存储，比如某一行的4块地的状态分别为：1 0 1 0，则可以用一个十进制的整型数10来存储这一行的地的状态。因为高位是在低位之前输入，所以要对每一行存储到用来存储每行地的状态的数组map中的数据进行一定量的移位操作。存储每一行地的状态的算法过程如图 7.2所示。

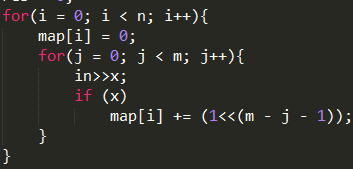


图 7.2 地的状态的压缩存储

在数组v中存储着N块全1的地中有效的放牛状态，但是没有考虑地的实际情况，因此还要对每一个状态判断是否能在该行的地的地上放。一块地上不能按某种放牛状态来放牛当且仅当该状态在一块不可用的地上放牛，更形式化的描述是：放牛状态中的1放在了土地的0上。将当前行地的状态取反后与放牛状态按位与操作后，如果结果不为0，则表示此状态不能用在该行的土地情况上。此判断过程的算法实现如图 7.3所示。

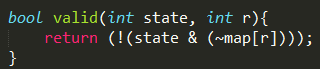


图 7.3 判断某种放牛状态能否应用到某一行中

除了考虑每一行的放牛的地不能相邻之外，还要考虑行与行之间不能相邻，为了简化问题的解决，只考虑除第一行外的M-1行与上一行是否冲突。如果两行冲突，则两行的放牛状态做按位与运算后的结果为非零。定义一个二维数组dp[N][max]，dp[i][j]表示在土地的第i行放入可行状态数组v中的状态v[j]时的可能行情况的总数。对于第一行，如果状态v[j]能够放在该行中，则置dp[0][j]为1，否则置为0。对于第i行（i>1）状态为v[j]时，如果满足能够放在该行的条件，则遍历上一行所有的状态，如果上一行的状态v[x]不与v[j]冲突，dp[i][j]的值加上dp[i-1][x]的值。计算状态个数的流程如图 7.4所示。

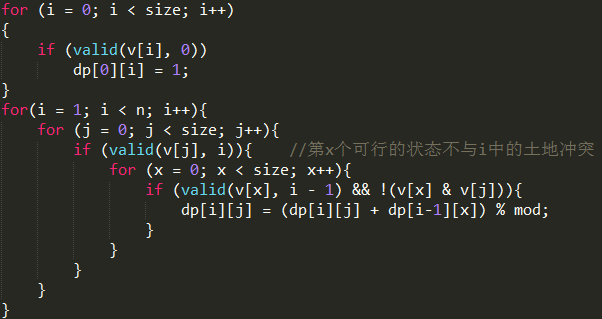


图 7.4 计算各行各状态的方案个数

最后一行各种状态的情况个数的和即为最后结果的总方案个数，需要注意的是，根据题目要求，最后的结果要对10,000,000做取模运算。计算最终结果的流程如图 7.5所示。

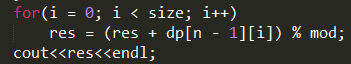


图 7.5 计算总方案个数

## 源代码

#include <iostream>

using namespace std;

#define mod 100000000

#define N 15

#define M 4100

int map[N]; //该行的输入状态

int v[M]; //所有可能的状态的集合

int dp[N][M]; //表示在第i行状态为j时候可以放牛的情况数量

int n, m;

int size;

void getValidState(){

size = 0;

int i = 0;

for (i = 0; i < (1<<m); i++){

if ((i & i<<1) == 0)

v[size++] = i;

}

return;

}

bool valid(int state, int r){

return (!(state & (~map[r])));

}

int main(){

int i = 0, j = 0, x = 0, y = 0;

int res = 0;

while(cin>>n>>m){

res = 0;

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

map[i] = 0;

for(j = 0; j < m; j++){

cin>>x;

if (x)

map[i] += (1<<(m - j - 1));

}

}

getValidState();

for (i = 0; i < size; i++)

{

if (valid(v[i], 0))

dp[0][i] = 1;

}

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

for (j = 0; j < size; j++){

if (valid(v[j], i) == true){ //第x个可行的状态不与i中的土地冲突

for (x = 0; x < size; x++){

if (valid(v[x], i - 1) && !(v[x] & v[j])){

dp[i][j] = (dp[i][j] + dp[i-1][x]) % mod;

}

}

}

}

}

for(i = 0; i < size; i++)

res = (res + dp[n - 1][i]) % mod;

cout<<res<<endl;

}

return 0;

}

## 测试分析

样例输入如下所示：

|  |
| --- |
| 2 3   1. 1 1   0 1 0 |

第一行表示土地为2行3列，第二行和第三行的2×3个二进制的数用来表示对应位置上是否能放牛。根据题意，有1种方式不放牛，有4种方式在一块地上放牛，有3种方式在两块地上放牛，有1种方式在三块地上放牛，没有其他的放牛方案，所以总共的方案个数为1+4+3+1 = 9 种。程序的运行结果如图 7.6所示。

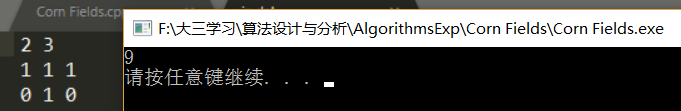


图 7.6 样例输入测试结果

如果所有的地都不能放牛，测试用例如下所示：

|  |
| --- |
| 1. 3 2. 0 0   0 0 0 |

根据分析，只有1种所有地都不放牛的方案符合要求，程序的运行结果如图 7.7所示。

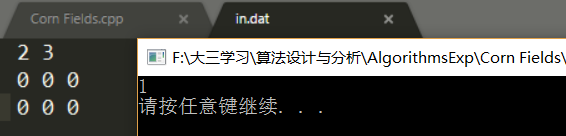


图 7.7 所有土地不可用情况测试

如果不存在地，即M=N=0时，测试用例如下所示：

|  |
| --- |
| 0 0 |

此时，不存在可行的放牛方案，测试结果如图 7.9所示。



图 7.8 土地不存在情况测试

提交到网上测试系统中进行黑箱测试的结果如图 7.9所示。

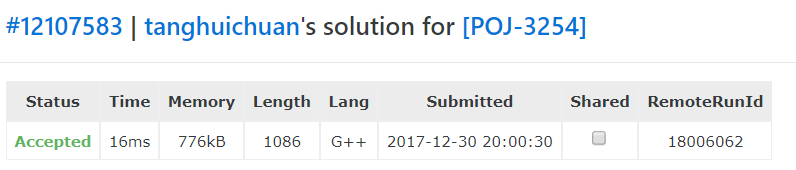


图 7.9 POJ-3254黑箱测试结果

根据上述的测试情况分析，在正常的情况下和边界情况下（没有土地）程序的计算结果都与理论的结果相同，证明程序的运行是正确的。

## 技术总结

本题的关键就在于如何使用状态压缩来存储每一种情况。状态压缩是指将状态数据（通常为二进制的数据）存储到一定数据类型的变量中，通过对变量的按位操作来实现对状态的判断。状态压缩算法适用的题目应该符合以下条件：

·解法需要保存一定的状态数据（表示一种状态的一个数据值），每个状态数据通常情况下是可以通过二进制来表示的。这就要求状态数据的每个单元只有两种状态，比如棋盘上的格子，可以用二进制的1来表示放置棋子，用二进制的0来表示不放棋子。这样用0或者1来表示状态数据的每个单元，而整个状态数据就是一个由一串0和1组成的二进制数。

·解法需要将状态数据存储在一个基本数据类型中，比如int、long等等，即所谓的状态压缩。状态压缩一方面是缩小了数据存储的空间，另一方面在状态对比和状态整体处理时能够提高运算效率。这样就要求状态数据中的单元个数不能太大，比如当本题中的N大于32时，就不能用32位的int型变量来进行存储了。

# Paid Roads ( POJ 3411 )

## 题目描述

·Description

|  |
| --- |
| A network of **m** roads connects **N** cities (numbered from 1 to **N**). There may be more than one road connecting one city with another. Some of the roads are paid. There are two ways to pay for travel on a paid road **i** from city **ai** to city **bi**:  ·in advance, in a city **ci** (which may or may not be the same as **ai**);  ·after the travel, in the city **bi**.  The payment is **Pi** in the first case and **Ri** in the second case.  Write a program to find a minimal-cost route from the city 1 to the city **N**. |

·Input

|  |
| --- |
| The first line of the input contains the values of **N** and **m**. Each of the following **m**lines describes one road by specifying the values of **ai**, **bi**, **ci**, **Pi**, **Ri** (1 ≤ **i**≤ **m**). Adjacent values on the same line are separated by one or more spaces. All values are integers, 1 ≤ **m, N** ≤ 10, 0 ≤ **Pi** , **Ri** ≤ 100, **P**i ≤ **Ri** (1 ≤ **i**≤ **m**). |

·Output

|  |
| --- |
| The first and only line of the file must contain the minimal possible cost of a trip from the city 1 to the city **N**. If the trip is not possible for any reason, the line must contain the word ‘**impossible**’. |

·Sample Input

|  |
| --- |
| 4 5  1 2 1 10 10   1. 3 1 30 50 2. 4 3 80 80   2 1 2 10 10  1 3 2 10 50 |

·Sample Output

|  |
| --- |
| 110 |

## 算法设计

根据题意，一条公路有五个属性：a表示起点城市，b表示终点城市，c是一个用来作判断的城市，如果在走这条公路之前途经过c代表的城市，则收费p元，否则收费r元。定义一个存储路的属性的结构，包括五个整型数据，分别用来存储上述五个变量的值，结构的定义如图 8.1所示。

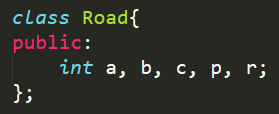


图 8.1 公路结构的定义

首先读入城市个数n以及公路条数m，再依次读入m条公路的五个属性，存储在以Road为基本元素的数组road中，如图所示。

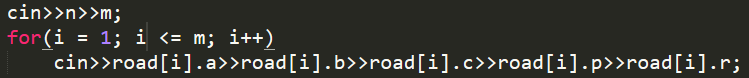


图 8.2 读入城市与公路数据

因为可以经过一个城市多次，所以定义一个整型数组visit，用来存储经过每个城市的次数。在初始状态下，因为是从第一个城市出发，所以visit[1]置为1，其他的数组元素中的值置为0，如图 8.3所示。

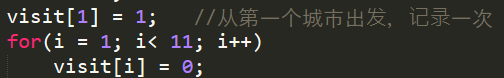


图 8.3 途经城市记录数组visit的初始化

采取深度优先的遍历方式来寻找所有从源到目的地的可行路径，一次搜索的终止条件是当前抵达目的地或者当前路径的费用已经超过了目前的最小费用了。如果当前没有抵达目的地且费用没有超过目前的最小费用，则对从此城市出的公路进行遍历。在进入下一个结点之前要先将下一个城市的visit的次数加一，并在此次递归返回后将该城市的visit的次数减1。在调用DFS函数对下一个城市进行递归调用之前，要根据是否经过了即将要走的路径规定的城市对费用进行修改，再作为参数传入函数。DFS函数的函数体如图 8.4所示。



图 8.4 DFS函数体实现

在DFS的递归调用中，如果检测到当前城市为目的城市且目前的费用比min\_cost的值要小，则将min\_cost的值更新成这个更小的值。因此在递归调用结束后，存放在全局变量min\_cost中的即为最小的花费。如果min\_cost为初始值2000，说明没有从源到目的结点的可达路径，此时输出“impossible”；如果不为初始值，则输出最小的花费。输出部分如图 8.5所示。

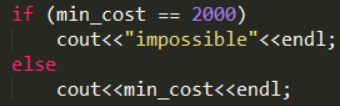


图 8.5 输出运算结果示意图

## 源代码

#include <iostream>

using namespace std;

class Road{

public:

int a, b, c, p, r;

};

Road \*road = new Road[11];

int \*visit = new int[11];

int min\_cost = 2000;

int m = 0, n = 0;

void DFS(int a, int fee){ //a为当前城市编号，fee为当前的费用

if(a == n && min\_cost > fee){

min\_cost = fee;

return;

}

int i = 1;

for(; i <= m; i++){

if (a == road[i].a && visit[road[i].b] <= 3){

int b = road[i].b;

visit[b]++;

if (visit[road[i].c])

DFS(b, fee + road[i].p);

else

DFS(b, fee + road[i].r);

visit[b]--;

}

}

return;

}

int main(){

int i = 0;

visit[1] = 1; //从第一个城市出发，记录一次

for(i = 1; i< 11; i++)

visit[i] = 0;

cin>>n>>m;

for(i = 1; i <= m; i++)

cin>>road[i].a>>road[i].b>>road[i].c>>road[i].p>>road[i].r;

DFS(1, 0);

if (min\_cost == 2000)

cout<<"impossible"<<endl;

else

cout<<min\_cost<<endl;

return 1;

}

## 测试分析

样例输入如下所示：

|  |
| --- |
| 4 5  1 2 1 10 10   1. 3 1 30 50 2. 4 3 80 80   2 1 2 10 10  1 3 2 10 50 |

其中，第一行的两个数据表示有四个城市，有五条公路。后跟五行，分别为这五条路的属性。以第二条为例，该行表示该公路可以从第四个城市去第三个城市，如果在上这条公路之前经过过第一个城市，则收费30,，否则收费50。程序运行结果如图 8.6所示。

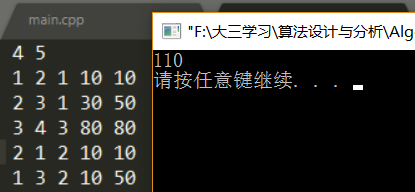


图 8.6 样例输入的运行结果

将程序的运行结果与样例输出比较可知，程序运行结果正确。提交到网上测试系统中进行黑箱测试的结果如图 8.7所示。



图 8.7 POJ-3411黑箱测试结果

## 技术总结

本题是在图的深度优先搜索的过程中增加一个对递归过程花费的计算。本题需要注意的是，并不是最短的路径花费最少。所以本题适合采用深度优先遍历而不是采用广度优先搜索。

在深度搜索过程中，需要注意两个点：

·剪枝的条件有两个：到达终点和当前花费已经超过了目前的最小花费值min\_cost

·递归传入花费时要根据此次递归有没有经过公路上要求的城市来相应地增加个各自的费用。