2010 浙江大学计算机考研机试题解

我要解决什么问题？
我要如何去解决？
编码！

第一题　A+B 问题

题目描述：

给定两个整数 A 和 B，其表示形式是：从个位开始，每三位数用逗号","隔开。

现在请计算 A+B 的结果，并以正常形式输出。

输入：

输入包含多组数据数据，每组数据占一行，由两个整数 A 和 B 组成（-10^9 < A,B < 10^9）。

输出：

请计算 A+B 的结果，并以正常形式输出，每组数据占一行。

样例输入：

-234,567,890 123,456,789
1,234 2,345,678

样例输出：

-111111101
2346912

解题思路：

中文题目我就不详细解释题目意思了。本题就是求两个数的和。问题在于，这两个数为了阅读方便采用逗号分隔了。而逗号分隔的整数在 C 语言中无法中整数直接读取的。同时注意到这两个数的范围是-10^9~10^9 那么，他们的和的范围是-2*10^9~2*10^9，是在 4 字节 int 型范围内的。所以可以用 int 型来存储这两个经过转换后的数以及他们的和。一旦将这两个数转换为 int 后，直接相加就可以了。所以本题需要解决的是如何将给定的两个数转换为 int 类型。

下面我就给出将字符串转换为整数的一般方法：采用一个循环从字符串的开头到结尾依次遍历，将每个字符转换为整数。将每次转换后的结果追加到已转换后的结果当中。不需要转换的字符跳过即可。需要注意的是先判断是否是负数。详见参考代码。

AC 代码：

```
#include <stdio.h>

/*
 * 将一个字符串转换为整数
 * */
int str2int(char *str){
    int ans = 0;        // 储存结果
    int i = 0;          // 迭代变量
    int sign = 1;       // 符号，可能是负数
    if(str[0] == '-'){  // 如果最开始有个符号，则是负数，符号变号
        i++;
        sign = -1;
    }
    while(str[i]){      // 对后面的字符进行遍历
        if(str[i]!=','){// 如果不是逗号，则追加到计算的整数当中
            ans = ans*10 + str[i]-'0';  // 这个是转换的关键，追加到后面时前面结果的要向前移动一位，所以要乘以10。
        }
        i++;
    }
    return sign*ans;    // 返回结果，sign 的符号决定了这个数字的符号
}

int main(){

    char str1[20], str2[20];
    while(scanf("%s%s", str1, str2) != EOF){            // 读入两个字符串直到文件结尾
        printf("%d\n", str2int(str1) + str2int(str2));  // 把转换结果相加并输出
    }

    return 0;
}
```

第二题　ZOJ 问题

题目描述：

对给定的字符串(只包含'z','o','j'三种字符),判断他是否能 AC。

是否 AC 的规则如下：

1. zoj 能 AC；

2. 若字符串形式为 xzojx，则也能 AC，其中 x 可以是 N 个'o' 或者为空；

3. 若 azbjc 能 AC，则 azbojac 也能 AC，其中 a,b,c 为 N 个'o'或者为空；

输入：

输入包含多组测试用例，每行有一个只包含'z','o','j'三种字符的字符串，字符串长度小于等于 1000。

输出：

对于给定的字符串，如果能 AC 则请输出字符串“Accepted”，否则请输出“Wrong Answer”。

样例输入：

```
zoj
ozojo
ozoojoo
oozoojoooo
zooj
ozojo
ooozojo
zojoooo
```

样例输出：

```
Accepted
Accepted
Accepted
Accepted
Accepted
Accepted
Wrong Answer
Wrong Answer
```

解题思路：

怎样的字符串才是符合条件的呢？这个需要仔细分析题目描述中的三个条件。前两个条件比较简单，主要分析第三个条件。我们注意前两个条件，每个条件只确定了一个比较明显的形式，第三个条件每次变换就会有很多形式了。那么，通过条件三的变换会得到怎样的形式呢？条件三的变化必然是基于一个初始状态的，这个初始状态就来自于条件一或者提交二。如果条件三种的 a 不为空，则经过条件三变换后就不能得到符合条件一和条件二的形式了，因为经过条件三变换后的形式中 z 与 j 之间的 o 的数目大于 1，而在前两个条件当中中间 o 的数目只能为 1。再仔细观察条件三的初始状态会发现起始时，a 和 c 中 o 的数目是相同的，这样每次经过条件三的变换 z 与 j 之间会增加一个 o，而 j 之后会增加与 a 中相同数目的 o。因而最终 j 之后 o 的数目是 z 之前 o 数目与 z 和 j 之间 o 数目的乘积。
详见参考代码。

AC 代码：

```
#include <stdio.h>

bool Judge(char *str){      // 判断字符串是否符合要求
    int oBeforeZ=0, oMid=0, oAfterJ=0;  // 分别存储 z 前面、z 与 j 之间以及 j 之后的o的数目。
    int i = 0;
    while(str[i] && str[i]=='o'){       // 计算 z 之前的 o 的数目，这里注意不能超出字符串边界（用 str[i]来判断，下同）
        i++;
        oBeforeZ++;
    }
    if(str[i] != 'z'){      // 如果第一次遍历 o 结束后的字符不是 z 则不符合要求
        return false;
    }

    i++;
    while(str[i] && str[i]=='o'){       // 计算 z 与 j 之间 o 的数目
        i++;
        oMid++;
    }
    if(!oMid || str[i]!='j'){           // 如果中间没有 o 或者中间 o 之后的字符不是 j，则不符合
        return false;
    }

    i++;
    while(str[i] && str[i]=='o'){       // 计算 j 之后 o 的数目
        i++;
        oAfterJ++;
```

```
    }

    if(str[i]){                          // 如果 j 之后的 o 都统计完了还没有到达字符串结尾，则不符合
        return false;
    }

    return oBeforeZ*oMid == oAfterJ;     // 只有 z 前面 o 的数目与中间 o 数目的乘积与 j 后面 o 的数目相同，才符合
}

int main(){

    char str[1100];      // 用来读入字符串
    while(scanf("%s", str) != EOF){
        puts(Judge(str) ? "Accepted" : "Wrong Answer");
    }

    return 0;
}
```

第三题　奥运排序问题

题目描述：

按要求，给国家进行排名。

输入：

有多组数据。

第一行给出国家数 N，要求排名的国家数 M，国家号从 0 到 N-1。

第二行开始的 N 行给定国家或地区的奥运金牌数，奖牌数，人口数（百万）。

接下来一行给出 M 个国家号。

输出：

排序有 4 种方式: 金牌总数 奖牌总数 金牌人口比例 奖牌人口比例

对每个国家给出最佳排名排名方式 和 最终排名

格式为: 排名:排名方式

如果有相同的最终排名，则输出排名方式最小的那种排名，对于排名方式，金牌总数 < 奖牌总数 < 金牌人口比例 < 奖牌人口比例

如果有并列排名的情况，即如果出现金牌总数为 100,90,90,80.则排名为 1,2,2,4.

每组数据后加一个空行。

样例输入：

```
4 4
4 8 1
6 6 2
4 8 2
2 12 4
0 1 2 3
4 2
8 10 1
8 11 2
8 12 3
8 13 4
0 3
```

样例输出：

```
1:3
1:1
2:1
1:2

1:1
1:1
```

解题思路：

本题需要解决的是奥运会中各国家最有利的排名方式以及名次。只要进行五次排序即可。首先读入各国家信息，写好国家编号，计算和存储排名所需要的数据。然后按四种排名方式分别对需要排名的国家进行排名，并记录名次。最后使用国家编号对国家进行排名。这样就可以输出结果了。
详见参考代码。

AC 代码：

```
#include <stdio.h>
#include <algorithm>
using namespace std;

struct Nation{                    // 国家信息结构类型
```

```
    int num;                    // 国家编号
    double orderValue[4];       // 用来排名的值
    int order[4];               // 名次
};

Nation na[100000];              // 存储国家信息

bool cmp0(Nation a,Nation b){   // 按国家编号排名
    return a.num < b.num;
}

bool cmp1(Nation a,Nation b){   // 按金牌数排名
    return a.orderValue[0] > b.orderValue[0];
}

bool cmp2(Nation a,Nation b){   // 按奖牌数排名
    return a.orderValue[1] > b.orderValue[1];
}

bool cmp3(Nation a,Nation b){   // 按金牌人口比排名
    return a.orderValue[2] > b.orderValue[2];
}

bool cmp4(Nation a,Nation b){   // 按奖牌人口比排名
    return a.orderValue[3] > b.orderValue[3];
}

int main(){

    int n, m;                   // 用来存储国家总数以及需要排名国家的数目
    int gold, total, pop;       // 用来读取金牌数、奖牌数以及人口
    int need;                   // 存储需要排名的国家编号数

    while(scanf("%d%d", &n, &m) != EOF){
        int i;
        for(i=0; i<n; i++){     // 读入国家信息，并计算金牌人口比以及奖牌人
口比
            scanf("%d%d%d", &gold, &total, &pop);
            na[i].orderValue[0] = gold;
            na[i].orderValue[1] = total;
            na[i].orderValue[2] = 1.0*gold/pop;
            na[i].orderValue[3] = 1.0*total/pop;
            na[i].num = i;
        }
```

```
for(i=0; i<m; i++){          // 只存储需要排名的国家
    scanf("%d", &need);
    na[i] = na[need];
}

// 按金牌数排名，计算名次
sort(na, na+m, cmp1);
for(i=0; i<m; i++){
    if(i>0 && na[i-1].orderValue[0]==na[i].orderValue[0]){
        na[i].order[0] = na[i-1].order[0];
    }else{
        na[i].order[0] = i+1;
    }
}

// 按奖牌数排名，计算名次
sort(na, na+m, cmp2);
for(i=0; i<m; i++){
    if(i>0 && na[i-1].orderValue[1]==na[i].orderValue[1]){
        na[i].order[1] = na[i-1].order[1];
    }else{
        na[i].order[1] = i+1;
    }
}

// 按金牌人口比排名，计算名次
sort(na, na+m, cmp3);
for(i=0; i<m; i++){
    if(i>0 && na[i-1].orderValue[2]==na[i].orderValue[2]){
        na[i].order[2] = na[i-1].order[2];
    }else{
        na[i].order[2] = i+1;
    }
}

// 按奖牌人口比排名，计算名次
sort(na, na+m, cmp4);
for(i=0; i<m; i++){
    if(i>0 && na[i-1].orderValue[3]==na[i].orderValue[3]){
        na[i].order[3] = na[i-1].order[3];
    }else{
        na[i].order[3] = i+1;
    }
```

```
        }

        // 按国家编号排序，输出结果
        sort(na, na+m, cmp0);
        int j, k, index;
        for(i=0; i<m; i++){
            index = 0;
            // 从四种方法中获得最佳结果
            for(j=0; j<4; j++){
                if(na[i].order[j] < na[i].order[index]){
                    index = j;
                }
            }
            printf("%d:%d\n", na[i].order[index], index+1);
        }
        putchar('\n');
    }

    return 0;
}
```

第四题　最短路径问题

题目描述：

给你 n 个点，m 条无向边，每条边都有长度 d 和花费 p，给你起点 s 终点 t，要求输出起点到终点的最短距离及其花费，如果最短距离有多条路线，则输出花费最少的。

输入：

输入 n,m，点的编号是 1~n,然后是 m 行，每行 4 个数 a,b,d,p，表示 a 和 b 之间有一条边，且其长度为 d，花费为 p。最后一行是两个数 s,t;起点 s，终点 t。n 和 m 为 0 时输入结束。

(1<n<=1000, 0<m<100000, s != t)

输出：

输出 一行有两个数， 最短距离及其花费。

样例输入：

3 2

```
1 2 5 6
2 3 4 5
1 3
0 0
```

样例输出：

```
9 11
```

解题思路：

拿到这题，第一印象就是最短路劲问题（题目写得清清楚楚）。当然，你可以直接套用最短路劲的解法。不过这里我要给大家介绍另一种解法，同样是图论里的算法——深度优先搜索。

从起点开始进行深度优先搜索，当搜索的结点到达终点时查看路径总长度与花费是否比已经得到的最短路径和最小花费小，如果要小的话就使用当前的路径和花费取代最短路径和最小花费。

当遍历到终点时是否就结束了？非也，还需要从其他路径遍历，判断其他路径是否又更短花费更小的。这里就需要用到回溯了。所谓回溯，就是还需要往回走来走其他的路径。那么会不会进入循环状态呢？对遍历过的结点进行标记就能够杜绝循环。当然需要在回溯时取消之前的标记。

详见参考代码。

AC 代码：

```
#include <stdio.h>
#include <string.h>

#define MAXSIZE 1100     // 最大节点数
int minDis, minCost;     // 最小距离和最小花费
int info[MAXSIZE][MAXSIZE][2];  // 记录两点之间的距离和花费
bool visited[MAXSIZE];      // 记录是否遍历过了，防止形成回路造成无限循环
int nodeE;               // 记录终点
int numOfNodes;          // 节点总数

void DFS(int curNode, int curDis, int curCost){
    if(curDis>minDis || curCost>minCost){   // 如果当前的距离大于最小距离或者当前花费大于最小花费，没必要继续搜索了
        return ;
    }

    if(curNode == nodeE){       // 如果到达终点了，则记录最小距离和最小花费
        minDis = curDis;
        minCost = curCost;
        return ;
    }
```

```
    int i;
    for(i=1; i<=numOfNodes; i++){
        if(!visited[i] && info[curNode][i][0]){     // 如果和当前节点连接的另一个节点没有遍历过则遍历之
            visited[i] = true;                      // 标记成已经遍历过
            DFS(i,  curDis+info[curNode][i][0],  curCost+info[curNode][i][1]);
    // 遍历相邻节点
            visited[i] = false;                     // 回溯回来时撤销标记
        }
    }
}

int main(){

    int lines;                      // 记录行数
    int nodeA, nodeB, dis, cost;    // 记录数据
    int nodeS;                      // 记录起始点
    while(scanf("%d%d", &numOfNodes, &lines), numOfNodes||lines){
        int i;
        memset(info, 0, sizeof(info));          // 将各节点信息清空
        memset(visited, 0, sizeof(visited));    // 将访问过的标记清空
        for(i=0; i<lines; i++){     // 读入各节点的信息，哪两个节点之间有通路、长度以及花费是多少
            scanf("%d%d%d%d", &nodeA, &nodeB, &dis, &cost);
            info[nodeA][nodeB][0] = dis;
            info[nodeA][nodeB][1] = cost;
            info[nodeB][nodeA][0] = dis;
            info[nodeB][nodeA][1] = cost;
        }
        scanf("%d%d", &nodeS, &nodeE);  // 读入起点和终点序号
        visited[nodeS] = true;          // 将起点设置为遍历过了
        minDis = 0x7FFFFFFF;            // 最小路径开始时设置为最大，这样在以后的搜索中逐渐减小
        minCost = 0x7FFFFFFF;           // 最小花费与最小路径是相似的
        DFS(nodeS, 0, 0);               // 通过深度优先搜索来找寻最短路劲以及最小花费
        printf("%d %d\n", minDis, minCost); // 输出结果
    }

    return 0;
}
```

第五题　二叉搜索树问题

题目描述：

判断两序列是否为同一二叉搜索树序列

输入：

开始一个数 n，(1<=n<=20) 表示有 n 个需要判断，n= 0 的时候输入结束。

接下去一行是一个序列，序列长度小于 10，包含(0~9)的数字，没有重复数字，根据这个序列可以构造出一颗二叉搜索树。

接下去的 n 行有 n 个序列，每个序列格式跟第一个序列一样，请判断这两个序列是否能组成同一颗二叉搜索树。

输出：

如果序列相同则输出 YES，否则输出 NO

样例输入：

```
2
567432
543267
576342
0
```

样例输出：

```
YES
NO
```

解题思路：

首先要了解什么是二叉搜索树，二叉搜索树有哪些特点。了解过够就需要分析题目的意思了。

本题给定一个字符串，根据这个字符串可以构成一个二叉搜索树。然后后面再给出若干字符串，每个字符串也能构成一个二叉搜索树，问后面字符串所构成的二叉搜索树与第一个是否相同。于是解决本题的关键步骤是构建二叉搜索树、比较二叉搜索树（如果有时间的话还需要释放二叉搜索树，由于本题的数据量不大，同时考研机试的时间比较紧，不释放也无所谓。）比较二叉树只要使用数的遍历算法即可，对于构建二叉搜索树，则使用二叉搜索树的搜索算法。将待插入的位置定位到合适的位置（根据二叉搜索树的性质）。

现在简要介绍二叉搜索树的构建算法。从树根开始往下定位，比较待插入的元素的值与已构建的搜索二叉树中相应元素的值。如果待插入的元素值比较小则遍历左孩子，否则遍历右孩子。直至遍历到空结点为止。

这里需要注意的是如果使用一个结点类型指针来指向某个结点，则在给该结点赋值时实

际上二叉树上的指向同一位置的指针是没有改变的。所以应当使用结点指针类型的指针类型。

详见参考代码。

AC 代码：

```
#include <stdio.h>
#include <stdlib.h>

struct BSTNodeType{
    int value;
    BSTNodeType *lChild, *rChild;
};  // 定义二叉搜索树结点结构类型

BSTNodeType * BuildBST(char *str){  // 通过字符串构建二叉搜索树
    int i = 0;
    BSTNodeType * root = NULL;      // 树根首先初始化为 NULL
    while(str[i]){                  // 对字符串中的数据进行遍历
        int value = str[i] - '0';   // 将字符转换为整数
        BSTNodeType ** pTmp = &root;    // 用来遍历二叉搜索树，注意这里是搜索树结点指针类型的指针类型
        while(*pTmp){
            if(value < (*pTmp)->value){// 如果需要插入的值比当前指针指向的值小，则放到该结点的左孩子中
                pTmp = &(*pTmp)->lChild;
            }else{                  // 否则就放到右孩子中
                pTmp = &(*pTmp)->rChild;
            }
        }
        *pTmp = (BSTNodeType *)malloc(sizeof(BSTNodeType)); // 分配新结点
        (*pTmp)->value = value;     // 存储值
        (*pTmp)->lChild = (*pTmp)->rChild = NULL;   // 分配的新结点的左右孩子均为空
        i++;
    }
    return root;
}

bool Judge(BSTNodeType *bstA, BSTNodeType *bstB){   // 给定两个二叉树，判断是否相等
    if(bstA && bstB){       // 如果两棵树均没有遍历完
        if(bstA->value != bstB->value){ // 如果有结点不相同，则两棵树不同
            return false;
        }else{
            if(bstA->lChild){                                   // 如果有左孩子
```

```
                if(Judge(bstA->lChild, bstB->lChild)){          // 如果判断左孩子没有问题
                    return Judge(bstA->rChild, bstB->rChild);   // 则返回右孩子判断的结果
                }else{                                          // 如果左孩子有问题，则返回false
                    return false;
                }
            }else{  // 如果没有左孩子，则返回右孩子判断的结果
                return Judge(bstA->rChild, bstB->rChild);
            }
        }
    }else if(!bstA && !bstB){   // 如果两棵树判断完毕，则返回true
        return true;
    }else{                      // 如果两者只有一个为空，则说明两棵树不同
        return false;
    }
}

void FreeBST(BSTNodeType *root){
    if(root->lChild){
        FreeBST(root->lChild);      // 释放左子树
    }

    if(root->rChild){
        FreeBST(root->rChild);      // 释放右子树
    }

    free(root);     // 释放空间
}

int main(){

    int n;
    while(scanf("%d", &n), n){      // 读入需要比较的数目
        char str[20];               // 用来读入字符串
        scanf("%s", str);           // 读入第一个字符串
        BSTNodeType *bst1=NULL, *bst2=NULL; // 用来存储构建好的搜索树
        bst1 = BuildBST(str);
        while(n--){
            scanf("%s", str);       // 读入以下的字符串
            bst2 = BuildBST(str);   // 构建需要比较的搜索树
            if(Judge(bst1, bst2)){  // 判断两棵树是否相同
                puts("YES");
```

```
            }else{
                puts("NO");
            }
            FreeBST(bst2);          // 比较完成后将内存释放
        }
        FreeBST(bst1);              // 将第一棵树的内存释放
    }

    return 0;
}
```