**目录**

[题目1 迷宫问题 3](#_Toc110010941)

[1 问题描述 3](#_Toc110010942)

[2 输入输出要求 3](#_Toc110010943)

[3 主要思路 4](#_Toc110010944)

[4 源代码 4](#_Toc110010945)

[5 运行结果 7](#_Toc110010946)

[题目1 编译器中的表达式求值问题 8](#_Toc110010947)

[1 问题描述 8](#_Toc110010948)

[2 输入输出要求 8](#_Toc110010949)

[3 主要思路 8](#_Toc110010950)

[4 源代码 9](#_Toc110010951)

[5 运行结果 13](#_Toc110010952)

[题目2 数据压缩中的编码问题 14](#_Toc110010953)

[1 问题描述 14](#_Toc110010954)

[2 输入输出要求 14](#_Toc110010955)

[3 主要思路 14](#_Toc110010956)

[4 源代码 15](#_Toc110010957)

[5 运行结果 19](#_Toc110010958)

[题目3 哈利波特的考试 20](#_Toc110010959)

[1 问题描述 20](#_Toc110010960)

[2 输入输出格式要求 20](#_Toc110010961)

[3 主要思路 21](#_Toc110010962)

[4 源代码 21](#_Toc110010963)

[5 运行结果 23](#_Toc110010964)

[题目4 平衡二叉树构建 24](#_Toc110010965)

[1 问题描述 24](#_Toc110010966)

[2 输入输出要求 24](#_Toc110010967)

[3 主要思路 25](#_Toc110010968)

[4 源代码 25](#_Toc110010969)

[5 运行结果 27](#_Toc110010970)

[题目5 链式基数排序 28](#_Toc110010971)

[1 问题描述 28](#_Toc110010972)

[2 具体设计要求 28](#_Toc110010973)

[3 主要思路 28](#_Toc110010974)

[4 源代码 28](#_Toc110010975)

[5 运行结果 30](#_Toc110010976)

[题目6 六度空间 31](#_Toc110010977)

[1 问题描述 31](#_Toc110010978)

[2 具体设计要求 31](#_Toc110010979)

[3 主要思路 32](#_Toc110010980)

[4 源代码 32](#_Toc110010981)

[5 运行结果 34](#_Toc110010982)

# 题目1 迷宫问题

## 1 问题描述

迷宫实验是取自心理学的一个古典实验。在该实验中，把一只老鼠从一个无顶大盒子的门放入，在盒中设置了许多墙，对行进方向形成了多处阻挡。盒子仅有一个出口，在出口处放置一块奶酪，吸引老鼠在迷宫中寻找道路以到达出口。对同一只老鼠重复进行上述实验，一直到老鼠从入口到出口，而不走错一步。老鼠经多次试验终于得到它学习走迷宫的路线。

迷宫maze由m行n列的二维数组设置，0表示无障碍，1表示有障碍。设入口为（1，1），出口为（m，n），每次只能从一个无障碍单元移到周围四个方向上任一无障碍单元。编程实现对任意设定的迷宫，求出一条从入口到出口的通路，或得出没有通路的结论。

## 2 输入输出要求

1. 输入

第一行输入m和n，其中m为迷宫的行数，你为迷宫的列数，随后的m行中每一行为以空格隔开的n个0和1。

1. 输出

一条从入口到出口的通路(一组坐标)，或得出没有通路的结论。

1. 样例

输入

0 1 0 0 0

0 1 0 1 0

0 0 0 0 0

0 1 1 1 0

0 0 0 1 0

输出

(0, 0)

(1, 0)

(2, 0)

(2, 1)

(2, 2)

(2, 3)

(2, 4)

(3, 4)

(4, 4)

## 3 主要思路

利用栈和广度优先算法（BFS）解决迷宫问题

题目所给样例有误，若入口处记为（1，1），则最终输出结果应该和运行结果一致

## 4 源代码

#include <bits/stdc++.h>

using namespace std;

int n, m;

int mp[105][105];

bool vis[105][105];

int dx[4] = {1, 0, 0, -1};

int dy[4] = {0, 1, -1, 0};

bool flag;

typedef struct

{

    int x;

    int y;

    int nxt;

} node;

typedef struct

{

    node data[1005];

    int top;

} ST;

void find(int x, int y)

{

    ST q; // STL创建队列

    q.top = -1;

    q.top++;

    q.data[q.top].x = x;

    q.data[q.top].y = y;

    q.data[q.top].nxt = 0;

    while (q.top > -1)

    {

        int da = q.data[q.top].x;

        int db = q.data[q.top].y;

        int nxt = q.data[q.top].nxt;

        q.top--;

        vis[da][db] = 0;

        cout << "(" << da << "," << db << ")" << endl;

        for (int i = 0; i < 4; i++)

        {

            int a = dx[i] + da;

            int b = dy[i] + db;

            if (vis[a][b] == 1)

            {

                q.top++;

                q.data[q.top].x = a;

                q.data[q.top].y = b;

                q.data[q.top].nxt = 0;

            }

        }

    }

}

void solve(int x, int y)

{

    ST q;

    q.top = -1;

    q.top++;

    q.data[q.top].x = x;

    q.data[q.top].y = y;

    q.data[q.top].nxt = 0;

    while (q.top > -1)

    {

        int da = q.data[q.top].x;

        int db = q.data[q.top].y;

        int nxt = q.data[q.top].nxt;

        if (nxt == 4)

        {

            q.top--;

            vis[da][db] = 0;

            continue;

        }

        vis[da][db] = 1;

        int ff = 0;

        if (da == m && db == n)

        {

            find(1, 1);

            flag = 1;

            return;

        }

        else

        {

            for (int i = nxt; i < 4; i++)

            {

                int a = da + dx[i];

                int b = db + dy[i];

                if (a < 1 || a > m || b < 1 || b > n || vis[a][b] || mp[a][b] == 1)

                {

                    continue;

                }

                ff = 1;

                q.data[q.top].x = da;

                q.data[q.top].y = db;

                q.data[q.top].nxt = nxt + 1;

                q.top++;

                q.data[q.top].x = a;

                q.data[q.top].y = b;

                q.data[q.top].nxt = 0;

                break;

            }

            if (ff == 0)

            {

                q.top--;

                vis[da][db] = 0;

            }

        }

    }

}

int main()

{

    m = 5, n = 5;

    for (int i = 1; i <= m; i++)

    {

        for (int j = 1; j <= n; j++)

        {

            scanf("%d", &mp[i][j]);

        }

    }

    vis[1][1] = 1;

    solve(1, 1);

    if (!flag)

    {

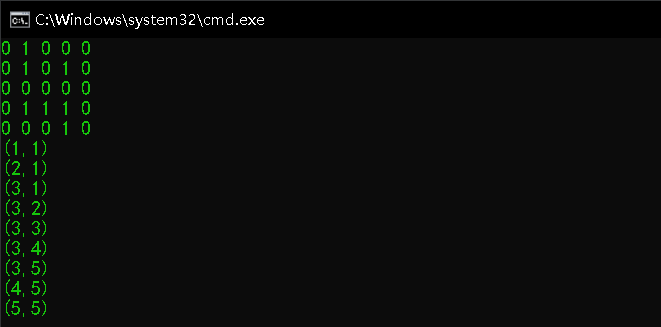
        printf("NO\n");

    }

    return 0;

}

## 5 运行结果



# 题目1 编译器中的表达式求值问题

## 1 问题描述

表达式求值是比较常见的问题，通常在编写程序时，直接写出表达式让编译器去处理，却很少关心编译器是如何对表达式求值。

整数加减乘除四则运算的规则如下：

1）先乘除，后加减；

2）同一优先级，从左算到右；

3) 先括号内，后括号外。

按中缀形式输入一个四则运算的表达式，利用算符优先关系把其转换为后缀表达式输出，并求表达式的值。假设输入的表达式中的操作数都是1位整数。

一个简单合法的中缀表达式可由“(”，“)”，“0”~“9”，“+”，“-”，“\*”，“/”(“/”表示整数除法)构成。

**教学内容：**综合实践栈结构。

**教学目标：**学会使用栈解决实际问题。

## 2 输入输出要求

1. 输入要求

输入数据可以有多组，每一组测试数据为带小括号的四则运算中缀表达式。

1. 输出要求

对于每组测试数据输出两行，第一行四则运算表达式的后缀表达式（每个数字或者操作符之间输出空格，最后一个元素后没有空格），第二行运算结果。 除法操作为整除，比如6/4=1。

1. 样例

输入

-7/(5-2)+2.5

输出

0.5

## 3 主要思路

1. 初始化两个栈：存放运算符的栈S1和存储中间结果的栈S2
2. 遍历中缀表达式
3. 遇到操作数时，入栈S2
4. 遇到运算符时，与S1栈顶运算符的优先级：

·如果S1为空，或者栈顶运算符为左括号，则直接入栈

·若优先级比栈顶运算符的高，则也入栈S1

·否则，将S1栈顶的运算符出栈并入栈S2中，然后与S1新的栈顶运算符进行比较

1. 遇到括号时

·如果是左括号，则直接入栈S1

·如果是右括号，依次弹出S1栈顶的运算符，并入栈S2，直到遇到左括号为止

·重复上述步骤，直到表达式最右边

1. 将S1中剩余的运算符依次入栈S2、
2. 依次出栈S2中的元素并输出，结果的逆序即为中缀表达式对应的后缀表达式
3. 表达式求值的思路：从左到右扫描表达式，遇到数字时，将数字入栈，遇到运算符时，出栈顶的两个数，用运算符对他们做相应的计算，并将结果入栈；重复上述过程直到表达式最右端。最后运算得出的值即为表达式的结果。

## 4 源代码

#include<bits/stdc++.h>

using namespace std;

string format(string str){

    for(int i = 0;i < str.length(); i++){

        if(str[i] == '-'){

            if(i == 0){

                str.insert(0,1,'0');

            }else if(str[i-1] == '('){

                str.insert(i,1,'0');

            }

        }

    }

    return str;

}

int prior(char c){

    if(c == '+' || c == '-'){

        return 1;

    }else if(c == '\*' || c == '/'){

        return 2;

    }else{

        return 0;

    }

}

vector<string> hz(string str){

    vector<string> vs;

    stack<char> st;

    for(int i = 0;i < str.length(); i++){

        string tmp = "";

        switch(str[i]){

            case '+':

            case '-':

            case '\*':

            case '/':

                if(st.empty() || st.top() == '('){

                    st.push(str[i]);

                }else{

                    while(!st.empty() && prior(st.top()) >= prior(str[i]) ){

                        tmp += st.top();

                        vs.push\_back(tmp);

                        st.pop();

                        tmp = "";

                    }

                    st.push(str[i]);

                }

                break;

            case '(':

                st.push(str[i]);

                break;

            case ')':

                while(st.top() != '('){

                    tmp += st.top();

                    vs.push\_back(tmp);

                    st.pop();

                    tmp = "";

                }

                st.pop();

                break;

            default:

            if((str[i]>='0' && str[i]<='9')){

                tmp += str[i];

                while(i+1<str.size() && str[i+1]>='0' && str[i+1]<='9'||str[i+1] == '.')

                {

                    tmp += str[i+1];//若是连续数字

                    ++i;

                }

                vs.push\_back(tmp);

            }

        }

    }

    while(!st.empty()){

        string tmp = "";

        tmp += st.top();

        vs.push\_back(tmp);

        st.pop();

    }

    return vs;

}

double result(vector<string> bh){

    stack<double> st;

    double num,op1,op2;

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

        string tmp = bh[i];

        if(tmp[0] >= '0'&&tmp[0] <= '9'){

            num = atof(tmp.c\_str());

            st.push(num);

        }

        else if(bh[i]=="+")

        {

            op2=st.top();

            st.pop();

            op1=st.top();

            st.pop();

            st.push(op1+op2);

        }

        else if(bh[i]=="-")

        {

            op2=st.top();

            st.pop();

            op1=st.top();

            st.pop();

            st.push(op1-op2);

        }

        else if(bh[i]=="\*")

        {

            op2=st.top();

            st.pop();

            op1=st.top();

            st.pop();

            st.push(op1\*op2);

        }

        else if(bh[i]=="/")

        {

            op2=st.top();

            st.pop();

            op1=st.top();

            st.pop();

            st.push(op1/op2);

        }

    }

    return st.top();

}

void solve(string str){

    str = format(str);

    vector<string> bh = hz(str);

    double k = result(bh);

    if((int)k == k){

        cout<<k<<endl;

    } else{

        printf("%.1f",k);

        cout<<endl;

    }

}

int main(){

    string str;

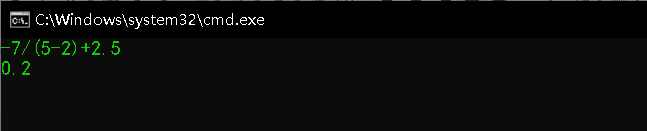
    while(getline(cin,str)){

        solve(str);

    }

}

## 5 运行结果



# 题目2 数据压缩中的编码问题

## 1 问题描述

ZC一天在看计算机的书籍的时候，看到了一个有趣的东西！每一串字符都可以被编码成一些数字来储存信息，但是不同的编码方式得到的储存空间是不一样的！所以ZC 就想是否有一种方式是可以得到字符编码最小的空间值！显然这是可以的，因为书上有这一块内容--哈夫曼编码(Huffman Coding)；一个字母的权值等于该字母在字符串中出现的频率。所以ZC 想让你帮忙，给你一串个字符串，并让你判断这个字符串编码之后的空间值(即01串的长度)？

**教学内容：**综合实践赫夫曼树的构建及赫夫曼编码

**教学目标：**学会根据权值构建赫夫曼树并进行赫夫曼编码来解决数据压缩问题

## 2 输入输出要求

1. 输入要求

第一行是一个数字n，表示有n组数据，然后每一组数据是一串字符串（没有空格只有包含小写字母组成,长度不超过100）.

1. 输出要求

对于每组测试数据输出这个串编码之后的空间值。

1. 样例

输入

2

aaabbbb

aaabbbbccccc

输出

7

19

## 3 主要思路

通过建立哈夫曼树解决数据压缩中的编码问题

## 4 源代码

#include <bits/stdc++.h>

//哈夫曼树结点结构

typedef struct

{

    int weight;              //结点权重

    int parent, left, right; //父结点、左孩子、右孩子在数组中的位置下标

} HTNode, \*HuffmanTree;

//动态二维数组，存储哈夫曼编码

typedef char \*\*HuffmanCode;

// HT数组中存放的哈夫曼树，end表示HT数组中存放结点的最终位置，s1和s2传递的是HT数组中权重值最小的两个结点在数组中的位置

void Select(HuffmanTree HT, int end, int \*s1, int \*s2)

{

    int min1, min2;

    //遍历数组初始下标为 1

    int i = 1;

    //找到还没构建树的结点

    while (HT[i].parent != 0 && i <= end)

    {

        i++;

    }

    min1 = HT[i].weight;

    \*s1 = i;

    i++;

    while (HT[i].parent != 0 && i <= end)

    {

        i++;

    }

    //对找到的两个结点比较大小，min2为大的，min1为小的

    if (HT[i].weight < min1)

    {

        min2 = min1;

        \*s2 = \*s1;

        min1 = HT[i].weight;

        \*s1 = i;

    }

    else

    {

        min2 = HT[i].weight;

        \*s2 = i;

    }

    //两个结点和后续的所有未构建成树的结点做比较

    for (int j = i + 1; j <= end; j++)

    {

        //如果有父结点，直接跳过，进行下一个

        if (HT[j].parent != 0)

        {

            continue;

        }

        //如果比最小的还小，将min2=min1，min1赋值新的结点的下标

        if (HT[j].weight < min1)

        {

            min2 = min1;

            min1 = HT[j].weight;

            \*s2 = \*s1;

            \*s1 = j;

        }

        //如果介于两者之间，min2赋值为新的结点的位置下标

        else if (HT[j].weight >= min1 && HT[j].weight < min2)

        {

            min2 = HT[j].weight;

            \*s2 = j;

        }

    }

}

// HT为地址传递的存储哈夫曼树的数组，w为存储结点权重值的数组，n为结点个数

void CreateHuffmanTree(HuffmanTree \*HT, int \*w, int n)

{

    if (n <= 1)

        return;                                          // 如果只有一个编码就相当于0

    int m = 2 \* n - 1;                                   // 哈夫曼树总节点数，n就是叶子结点

    \*HT = (HuffmanTree)malloc((m + 1) \* sizeof(HTNode)); // 0号位置不用

    HuffmanTree p = \*HT;

    // 初始化哈夫曼树中的所有结点

    for (int i = 1; i <= n; i++)

    {

        (p + i)->weight = \*(w + i - 1);

        (p + i)->parent = 0;

        (p + i)->left = 0;

        (p + i)->right = 0;

    }

    //从树组的下标 n+1 开始初始化哈夫曼树中除叶子结点外的结点

    for (int i = n + 1; i <= m; i++)

    {

        (p + i)->weight = 0;

        (p + i)->parent = 0;

        (p + i)->left = 0;

        (p + i)->right = 0;

    }

    //构建哈夫曼树

    for (int i = n + 1; i <= m; i++)

    {

        int s1, s2;

        Select(\*HT, i - 1, &s1, &s2);

        (\*HT)[s1].parent = (\*HT)[s2].parent = i;

        (\*HT)[i].left = s1;

        (\*HT)[i].right = s2;

        (\*HT)[i].weight = (\*HT)[s1].weight + (\*HT)[s2].weight;

    }

}

// HT为哈夫曼树，HC为存储结点哈夫曼编码的二维动态数组，n为结点的个数

void HuffmanCoding(HuffmanTree HT, HuffmanCode \*HC, int n)

{

    \*HC = (HuffmanCode)malloc((n + 1) \* sizeof(char \*));

    char \*cd = (char \*)malloc(n \* sizeof(char)); //存放结点哈夫曼编码的字符串数组

    cd[n - 1] = '\0';                            //字符串结束符

    for (int i = 1; i <= n; i++)

    {

        //从叶子结点出发，得到的哈夫曼编码是逆序的，需要在字符串数组中逆序存放

        int start = n - 1;

        //当前结点在数组中的位置

        int c = i;

        //当前结点的父结点在数组中的位置

        int j = HT[i].parent;

        // 一直寻找到根结点

        while (j != 0)

        {

            // 如果该结点是父结点的左孩子则对应路径编码为0，否则为右孩子编码为1

            if (HT[j].left == c)

                cd[--start] = '0';

            else

                cd[--start] = '1';

            //以父结点为孩子结点，继续朝树根的方向遍历

            c = j;

            j = HT[j].parent;

        }

        //跳出循环后，cd数组中从下标 start 开始，存放的就是该结点的哈夫曼编码

        (\*HC)[i] = (char \*)malloc((n - start) \* sizeof(char));

        strcpy((\*HC)[i], &cd[start]);

    }

    //使用malloc申请的cd动态数组需要手动释放

    free(cd);

}

//打印哈夫曼编码的函数

void PrintHuffmanCode(HuffmanCode htable, int \*w, int n, int \*length)

{

    // printf("Huffman code : \n");

    for (int i = 1; i <= n; i++)

    {

        // printf("weight=%d\tcode = %s\tlength=%d\n", w[i - 1], htable[i], strlen(htable[i]));

        length[i - 1] = strlen(htable[i]);

    }

}

int main()

{

    int num, n, j, length[100] = {0};

    int \*w, \*temp, \*codelength;

    char str[100][100] = {'\0'};

    scanf("%d", &num);

    getchar();

    for (int k = 0; k < num; k++)

    {

        j = 0;

        int w[26] = {0}, temp[26] = {0}, codelength[26] = {0};

        gets(str[k]);

        for (int i = 0; str[k][i] != '\0'; i++)

        {

            temp[(unsigned int)(str[k][i]) - 97] += 1;

        }

        for (int i = 0; i < 26; i++)

        {

            if (temp[i] > 0)

            {

                w[j] = temp[i];

                j++;

            }

        }

        for (n = 0; w[n] != 0; n++)

            ;

        HuffmanTree htree;

        HuffmanCode htable;

        CreateHuffmanTree(&htree, w, n);

        HuffmanCoding(htree, &htable, n);

        PrintHuffmanCode(htable, w, n, codelength);

        for (int i = 0; i < n; i++)

            length[k] += w[i] \* codelength[i];

    }

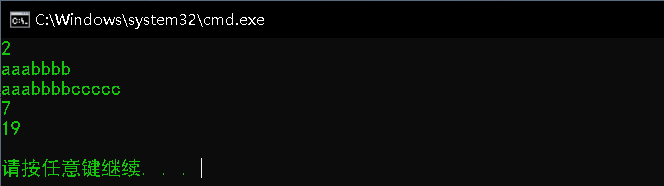
    for (int i = 0; i < n - 1; i++)

        printf("%d\n", length[i]);

    return 0;

}

## 5 运行结果



# 题目3 哈利波特的考试

## 1 问题描述

哈利·波特要考试了，他需要你的帮助。这门课学的是用魔咒将一种动物变成另一种动物的本事。例如将猫变成老鼠的魔咒是haha，将老鼠变成鱼的魔咒是hehe等等。反方向变化的魔咒就是简单地将原来的魔咒倒过来念，例如ahah可以将老鼠变成猫。另外，如果想把猫变成鱼，可以通过念一个直接魔咒lalala，也可以将猫变老鼠、老鼠变鱼的魔咒连起来念：hahahehe。

现在哈利·波特的手里有一本教材，里面列出了所有的变形魔咒和能变的动物。老师允许他自己带一只动物去考场，要考察他把这只动物变成任意一只指定动物的本事。于是他来问你：带什么动物去可以让最难变的那种动物（即该动物变为哈利·波特自己带去的动物所需要的魔咒最长）需要的魔咒最短？例如：如果只有猫、鼠、鱼，则显然哈利·波特应该带鼠去，因为鼠变成另外两种动物都只需要念4个字符；而如果带猫去，则至少需要念6个字符才能把猫变成鱼；同理，带鱼去也不是最好的选择。

**教学内容：**利用最短路径算法解决哈利波特的考试。

**教学目标：**学会利用floyd算法求出图的最短路径算法，并进一步解决哈利波特的考试问题。

## 2 输入输出格式要求

1. 输入要求

第1行输入两个正整数N (≤100)和M，其中N是考试涉及的动物总数，M是用于直接变形的魔咒条数。为简单起见，我们将动物按1~N编号。随后M行，每行给出了3个正整数，分别是两种动物的编号、以及它们之间变形需要的魔咒的长度(≤100)，数字之间用空格分隔。

1. 输出要求

输出哈利·波特应该带去考场的动物的编号、以及最长的变形魔咒的长度，中间以空格分隔。如果只带1只动物是不可能完成所有变形要求的，则输出0。如果有若干只动物都可以备选，则输出编号最小的那只。

1. 样例

输入

6 11

3 4 70

1 2 1

5 4 50

2 6 50

5 6 60

1 3 70

4 6 60

3 6 80

5 1 100

2 4 60

5 2 80

输出

4 70

## 3 主要思路

利用floyd算法求出图的最短路径

## 4 源代码

#include <bits/stdc++.h>

using namespace std;

const int N=105;

int a[N][N];

int main(){

    int n,m;

    scanf("%d%d",&n,&m);

    int u,v,w;

    for (int i=1;i<=n;i++){

        for (int j=1;j<=n;j++){

            a[i][j]=INT\_MAX/2;

            if (i==j) a[i][j]=0;

        }

    }

    for (int i=1;i<=m;i++){

        scanf("%d%d%d",&u,&v,&w);

        a[u][v]=w;

        a[v][u]=w;

    }

    for (int k=1;k<=n;k++){

        for (int i=1;i<=n;i++){

            for (int j=1;j<=n;j++){

                if (a[i][j]>a[i][k]+a[k][j]){

                    a[i][j]=a[i][k]+a[k][j];

                }

            }

        }

    }

    int b,c;

    int mi=INT\_MAX;

    int mii=1;

    int ans=0;

    for (int i=1;i<=n;i++){

        ans=0;

        for (int j=1;j<=n;j++){

            ans=max(a[i][j],ans);

        }

        if (mi>ans){

            mii=i;

            mi=ans;

        }

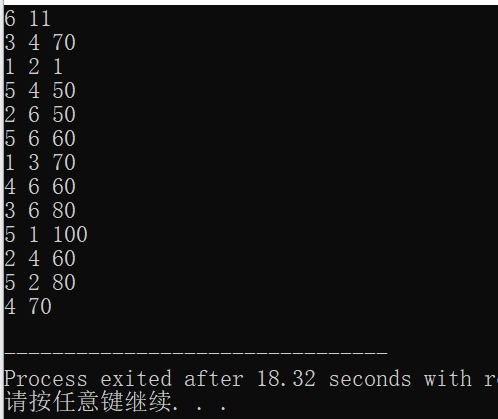
    }

    printf("%d %d",mii,mi);

    return 0;

}

## 5 运行结果



# 题目4 平衡二叉树构建

## 1 问题描述

建立二叉排序树的目的是为查找数据。同一组数据，由于各数据的插入顺序不同，得到不同的二叉排序树，有些二叉排序树的搜索查找的时间复杂度较低，有些则较高，几乎接近顺序查找的复杂度。为了保证所构建的二叉排序树的搜索查找复杂度接近最好，G. M. Adelson-Velsky和E.M.Landis发明了自平衡二叉查找树（Self-Balancing Binary Search Tree）,简称平衡二叉树或AVL树。AVL树树中的任何节点的两个子树的最大高度差为1，所以AVL树也被称为高度平衡树。在构建AVL的过程中，一旦出现树中每个节点的两个子树的最大高度差大于1，就需要进行平衡。因此AVL树也是一种特殊的二叉排序树。

实现AVL树的建立、插入等操作，根据输入的数，建一颗AVL树，初始树为空，重复的数不用插入树中。

**教学内容：**综合实践平衡二叉树的构建及查找。

**教学目标：**学会在构建二叉排序树的过程中进行平衡处理以实现平衡二叉树。

## 2 输入输出要求

1. 输入要求

第一行一个整数N，表示要插入到AVL树的数的个数，第二行N个数，表示要插入的数。（5<=N<=10000）

1. 输出要求

输出两行，第一行为AVL树中元素的个数，第二行为AVL树的先序序列。

1. 样例

输入

6

12 23 32 4 8 12

输出

5

23 8 4 12 32

## 3 主要思路

就是平衡二叉树的构建及查找

## 4 源代码

#include <bits/stdc++.h>

using namespace std;

struct AVL{

    int num;

    int h;

    struct AVL\*l;

    struct AVL\*r;

};

int geth(struct AVL\*T){

    if(T==NULL)

        return 0;

    else

        return T->h;

}

struct AVL\* Lturn(struct AVL\*K2) {

    struct AVL \*K1;

    K1=K2->l;

    K2->l=K1->r;

    K1->r=K2;

    K2->h=max(geth(K2->l), geth(K2->r)) + 1;

    K1->h=max(geth(K1->l), K2->h) + 1;

    return K1;

}

struct AVL \*Rturn(struct AVL \*K1) {

    struct AVL \*K2;

    K2=K1->r;

    K1->r=K2->l;

    K2->l=K1;

    K1->h=max(geth(K1->l), geth(K1->r)) + 1;

    K2->h=max(K1->h, geth(K2->r)) + 1;

    return K2;

}

struct AVL \*LLturn(struct AVL \*K2)

{

    K2->l=Rturn(K2->l);

    return Lturn(K2);

}

struct AVL \*RRturn(struct AVL \*K1)

{

    K1->r=Lturn(K1->r);

    return Rturn(K1);

}

struct AVL \*InsertAVL(struct AVL\*T,int X){

    if (T==NULL){

        T=(struct AVL\*)malloc(sizeof(struct AVL));

        T->num=X;

        T->h=1;

        T->l=T->r=NULL;

    } else if (X<T->num){

        T->l=InsertAVL(T->l,X);

        if (geth(T->l)-geth(T->r)==2)

            if (X<T->l->num)

                T=Lturn(T);

            else

                T=LLturn(T);

    } else if (X>T->num) {

        T->r=InsertAVL(T->r,X);

        if (geth(T->r)-geth(T->l)==2)

            if (X>T->r->num)

                T=Rturn(T);

            else

                T=RRturn(T);

    }

    T->h=max(geth(T->l),geth(T->r))+1;

    return T;

}

void InOrder(struct AVL\*T){

    if(T!=NULL){

        printf("%d ",T->num);

        InOrder(T->l);

        InOrder(T->r);

    }

}

set<int> se;

int main(){

    int i;

    int a[6];

    struct AVL \*T;

    T=NULL;

    int n;scanf("%d",&n);

    for(i=0;i<n;i++){

        scanf("%d",&a[i]);

        se.insert(a[i]);

        T=InsertAVL(T,a[i]);

    }

    printf("%d\n",se.size());

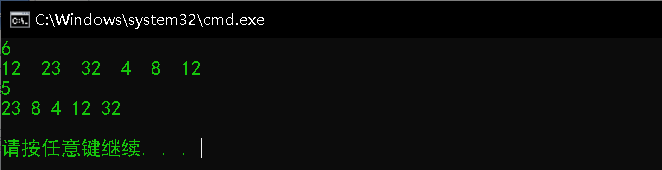
    InOrder(T);

    printf("\n");

    return 0;

}

## 5 运行结果



# 题目5 链式基数排序

## 1 问题描述

基数排序是借助“分配”和“收集”两种操作进行排序的一种内部排序算法，对已知的数据进行基数排序。

**教学内容：**综合实践链式基数排序。

**教学目标：**学会利用链式基数排序解决实际问题。

## 2 具体设计要求

1. 输入要求

输入多行数据 第一行输入一个整数n(n <= 20000)，表示有n个整数要进行排序，第二行输入n个整数。

1. 输出要求

对于每组测试数据，输出每轮分配收集后，生成的n个整数序列，每个数以空格隔开。

1. 样例

输入

9

9 8 7 6 5 4 3 2 1

输出

1 2 3 4 5 6 7 8 9

## 3 主要思路

利用链式基数排序算法来对给定数据进行排序

## 4 源代码

#include<bits.stdc++.h>

using namespace std;

typedef struct Node {

    int data;

    Node \*next;

}Node;

void AddNode(Node \*buc,int data) {

    Node \*p;

    for(p=buc;p->next!=NULL;p=p->next);

    p->next=(Node\*)malloc(sizeof(Node));

    p->next->data=data;

    p->next->next=NULL;

}

void DeleteNode(Node \*buc,Node \*pos) {

    Node \*p;

    for(p=buc;p->next!=pos;p=p->next);

    p->next=pos->next;

    free(pos);

}

void RadixSort(int a[],int n){

    int i,j,tmp,pow=10;

    Node bucket[10],q,\*p;

    for(i=0;i<10;i++){

        bucket[i].next=NULL;

        bucket[i].data=0;

    }

    for(i=0;i<n;i++){

        tmp=a[i]%pow;

        AddNode(&bucket[tmp],a[i]);

    }

    pow\*=10;

    for(j=0;j<8;j++){

        for(i=0;i<10;i++){

            for(p=bucket[i].next;p!=NULL;p=q.next){

                tmp =(p->data%pow)/(pow/10);

                q.data=p->data;

                q.next=p->next;

                if(tmp!=i){

                    DeleteNode(&bucket[i],p);

                    AddNode(&bucket[tmp],q.data);

                }

            }

        }

        pow\*=10;

    }

    i=0;

    for(p=bucket[0].next;p!=NULL;p=p->next)

        a[i++]=p->data;

}

void PrintArray(int a[],int n) {

    for(int i=0;i<n;i++)

        cout<<a[i]<<" ";

    cout<<endl;

}

int a[200000];

int main() {

    int n;cin>>n;

    for (int i=0;i<n;i++){

        cin>>a[i];

    }

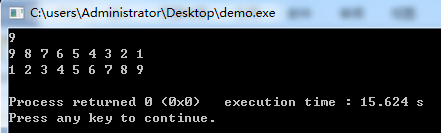
    RadixSort(a,n);

    PrintArray(a,n);

    return 0;

}

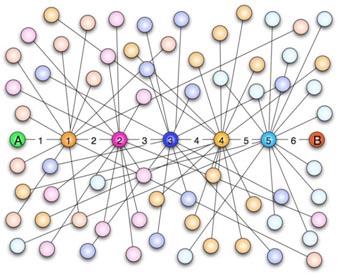
## 5 运行结果



# 题目6 六度空间

## 1 问题描述

“六度空间”理论又称作“六度分隔（Six Degrees of Separation）”理论。这个理论可以通俗地阐述为：“你和任何一个陌生人之间所间隔的人不会超过六个，也就是说，最多通过五个人你就能够认识任何一个陌生人。”如下图所示。



“六度空间”理论虽然得到广泛的认同，并且正在得到越来越多的应用。但是数十年来，试图验证这个理论始终是许多社会学家努力追求的目标。然而由于历史的原因，这样的研究具有太大的局限性和困难。随着当代人的联络主要依赖于电话、短信、微信以及因特网上即时通信等工具，能够体现社交网络关系的一手数据已经逐渐使得“六度空间”理论的验证成为可能。

假如给你一个社交网络图，请你对每个节点计算符合“六度空间”理论的结点占结点总数的百分比。

**教学内容：**综合实践广度优先搜索BFS。

**教学目标：**学会利用BFS解决实际问题。

## 2 具体设计要求

1. 输入要求

输入 第1行输入两个正整数，分别表示社交网络图的结点数N（1<N≤10​4​​，表示人数）、边数M（≤33×N，表示社交关系数）。随后的M行对应M条边，每行给出一对正整数，分别是该条边直接连通的两个结点的编号（节点从1到N编号）。

1. 输出要求

对每个结点输出与该结点距离不超过6的结点数占结点总数的百分比，精确到小数点后2位。每个结点输出一行，格式为“结点编号:（空格）百分比%”。

1. 样例

输入

10 9

1 2

2 3

3 4

4 5

5 6

6 7

7 8

8 9

9 10

输出

1: 70.00%

2: 80.00%

3: 90.00%

4: 100.00%

5: 100.00%

6: 100.00%

7: 100.00%

8: 90.00%

9: 80.00%

10: 70.00%

## 3 主要思路

利用队列还有BFS算法解决问题

## 4 源代码

#include<bits/stdc++.h>

#define MAX 1001

int count(int num,int N,int \*\*relation){

    int queue[MAX];

    int front=-1,rear=-1;

    int sum=1;

    int level=0;

    int levellast=num;

    int visit[MAX]={0};

    int last;

    visit[num]=1;

    queue[++rear]=num;

    while(front<rear){

        int  t=queue[++front];

        for(int i=1;i<=N;i++) {

            if (!visit[i] && relation[t][i] == 1) {

                queue[++rear] = i;

                visit[i] = 1;

                sum+=1;

                last=i;

            }

        }

            if(t==levellast){

                level+=1;

                levellast=last;

            }

            if(level==6) break;

        }

        return sum;

    }

int main(void){

    int num1,num2;

    scanf("%d",&num1);

    scanf("%d",&num2);

    int \*\*relation;

    relation = (int\*\*)malloc(sizeof(int\*)\*(num1+1));

    for (int i=0;i<=num1;i++){

        \*(relation+i)=(int \*)malloc(sizeof(int)\*(num1+1));

    }

    for(int i=0;i<=num2-1;i++){

        int x,y;

        scanf("%d",&x);

        scanf("%d",&y);

        relation[x][y]=relation[y][x]=1;

    }

    for(int i=1;i<=num1;i++){

        printf("%d: %.2f%%\n",i,((float)count(i,num1,relation)/num1)\*100);

    }

    return 0;

}

## 5 运行结果

