# 信息科学技术学院

# 课程（实习）设计

课程实习名称：数据结构实习

专　　业：计算机科学与技术

学　　号：2351610105

学 生 姓 名：方泽宇

成　　绩：

批 改 日 期：

指 导 教 师：

# 完成任务汇总表

| 序号 | 模块 | 任务 |
|---|---|---|
| 1 | 线性结构 | 约瑟夫环 |
| 2 | 线性结构 | 纸牌游戏 |
| 3 | 线性结构 | 一元多项式计算 |
| 4 | 线性结构 | 迷宫求解 |
| 5 | 线性结构 | 八皇后问题 |
| 6 | 线性结构 | 运动会分数统计 |
| 7 | 线性结构 | 订票系统 |
| 8 | 线性结构 | 文章编辑 |
| 9 | 树形结构 | 二叉树 |
| 10 | 树形结构 | 哈夫曼编码 |
| 11 | 树形结构 | 并查集 |
| 12 | 图形结构 | 最小代价生成树 |
| 13 | 图形结构 | 拓扑排序及关键路径 |
| 14 | 图形结构 | 交通咨询系统 |
| 15 | 查找技术 | 查找技术 |
| 16 | 排序技术 | 排序技术 |

# 目录（自动生成）



（宋体，五号字）

# 1. 约瑟夫环

**任务**

一堆猴子都有编号，编号是 1，2，3 . . . n ,这群猴子(n 个)按照 1 ~ n 的顺序围坐一 圈，从第 1 开始数，每数到第 m 个，该猴子就要离开此圈，这样依次下来，直到圈中只剩 下最后一只猴子，则该猴子为大王。请设计算法编写程序输出为大王的猴子的编号。

（1）需求分析

分析总结逻辑结构：

这是一个约瑟夫环问题，n 只猴子构成了一个环，每数到 m 只猴子就让这只猴子出圈，然后再从下一只猴子开始数，一直数到只剩下 1 只猴子。这样的问题可以使用递归进行解决，也可以不使用递归，循环数数即可。需要初始化猴子的编号，将 1-n 放在循环链表当中。由于使用头插法建表的速度比较快，在设计算法过程中可以从 n 到 1 倒序使用头插法建表。从链表的第一个结点开始，每次向前移动 m-1 个单位，并将该节点删除即可。当循环链表中只剩下一个结点的时候，说明只剩下了一只猴子，可以直接输出结果。

分析总结运算集合：

使用头插法向单链表中插入一个元素

删除单链表中的一个元素

（2）概要设计

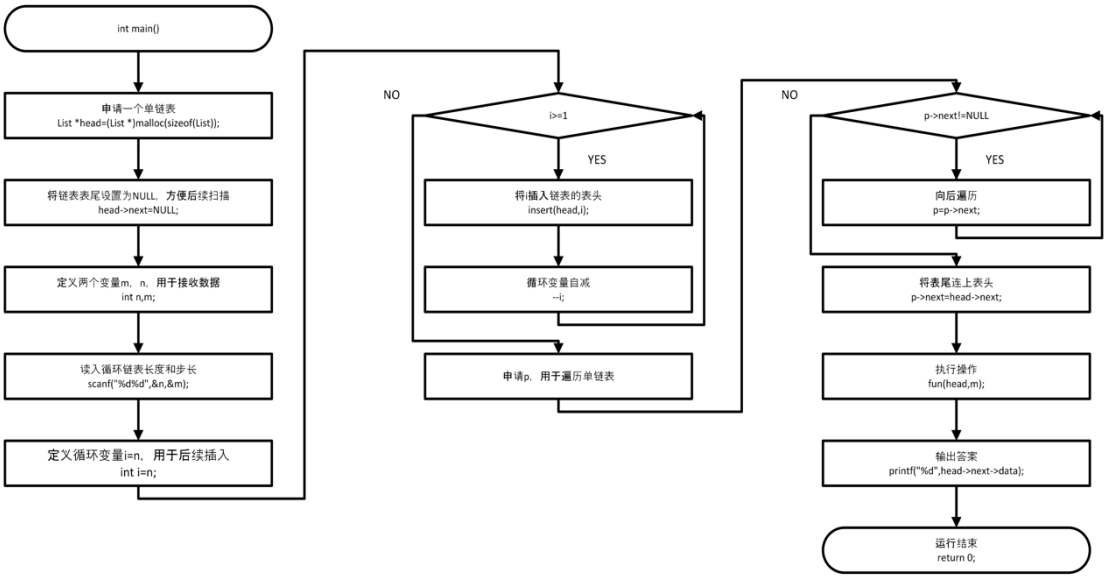
存储结构设计：

采用循环链表的数据结构，让 head 指针能够指向第一支猴子的位置，完成建表以后，将单链表最后的 NULL 指针改成指向头节点的指针。在删除过程中如果删除了头节点，那么就需要将 head 指针的指向发生改变。需要设计至少 2 个链表的操作函数，insert(*L,x)使用头插法将 x 插入链表，fun(*L,m)执行删除操作，直到链表中只剩下一个元素。

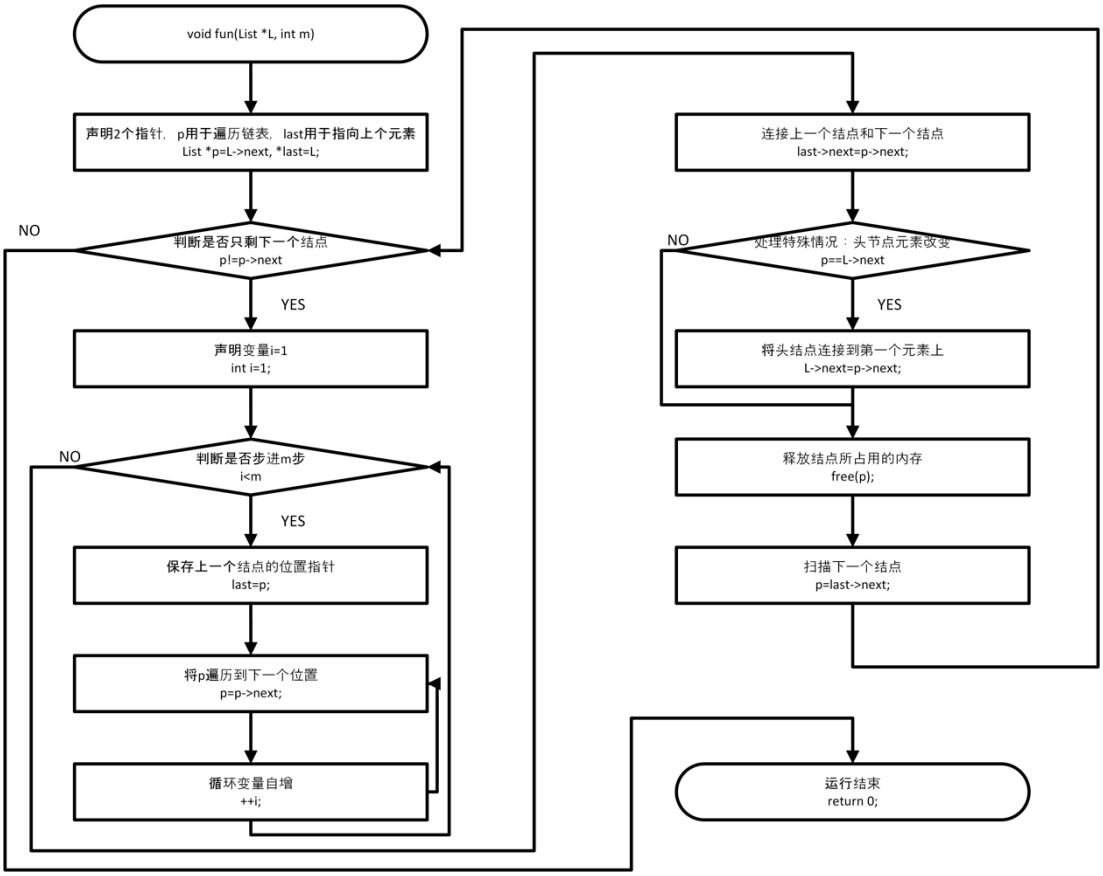
算法设计（流程图）：

采用循环链表的方法，每次经过 m-1 个结点，删除掉这个结点即可。

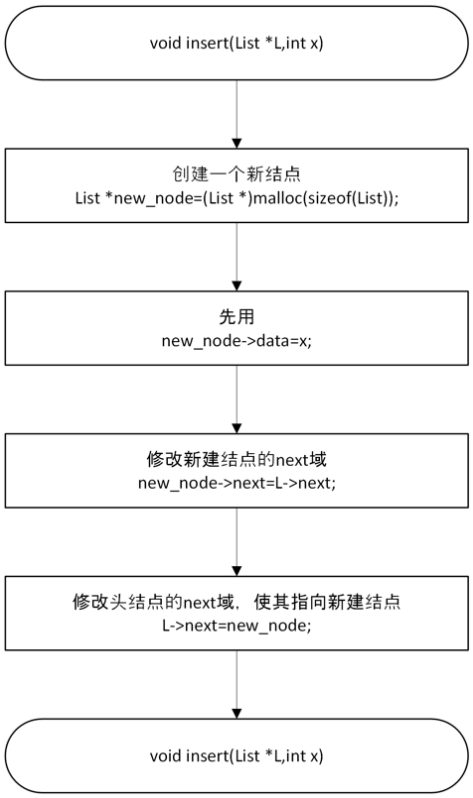
Int main()的函数实现：




Void fun(List *L,int m)的函数实现：




Insert(List *L,int x)的函数实现：




（3）详细设计

源代码（注释）

```
#include<stdio.h>
#include<stdlib.h>//因为要使用 malloc，所以需要 include<stdlib.h>
typedef struct List
{
    int data;
    struct List *next;
}List;//创建链表的结点结构题
void insert(List *L,int x)//使用头插法插入数字 x
{
    List *new_node=(List *)malloc(sizeof(List));//申请一个空间，插入链表的表头
    new_node->data=x;//先用
    new_node->next=L->next;//后改
    L->next=new_node;//修改头指针指向的位置
}
void fun(List *L, int m)
{
    List *p=L->next, *last=L;
    while (p!=p->next)// 仅剩一个节点时停止
    {
        for (int i = 1; i < m; ++i)
        {
            last=p;//保存上一个结点的位置，方便后面修改 next 指针
            p=p->next;//向后扫描一个结点
        }
        last->next=p->next;
        if(p==L->next) L->next=p->next;//处理一个特殊情况，如果表头所指向的元素发生改变
        free(p);//释放内存，节省内存占用
        p=last->next;//下一个结点
    }
}
int main()
{
    List *head=(List *)malloc(sizeof(List));//申请一个单链表
    head->next=NULL;//将头节点的 next 域赋值为 NULL，方便后面扫描单链表的表尾
    int n,m;
    scanf("%d%d",&n,&m);
    for(int i=n;i>=1;--i) insert(head,i);//倒着将数据使用头插法将数据插入单链表
    List *p=head->next;//遍历单链表
    while(p->next!=NULL) p=p->next;//访问链表中的最后一个元素
    p->next=head->next;//将单链表的尾结点连上头节点所连接的结点，创建循环链表
    fun(head,m);//执行删除操作
    printf("%d",head->next->data);//输出结果
    return 0;
```

}

（4）调试分析

样例 1:

输入 10，3，输出 4

```
10 3
4%
```

以下是对样例 1 输出结果的分析：

第一次删除：从编号为 1 的猴子开始数，删除第 3 只猴子。即删除编号为 3 的猴子。剩下的猴子是：1，2，4，5，6，7，8，9，10。

第二次删除：从编号为 4 的猴子开始数，再数到第 3 只猴子，删除编号为 6 的猴子。剩下的猴子是：1，2，4，5，7，8，9，10。

第三次删除：从编号为 7 的猴子开始数，删除编号为 9 的猴子。剩下的猴子是：1，2，4，5，7，8，10。

第四次删除：从编号为 10 的猴子开始数，删除编号为 2 的猴子。剩下的猴子是：1，4，5，7，8，10。

第五次删除：从编号为 4 的猴子开始数，删除编号为 7 的猴子。剩下的猴子是：1，4，5，8，10。

第六次删除：从编号为 8 的猴子开始数，删除编号为 1 的猴子。剩下的猴子是：4，5，8，10。

第七次删除：从编号为 4 的猴子开始数，删除编号为 8 的猴子。剩下的猴子是：4，5，10。

第八次删除：从编号为 10 的猴子开始数，删除编号为 5 的猴子。剩下的猴子是：4，10。

第九次删除：从编号为 10 的猴子开始数，删除编号为 10 的猴子。剩下的猴子是：4。

样例 2:

输入 30 7，输出 23

```
30 7
23%
```

（5）小结

在本次约瑟夫环实习的过程中，关于仅剩一个结点的判断，我进行了很长时间的调试，我尝试过很多判断条件均没有用。后来我尝试手动更改循环，发现问题出现在如果 head 指向的元素被删除，head->next 没有被及时更新，于是对代码进行了修改。

# 2. 纸牌游戏

**任务**

编号为 1-52 张牌，正面向上，从第 2 张开始，以 2 为基数，是 2 的倍数的牌翻一次， 直到最后一张牌;然后，从第 3 张开始，以 3 为基数，是 3 的倍数的牌翻一次，直到最后一张牌;然后从第 4 张开始，以 4 为基数，是 4 的倍数的牌翻一次，直到最后一张牌;.....直 到以 52 为基数的牌翻过，这时正面向上的牌有哪些?请设计算法编写程序输出最终正面向 上的纸牌的编号。

（1）需求分析

分析总结逻辑结构：

这是一个翻牌问题，因为一张牌有 2 面，如果一张牌被翻了奇数次，那么这张牌的状态和初始状态不一致，如果一张牌被翻了偶数次，那么这张牌的状态会和初始状态相一致。所以我们只需要依次扫描这些牌，统计被翻牌的次数即可，如果被翻了奇数次，那么这些牌是反面朝上，如果被翻了偶数次，那么就是反面朝上。初始化一个顺序表，清空顺序表中的数据项，从 2 到 52 依次将 n 到倍数翻牌，最后统计被翻牌的次数即可。

分析总结运算集合：

初始化顺序表(数组)

翻牌(将该数组元素+1)

判断该牌被翻的次数是否是偶数

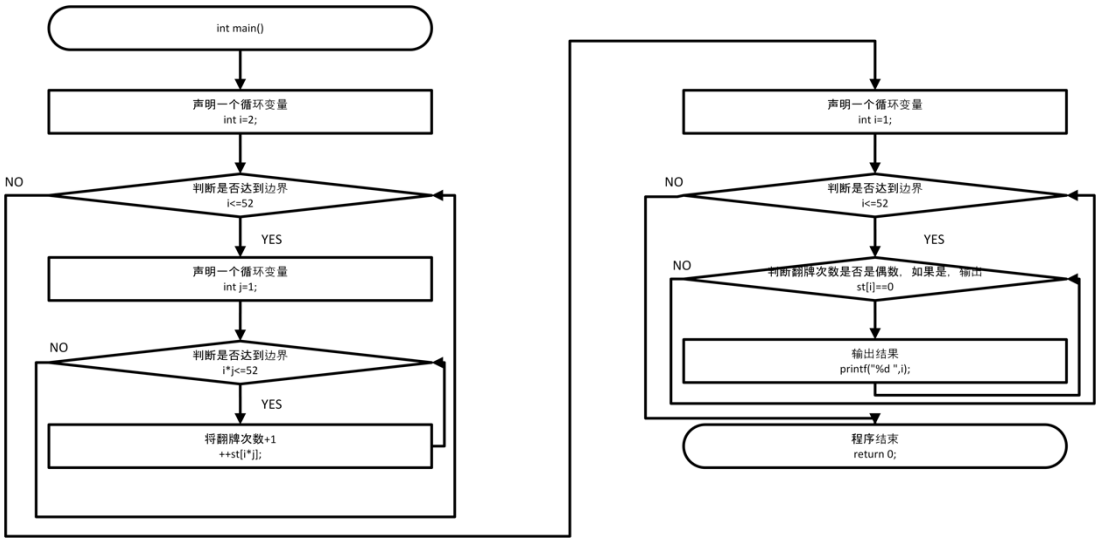
（2）概要设计

存储结构设计

因为需要很多次随机访问，为加快程序运行速度，采用顺序表的方式存储。

算法设计（流程图）

Int mian()的函数实现：



（3）详细设计

源代码（注释）

```
#include<stdio.h>
int st[53];
int main()
{
    for(int i=2;i<=52;++i)//以 1-52 为基数
        for(int j=1;i*j<=52;++j)//遇到基数的整数倍就翻牌一次
            ++st[i*j];//记录翻牌的次数
    for(int i=1;i<=52;++i)//检查被翻牌的牌
        if(st[i]%2==0)//如果被翻了偶数次，那么还是正面朝上
            printf("%d ",i);//输出结果
    return 0;//程序结束
}
```

（4）调试分析

输出结果：1 4 9 16 25 36 49

```
1 4 9 16 25 36 49
```

以下是对输出结果的分析：

1 没有被翻牌，所以正面朝上

4 被 2 和 4 翻牌 2 次，所以正面朝上

9 被 3 和 9 翻牌 2 次，所以正面朝上

16 被 2、4、8、16 翻牌 4 次，所以正面朝上

25 被 5 和 25 翻牌 2 次，所以正面朝上

36 被 2、3、4、6、9、12、18、36 翻牌 8 次，所以正面朝上

49 被 7 和 49 翻牌 2 次，所以正面朝上

类似地，

所有的质数均只会被自己翻牌 1 次，所以反面朝上

8 被 2、4、8 翻牌 3 次，所以反面朝上

10 被 2、5、10 翻牌 3 次，所以反面朝上

12 被 2、3、4、6、12 翻牌 5 次，所以反面朝上

等等……

因此，只有完全平方数的翻牌次数为偶数次，因此输出结果均为完全平方数

（5）小结

在本次实习中，我们使用顺序表来存储纸牌被翻牌的次数和状态，并使用随机访问的形式来更新每一个结点，最后顺序访问输出结果。通过对结果的观察不难发现，所有的答案均为完全平方数，因此我们可以直接输出所有的完全平方数，也可以得到本题的答案。

# 3. 一元多项式计算

**任务**

设计合适的存储结构，完成一元多项式的相关运算。

要求:(1)能够按照指数降序排列建立并输出多项式;(2)能够完成两个多项式的相加、相 减，并将结果输出。

（1）需求分析

分析总结逻辑结构：

题目中所说到的降序排序，其实是按照降序排序的规则，对一个表达式中变量次数的排序。因为这道题中经常会用到对线性表的插入和删除的操作，所以采用单链表的形式存储会更加便捷。

分析总结运算集合：

初始化线性表(链表)

基于值查找，并插入链表元素

将两个有序链表归并

对串的处理：从串中分离指数和系数

（2）概要设计

存储结构设计：

因为需要多次在线性表中插入元素，所以采用链式存储结构

算法设计（流程图）：

初始化结果链表：创建一个空的结果链表 result，用来存储两个多项式相加的结果。

定义指针遍历链表：pa 指向第一个多项式链表 A 的头节点，pb 指向第二个多项式链表 B 的头节点。这两个指针用来遍历两个多项式链表。

遍历两个链表：使用 while 循环同时遍历两个链表 A 和 B。在每次循环中，比较 pa 和 pb 所指向节点的指数 power 大小，分以下几种情况处理：情况 1：pa->power > pb->power。如果 A 中当前节点的指数大于 B 中当前节点的指数，将 A 中当前节点插入结果链表 result 中。然后，pa 指针向后移动，指向下一个节点。情况 2：pa->power < pb->power 如果 B 中当前节点的指数大于 A 中当前节点的指数，将 B 中当前节点插入结果链表 result 中。然后，pb 指针向后移动，指向下一个节点。情况 3：pa->power == pb->power 如果 A 和 B 中当前节点的指数相同，则将它们的系数相加，生成一个新的节点插入结果链表 result 中。然后同时移动 pa 和 pb 指针，分别指向下一个节点。

处理剩余项：当一个链表遍历结束后（pa 或 pb 为 NULL），继续处理另一个链表中的剩余项：将剩余的 pa 链表中的项插入结果链表 result。将剩余的 pb 链表中的项插入结果链表 result。

返回结果链表：addPoly 函数返回存储加法结果的链表 result。

（流程图实在太复杂了，这题就不画了，因为实习要求是画 4 个题目）

（3）详细设计

源代码（注释）

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
typedef struct PolyNode
{
    int factor;//系数
    int power;//指数
    struct PolyNode *next;
    //next 指针，指向下一个结点
}PolyNode, *PolyList;
//创建结点类型和结点指针类型
void insertTerm(PolyList *L, int factor, int power)
{
    PolyNode *newNode = (PolyNode *)malloc(sizeof(PolyNode));
    //创建一个新的结点
    newNode->factor = factor;
    //传入新结点的系数
    newNode->power = power;
    //传入新结点的指数
    newNode->next = NULL;
    //新建的结点没有下一个指针位置
    if (*L==NULL||(*L)->power<power)
    //如果说发现需要插入的链表为空，或者是新插入的这个结点的次数高于原有的最高次数
    {
        newNode->next=*L;
        //那么直接将该结点插入头节点即可
        *L=newNode;
        //修改头指针的位置
    }
    else
    {
        PolyNode *p=*L;
        //新建一个 p 指针，用于遍历链表
        while(p->next&&p->next->power>power) p=p->next;
        //扫描链表，直到次数相等或者次数刚好大于这个点
        if (p->power == power)
        //如果恰巧停下来的时候，次数相等，那么合并
        {
            p->factor+=factor;
            //将系数直接相加
            free(newNode);
            //释放这个已经合并同类项的点
            if (p->factor == 0)
            //如果次数干好为 0，那么把合并以后的点也直接释放掉
```

```
            {
                PolyNode *temp = p->next;
                //删除操作，先保存后面一个结点的指针，先连接后删除
                free(p);
                *L = temp;
            }
        }
        else
        {
            newNode->next = p->next;
            //否则，将新的结点插入链表
            p->next = newNode;
        }
    }
}
void print(PolyList L)
//这个函数用于输出表达式
{
    if (L == NULL)
    //如果表达式保存的链表为空，说明表达式不存在，直接输出 0 即可
    {
        printf("0\n");
        return;
    }
    PolyNode *p = L;
    //声明一个指针类型 p，用于遍历链表
    while (p)
    {
        if (p->factor>0&&p!=L) printf("+");
        //在非头节点的情况底下，如果系数大于 0，那么需要输出正号
        if (p->power==0) printf("%d",p->factor);
        //如果次数正号为 0，那么直接输出系数即可
        else if(p->power==1&&p->factor==1) printf("x");
        //如果系数和次数都恰好为 1，那么直接输出 x
        else if(p->power==1&&p->factor==-1) printf("-x");
        //如果次数为 1，系数为-1，那么直接输出-x
        else if(p->power==1&&p->factor!=1&&p->factor!=-1)
            printf("%dx",p->factor);
            //如果次数恰好为 1，系数无特殊输出系数和 x
        else if(p->factor==1) printf("x%d", p->power);
        //如果系数恰好为 1，且次数无特殊，那么输出 x 和次数
        else if(p->factor==-1) printf("-x%d",p->power);
        //如果系数为-1，次数无特殊，那么输出-x 和次数
        else printf("%dx%d", p->factor, p->power);
```

```
        //否则，输出系数、x、次数
        p = p->next;
        //遍历下一个结点
    }
    printf("\n");
    //输出换行，输出完成
}
PolyList addPoly(PolyList A, PolyList B)
//新建加法运算
{
    PolyList result=NULL;
    //首先先清空答案 result 链表
    PolyNode *pa=A,*pb=B;
    //创建两个指针类型变量 pa，pb，用于遍历两个链表
    while (pa&&pb)
    //如果在 pa 和 pb 都没有走到头掉情况底下，需要比较两个链表的次数
    {
        if (pa->power>pb->power)
        //如果说 a 链表的次数大于 b 链表的次数，那么需要先计算 a 链表里值
        {
            insertTerm(&result,pa->factor,pa->power);
            //调用 insert 函数将 a 链表中 pa 指向的系数和次数插入 result 链表中
            pa=pa->next;
            //pa 走向下一个结点
        }
        else if(pa->power<pb->power)
        //同样，如果说 b 链表的次数大于 a 链表的次数，那么需要先计算 b 链表里值
        {
            insertTerm(&result,pb->factor,pb->power);
            //调用 insert 函数将 b 链表中 pb 指向的系数和次数插入 result 链表中
            pb=pb->next;
            //pb 走向下一个结点
        }
        else
        //最后只剩下了两者次数相等的情况
        {
            int sum=pa->factor+pb->factor;
            //这种情况下，把两个系数相加即可
            if(sum!=0) insertTerm(&result,sum,pa->power);
            //如果说相加得到的结果为 0，也没有必要自己给自己找麻烦，直接跳过插入操作
            //如果相加得到的结果不为 0，那么直接将这样一个系数和次数插入到 result 链表当中
            pa=pa->next;
            pb=pb->next;
            //两个指针移向下位
```

```
        }
    }
    while (pa)
    //收尾，如果 pb 已经走到了链表的表尾，将 pa 走到底就行了
    {
        insertTerm(&result,pa->factor,pa->power);
        pa=pa->next;
    }
    while (pb)
    //收尾，如果 pa 已经走到了链表的表尾，将 pb 走到底就行了
    {
        insertTerm(&result,pb->factor,pb->power);
        pb=pb->next;
    }
    return result;
    //把答案返回给主调函数
}
PolyList subPoly(PolyList A,PolyList B)
//这是一个减法的过程，实现原理和上面是类似的
{
    PolyList result=NULL;
    //首先定义一个结果链表，这个链表原始是空的，因为里面什么也没有
    PolyNode *pa=A,*pb=B;
    //定义两个用于循环的指针 pa 和 pb
    while (pa&&pb)
    //如果指向两个链表的指针都没有走到表尾
    {
        if (pa->power>pb->power)
        //如果 pa 指向的结点的次数高于 pb 指向结点的次数
        {
            insertTerm(&result,pa->factor,pa->power);
            //将 pa 的结点直接插入链表中即可
            pa = pa->next;
            //pa 走向下一个结点
        }
        else if (pa->power<pb->power)
        //相反的，如果 pb 指向的结点的次数高于 pa 指向的结点的次数
        {
            insertTerm(&result,-pb->factor,pb->power);
            //那么 pb 指向的结点，取相反数以后插入 result 链表中即可
            //注意，这里和上面的加法不一样，因为 b 是被减数，所以需要取负数
            pb = pb->next;
            //pb 走向下一个结点
        }
```

```
        else
        //最后只剩下了一种情况，就是两者的次数恰好相等
        {
            int ans=pa->factor-pb->factor;
            //那么将两者的系数相减
            if(ans!=0) insertTerm(&result,ans,pa->power);
            //如果两者系数相减正号为 0，也没必要白费功夫了，直接跳过即可
            //将结点插入 result 当中
            pa=pa->next;
            pb=pb->next;
            //pa，pb 均可以走向下一个结点
        }
    }
    while (pa)//收尾工作
    {
        insertTerm(&result,pa->factor,pa->power);
        pa = pa->next;
        //将 pa 剩下的结点全部插入到 result 中
    }
    while (pb)
    {
        insertTerm(&result,-pb->factor,pb->power);
        pb=pb->next;
        //将 pb 剩下的结点全部插入道 result 中
    }
    return result;
    //将结果返回给主调函数
}
void add(PolyList *L)
//这是一个读入字符串分离字符建立链表的字符串操作函数
{
    char ch[100];
    //本程序可以接受的最大表达式长度为 100
    scanf("%s",&ch);
    //将字符串读入程序中
    int factor=0,sign=1,power=0;
    //定义三个变量，factor 用于指示系数
    //sign 用于指示符号
    //power 用于指示次数
    for(int i=0;i<strlen(ch);++i)
    //依次遍历整个字符串
    {
        if(ch[i]=='-')
        //如果遇到了负号，将系数取相反数
```

```
        {
            sign=-1;
            continue;
        }
        else if(ch[i]=='+') continue;
        //如果遇到了加号，说明不需要处理
        else if(ch[i]=='x')
        //如果遇到了 x，那就说明遇到了系数和次数的分界点
        {
            if(factor==0) factor=1;
            //如果系数为 0，那就说明 x 前面什么都没有，那系数就是 1
            for(++i;ch[i]!='+'&&ch[i]!='-';++i)
            //通过一个循环读取次数
            {
                power*=10;
                power+=ch[i]-'0';
            }
            if(power==0&&(ch[i]=='+'||ch[i]=='-')) power=1;
            //如果还没读就结束了，说明次数也是 1
            insertTerm(L,sign*factor,power);
            //将这个结点插入道链表当中
            --i;
            //多读了一个符号位，说不定后面需要特殊处理，返回去重新处理
            power=0;factor=0;sign=1;//将标志还原
        }
        else
        {
            factor*=10;
            factor+=ch[i]-'0';
            //这两个代码用来读取系数
        }
    }
    if(factor!=0) insertTerm(L,factor*sign,0);
    //收尾工作，如果最后没有读到 x，那就说明存在 0 次方项，将 0 次方项插入链表
}
int main()
{
    PolyList A=NULL;//新建两个链表
    PolyList B=NULL;
    add(&A);//读入 A
    add(&B);
    PolyList C = addPoly(A,B);//C 是答案，AB 相加
    printf("A+B:");
    print(C);
```

```
    C=subPoly(A,B);//C 是答案，AB 相减
    printf("A-B:");
    print(C);
    return 0;//程序运行结束
}
```

（4）调试分析

样例 1 输入：3x3+2x2+1，3x3+4

输出：A+B:6x3+2x2+5，A-B:2x2-3

样例分析：在第一个表达式中，x^3 项为 3，与第二个表达式相一致，所以在结果中，没有 x^3 项。而其他项均存在，显然输出答案为 6x3+2x2+5，A-B:2x2-3

```
3x3+2x2+1
3x3+4
A+B:6x3+2x2+5
A-B:2x2-3
```

样例 2 输入：0，0

输出：A+B:0，A-B:0

```
0 0
A+B:0
A-B:0
```

样例 3 输入：0，-3x5-6

输出：A+B:3x5+6，A-B:-3x5-6

```
0
-3x5-6
A+B:-3x5-6
A-B:3x5+6
```

（6）小结

在本次实习过程中，我遇到了很多的困难，首先是在字符串处理方面。需要从段字符串文本中分离出系数、x 和次数，然后再插入链表中进行运算。通过这次实习，我更加深入理解了链表的链式存储的原理，以及两个有序链表归并的问题。

# 4. 迷宫求解

**任务**

（1）需求分析

分析总结逻辑结构：

采取数组坐标系，从入口(1,1)走到出口(n,m)所可能走过的路径。可以通过搜索和遍历来解决这样一个问题。遍历有 2 种常见的方式，一种是深度优先搜索(DFS，Depth First Search)，另一种是广度优先搜索(BFS,Breadth First Search)。前者是基于栈来实现的，后者是基于队列来实现的。为了在这道题中体现栈和队列两种数据结构，本题采用广度优先搜索遍历的方式搜索，其特点是可以搜索到从起点到终点的最短路径。回溯路径的方法是，到达终点之后逐个返回寻找路径即可。

分析总结运算集合：

初始化队列

将数据从队尾加入队列

从队首取出元素，并删除该元素

初始化栈

将数据加入栈，并更新栈顶

将栈顶元素去除，并删除栈顶元素

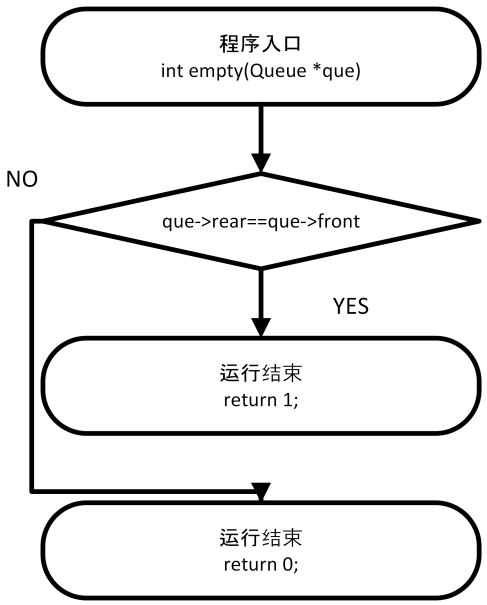
（2）概要设计

存储结构设计：

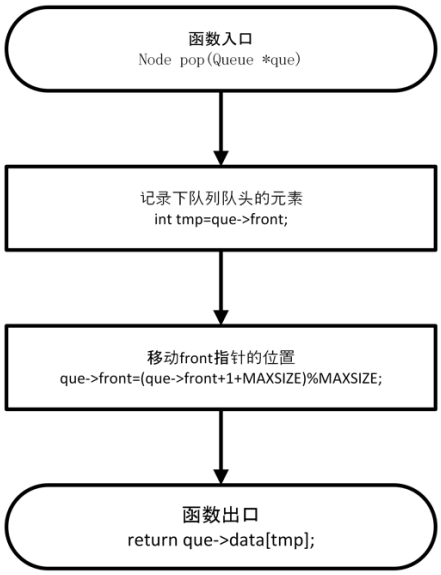
采用队列和栈两种数据结构来完成运算，通过队列完成广度优先搜索(BFS)的搜索过程，通过栈对答案进行逆序输出。现将终点入栈，再依次将所经过的点加入栈中，然后再逐个出栈输出。

算法设计（流程图）：

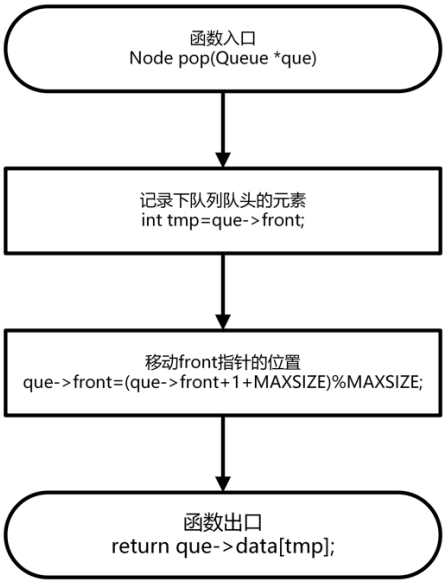
Int empty(Queue *que)的函数实现：



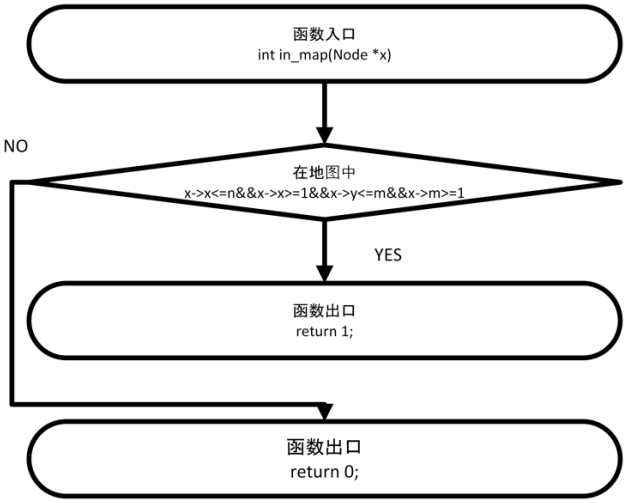
Node pop(Queue *que)的函数实现：



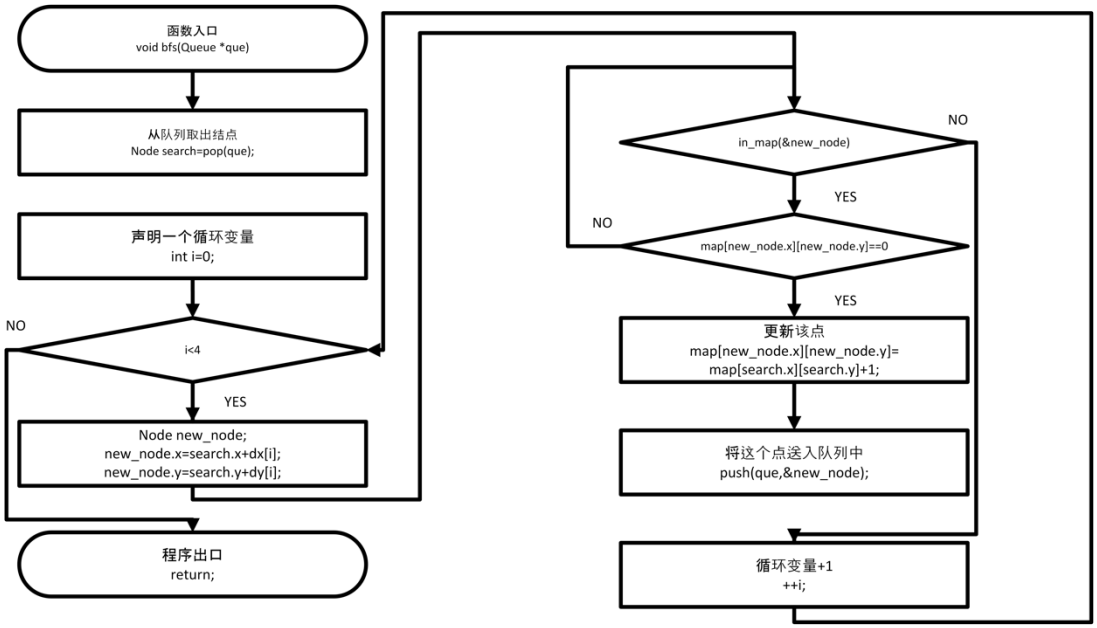

Void push(Queue *que,Node *x)的函数实现：



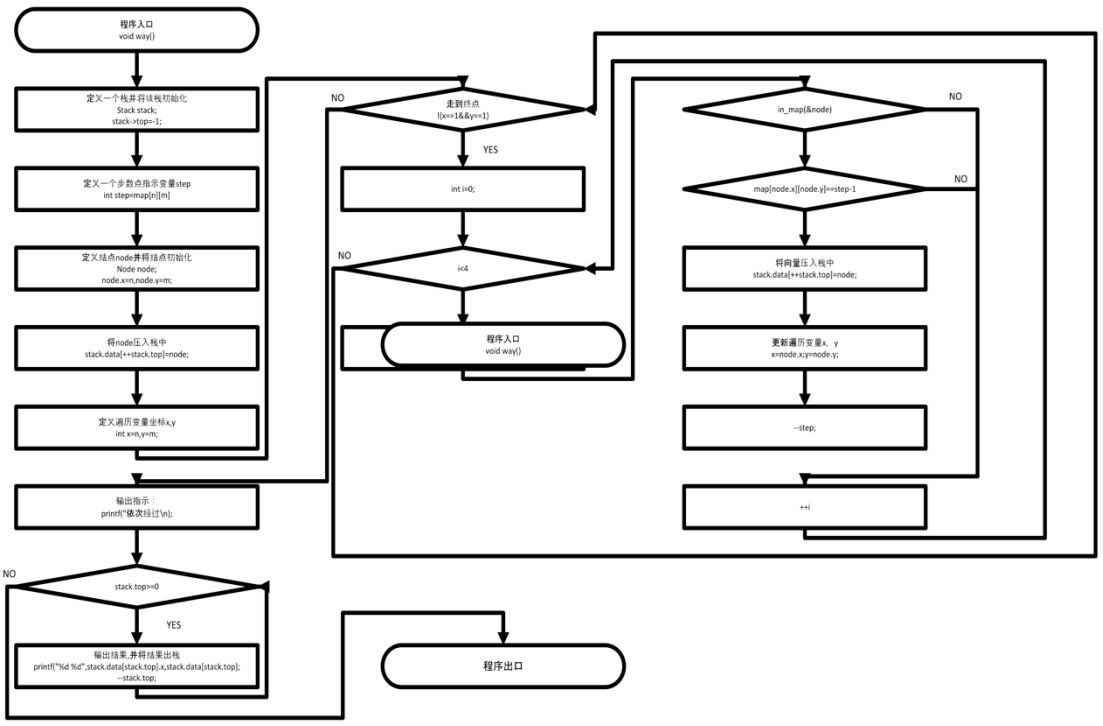

Int in_map(Node *x)的函数实现：



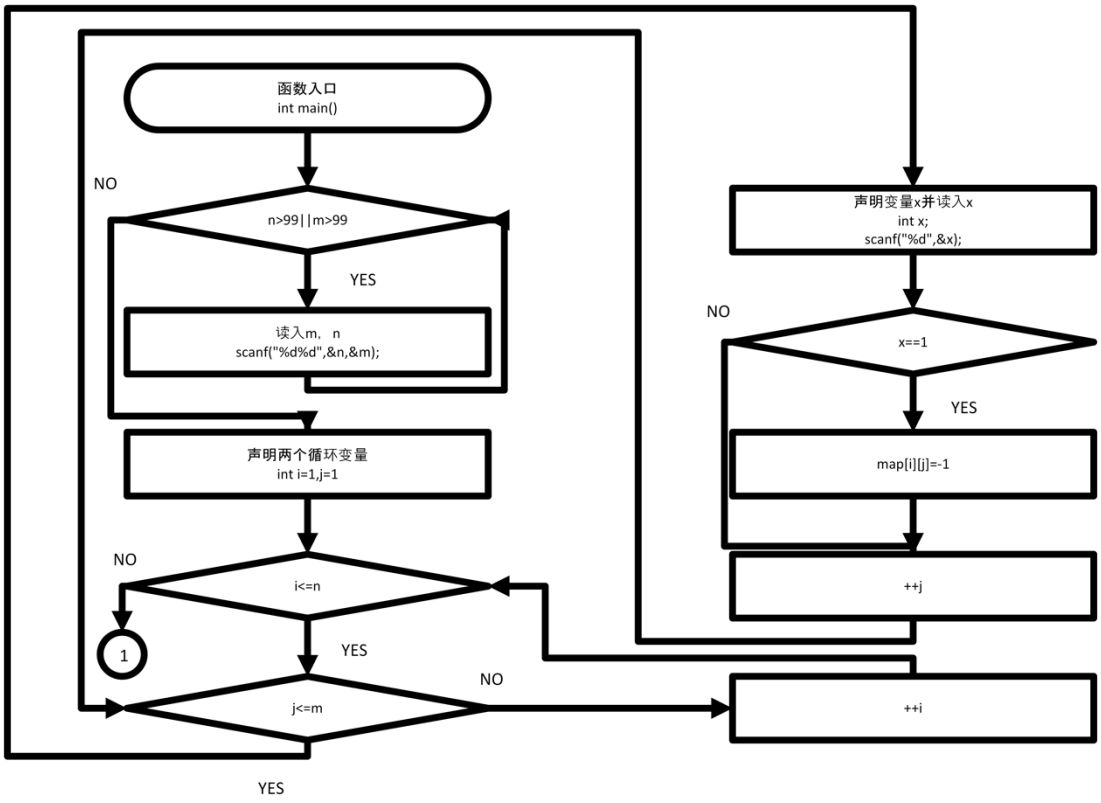
Void bfs(Queue *que)的函数实现：

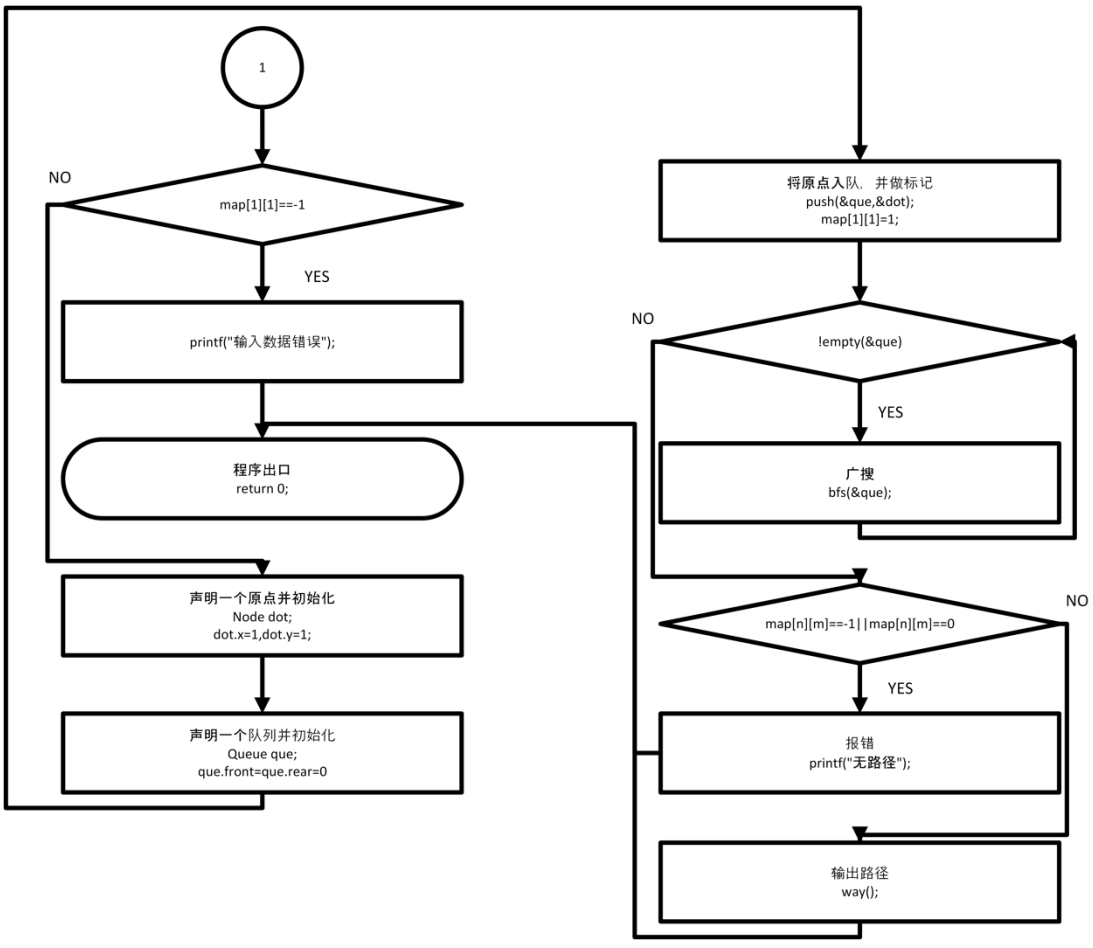



Void way()的函数实现：



Int main()的函数实现：



（3）详细设计

源代码（注释）

```
#include<stdio.h>
const int MAXSIZE=100;
//用一个全局的常量去定义地图的最大大小
int map[MAXSIZE][MAXSIZE],n=0x3f3f3f3f,m=0x3f3f3f3f;
//定义一个地图，最大的大小为 100*100
const int dx[4]={0,0,1,-1};
const int dy[4]={1,-1,0,0};
//定义两个向量，方便以后想下一步走
typedef struct Node
{
    int x,y;
}Node;
//为每一个点的坐标定义一个结点类型
typedef struct Queue
{
    Node data[MAXSIZE*MAXSIZE];
    int front,rear;
}Queue;
//定义一个队列，用于存放广度优先搜索(bfs)
typedef struct Stack
{
    Node data[MAXSIZE*MAXSIZE];
    int top;
}Stack;
//定义一个栈，用于逆序输出路径
int empty(Queue *que)
{
    if(que->rear==que->front) return 1;
    else return 0;
}
//判断队列是否为空(用处：如果队列为空说明 bfs 结束)
Node pop(Queue *que)
{
    int tmp=que->front;
    //保存临时变量，指向需要输出的结点的位置
    que->front=(que->front+1)%MAXSIZE;
    //front 指针向前移动
    return que->data[tmp];
}
//将队列中的最前面的一个元素返回，并且删除掉这个元素
void push(Queue *que,Node *x)
{
```

```
    que->data[que->rear].x=x->x;
    que->data[que->rear].y=x->y;
    //将结点的两个域分别更新
    que->rear=(que->rear+1)%MAXSIZE;
    //rear 指针后移
}
int in_map(Node *x)
{
    if(x->x<=n&&x->x>=1&&x->y<=m&&x->y>=1) return 1;
    //如果没有超出地图边界，那么返回 1，否则返回 0
    else return 0;
}
void bfs(Queue *que)
//这是一个广度优先搜索遍历的算法，当然我们也可以对这张图进行深度优先搜索
//但是 bfs 搜索的是最短路径，这道题我只输出一种最短路径
{
    Node search=pop(que);
    //从队列里取出一个结点
    for(int i=0;i<4;++i)
    {
        Node new_node;
        new_node.x=search.x+dx[i];
        new_node.y=search.y+dy[i];
        //使用向量，更新待搜索的结点
        if(in_map(&new_node))
        //如果没有超出边界
        {
            if(map[new_node.x][new_node.y]!=0) continue;
            //如果这个地方是障碍物，或者已经更新过到达这个点的最短路径，那么直接 continue
            map[new_node.x][new_node.y]=map[search.x][search.y]+1;
            //如果满足要求，则这个点到(1,1)点的最短路径就是其上一个结点+1
            push(que,&new_node);
            //将这个结点送入队列中
        }
    }
}
void print()
{
    for(int i=1;i<=n;++i)
    {
        for(int j=1;j<=m;++j)
        {
            printf("%d ",map[i][j]);
            //依次输出每一个点到(1,1)的最短路，输出-1 代表该点无法到达(1,1)
```

```
        }
        printf("\n");
    }
}
void way()
{
    Stack stack;
    //定义一个栈
    //因为 bfs 到终点以后，可以倒着推出路径，但是正着无法推出，所以需要用一个栈来将结果逆序
    stack.top=-1;
    //初始化栈
    int step=map[n][m];
    //将最终步数放入临时变量 step 中，上一次走过的结点的步数一定是 step-1
    Node node;
    //创建一个新的结点
    node.x=n;node.y=m;
    stack.data[++stack.top]=node;
    //将结果放入栈中
    int x=n,y=m;
    while(!(x==1&&y==1))
    {
        for(int i=0;i<4;++i)
        {
            node.x=x+dx[i];
            node.y=y+dy[i];
            //通过向量更新出上一步有可能走过的点
            if(in_map(&node))
            //这个点是否在地图内
            {
                if(map[node.x][node.y]==step-1)
                //如果步数正好是 step-1，则说明这是路径上的一点
                {
                    stack.data[++stack.top]=node;
                    //将这个点加入栈
                    x=node.x;
                    y=node.y;
                    --step;
                    //循环，继续寻找上一个结点
                    break;
                }
            }
        }
    }
    printf("依次经过\n");
```

```
    //输出结果
    while(stack.top>=0)
    //依次出栈
    {
        printf("%d %d\n",stack.data[stack.top].x,stack.data[stack.top].y);
        --stack.top;
    }
}
int main()
{
    while(n>99||m>99) scanf("%d%d",&n,&m);
    //输入地图的大小，保证输入的大小是在一个合法的范围内，防止超出地图的边界
    for(int i=1;i<=n;++i)
    {
        for(int j=1;j<=m;++j)
        {
            int x;
            scanf("%d",&x);
            if(x==1) map[i][j]=-1;
            //输入地图，0 代表可以走，1 代表是墙壁，但是 1 和第一步冲突，为了防止冲突，读入数
据以后改成-1，代表走不了
        }
    }
    if(map[1][1]==-1)
    //如果起点为-1，墙壁
    {
        printf("输入数据错误");
        //出错返回
        return 0;
    }
    Node dot;
    //新建一个点，这个点是原点
    Queue que;
    //新建一个队列，用于存储 bfs 序列
    que.front=que.rear=0;
    //初始化队列，方式 segmentation fault
    dot.x=dot.y=1;
    push(&que,&dot);
    //将原点推入队列中
    map[1][1]=1;
    while(!empty(&que)) bfs(&que);
    if(map[n][m]==-1||map[n][m]==0)
    {
        printf("无路径\n");
```

```
            //最后一个点无法到达，出错返回
            return 0;
        }
        //print();
        way();
        return 0;
}
```

（4）调试分析

样例 1:

输入：

```
4 5
0 1 1 1 1
0 0 1 1 1
0 0 1 1 1
1 0 0 0 0
```

输出：

```
依次经过
1 1
2 1
3 1
3 2
4 2
4 3
4 4
4 5
```

```
4 5
0 1 1 1 1
0 0 1 1 1
0 0 1 1 1
1 0 0 0 0
依次经过
1 1
2 1
3 1
3 2
4 2
4 3
4 4
4 5
```

解释：通过 bfs 扩展得到的结果如下：

```
1 -1 -1 -1 -1
2  3 -1 -1 -1
3  4 -1 -1 -1
-1 5  6  7  8
```

其中-1 表示无法到达的点，正整数表示从坐标(1,1)走到该点所需要的步数(设走到点(1,1)需要 1 步)

样例 2:

输入：

9 6
0 0 0 0 0 0
1 1 1 1 1 0
0 0 0 0 0 0
0 1 1 1 1 1
0 0 0 0 0 0
1 1 1 1 1 0
0 0 0 0 0 0
0 1 1 1 1 1
0 0 0 0 0 0

输出：

依次经过

1 1，1 2，1 3，1 4，1 5，1 6，2 6，3 6，3 5，3 4，3 3，3 2，3 1，4 1，5 1，5 2，5 3，5 4，5 5，5 6，6 6，7 6，7 5，7 4，7 3，7 2，7 1，8 1，9 1，9 2，9 3，9 4，9 5，9 6

（为排版方便，将左右的换行改成了逗号”,”）

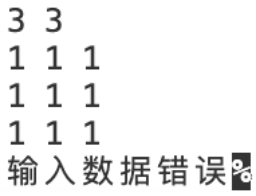
样例 3(错误样例 1):

输入：

3 3
1 1 1
1 1 1
1 1 1

输出：

输入数据错误




样例 4(错误样例 2):

输入：

5 4
0 1 1 1
0 0 1 1
1 1 0 1
1 1 1 1
1 1 0 0

输出：

无路径

```
5 4
0 1 1 1
0 0 1 1
1 1 0 1
1 1 1 1
1 1 0 0
无路径
```

（5）小结

在本次实习的过程中，我采用 BFS 广度优先搜索的方法遍历图，并逆序将结果输出到一个栈中，再利用栈先进后出的特性输出了迷宫所走过的路径。其中加上了异常处理的程序。一开始程序运行中出现段错误(segmentation fault)，后来经过仔细的检查发现错误出现在队列没有被初始化上，把队列初始化后这样的错误就没有再次发生。这段代码只能找到从坐标原点到右下角的最短路径，并不能输出所有的路径，并且时间复杂度较高。为了输出所有的路径，我们可以尝试改用基于栈/递归的 DFS 深度优先搜索算法，基于堆优化的启发式搜索 A*搜索算法，这段代码仍然有很多改进的空间。

# 5. 八皇后问题

**任务**

国际西洋棋棋手马克斯·贝瑟尔于 1848 年提出在 8X8 格的国际象棋上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种 摆法。请设计算法编写程序解决。

（1）需求分析

分析总结逻辑结构：

这道题需要在一个 8x8 的地图上生成 8 个皇后，让他在每一行，每一列和每一个对角线上均有且只有一个皇后，对于这样的一个问题，我们可以对 64 个位置上生成 8 个皇后，并且一次判断。但是这样做的话时间复杂度会来到 2 的 64 次方的级别，显然超出了计算机最大的计算规模。因此需要对这样的一个计算进行剪枝。其中一个剪枝的方法，就是首先生成一个长度为 8 的全排列，分别表示在每一列上皇后所在的行数，这样可以在生成的过程中就完成了对不在同一行、不在同一列上的判断，完成了剪枝。而生成全排列的算法，就是 dfs。生成好全排列以后，只需要对对角线进行判断。通过观察我们发现，在同一主对角线上的元素，其横纵坐标的差值是一定的，而在同一副对角线上的元素，其横纵坐标之和是一定的。因此我们只需要对横纵坐标的和、差进行判重即可。

分析总结运算集合：

Dfs 生成全排列

计算和差进行判重

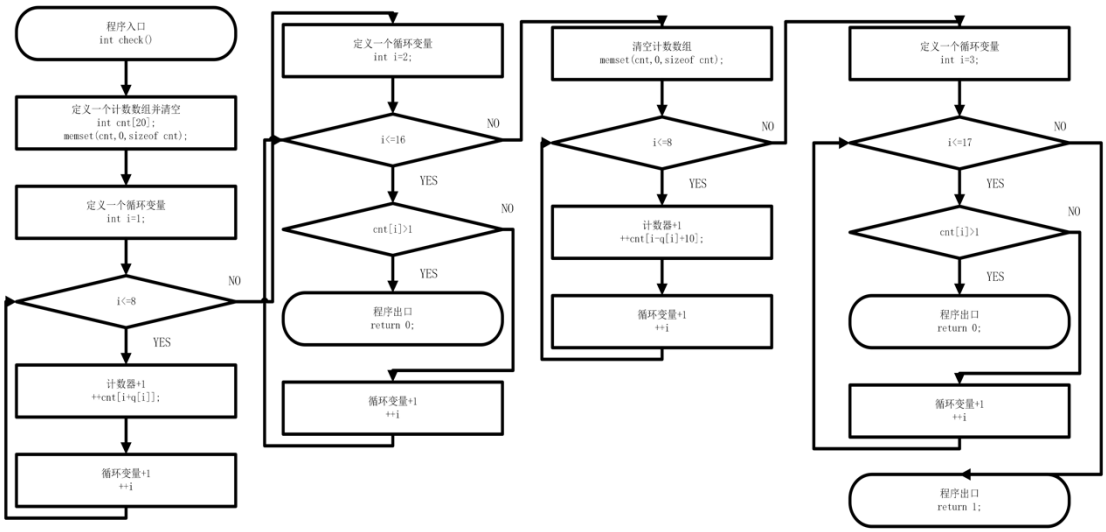
（2）概要设计

存储结构设计：

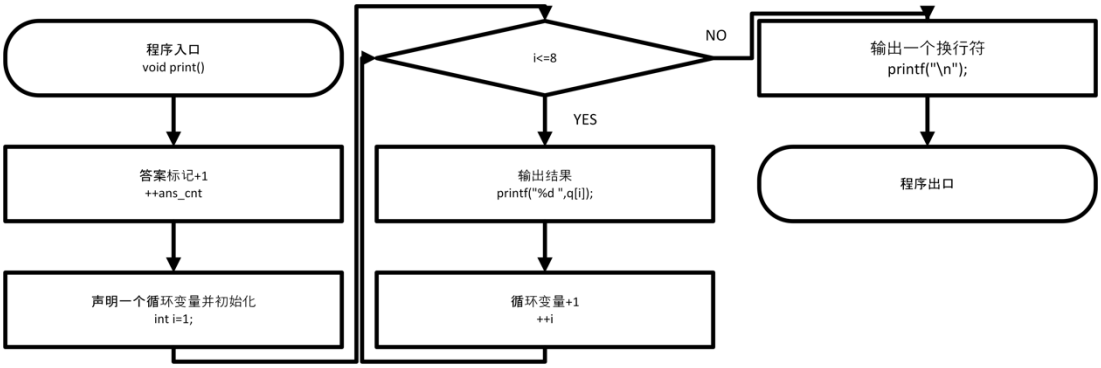
通过一个长度为 8 的一维数组生成全排列，代表八皇后的位置。通过递归的方式模拟出一个栈，完成 dfs 的过程。通过一个长度为 14 的一位数组对八皇后的位置进行判重，但是对于差运算，需要计算一个偏移值。

算法设计（流程图）：

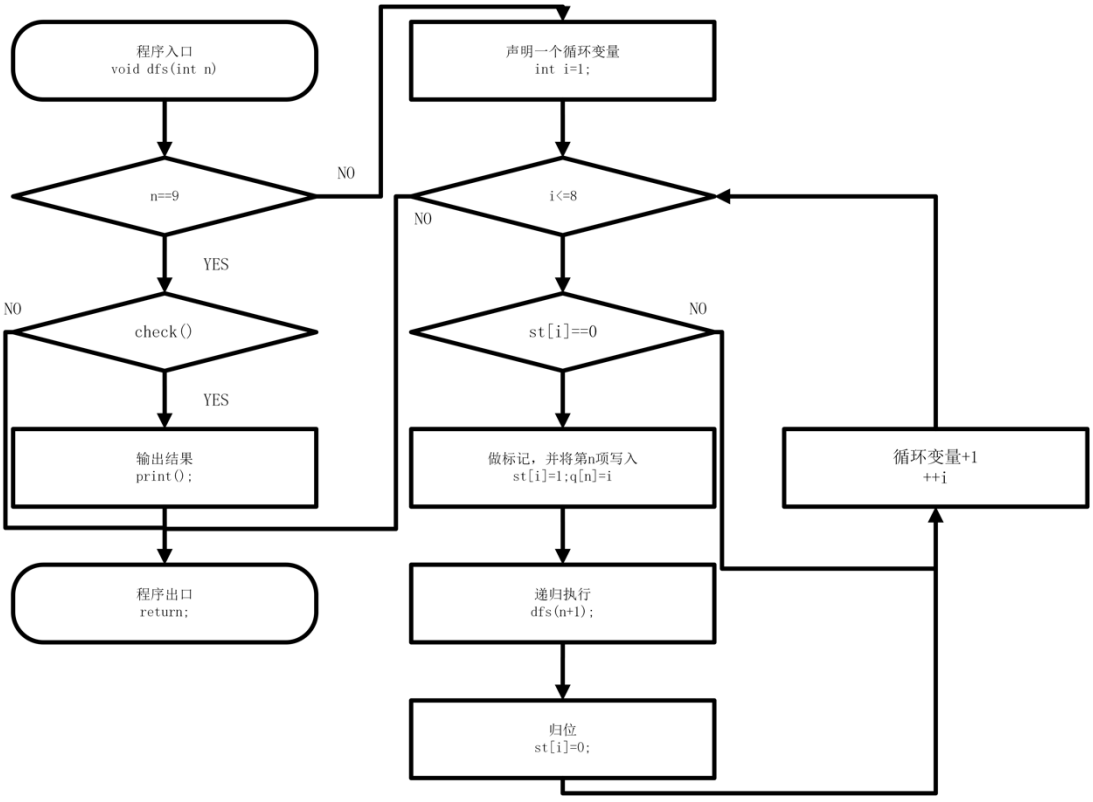
Int check()函数的实现过程：



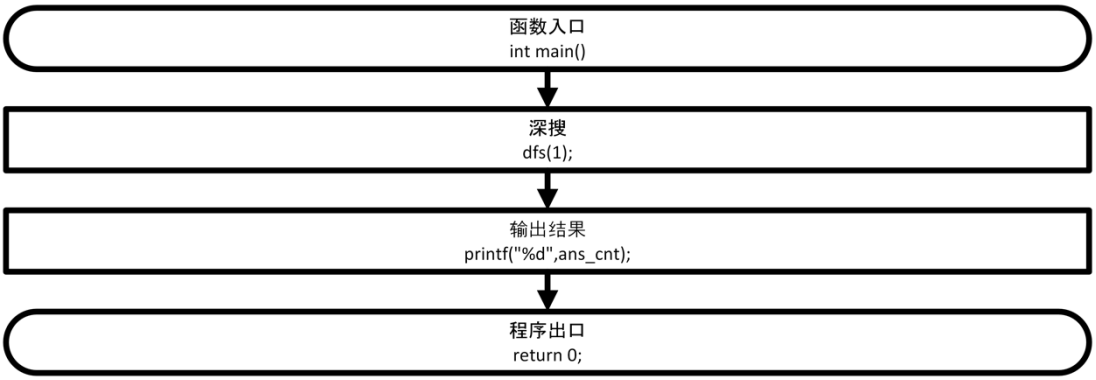
Void print()函数的实现过程：




Void dfs(int n)的函数实现过程：




Int main()函数的实现过程：



（3）详细设计

源代码（注释）

```
#include<stdio.h>
#include<string.h>
int q[9],st[9],ans_cnt;
//q 数组用于存放答案，st 代表这个数字是否被使用过，ans_cnt 统计答案的个数
int check()
{
    //先统计序号和 q 中值之和
    int cnt[20];
    memset(cnt,0,sizeof cnt);
    for(int i=1;i<=8;++i) ++cnt[i+q[i]];
    //统计每一个皇后所在位置的横坐标和纵坐标之和
    for(int i=2;i<=16;++i)
        if(cnt[i]>1) return 0;
    //如果存在两个数字的横坐标和纵坐标之和相等
    //那就说明存在两个或者更多的皇后出现在同一个副对角线上
    //先统计序号和 q 中值之差(带偏移值)
    memset(cnt,0,sizeof cnt);
    for(int i=1;i<=8;++i) ++cnt[i-q[i]+10];
    //统计每一个皇后坐在位置横坐标和纵坐标位置之差(算偏移值)
    for(int i=3;i<=17;++i)
        if(cnt[i]>1) return 0;
    //如果存在两个数字其横坐标和纵坐标点差相等
    //那就说明存在两个或更多的皇后出现在同一个主对角线上
    return 1;
}
void print()
{
    ans_cnt++;//print 答案说明答案正确
    for(int i=1;i<=8;++i) printf("%d ",q[i]);
    //将答案数组 q 里的元素全部输出
    printf("\n");
    //输出结束
}
void dfs(int n)
{
    if(n==9)
    //如果已经搜索到了第九层，说明全排列已经生成完毕
    {
        if(check()) print();
        //如果满足要求，那么输出结果
        return;//递归退层
    }
```

```
    for(int i=1;i<=8;++i)
    //生成全排列
    {
        if(st[i]==0)
        {
            st[i]=1;
            q[n]=i;
            //如果这个数字没有被使用过，那么使用
            dfs(n+1);
            st[i]=0;
            //归位
        }
    }
}
int main()
{
    dfs(1);
    //从第一位开始生成
    printf("%d",ans_cnt);
    //输出结果的总数
    return 0;
}
```

（4）调试分析

输出结果如下：

| | | |
|---|---|---|
| 1 5 8 6 3 7 2 4 | 3 6 2 7 5 1 8 4 | 4 6 8 3 1 7 5 2 |
| 1 6 8 3 7 4 2 5 | 3 6 4 1 8 5 7 2 | 4 7 1 8 5 2 6 3 |
| 1 7 4 6 8 2 5 3 | 3 6 4 2 8 5 7 1 | 4 7 3 8 2 5 1 6 |
| 1 7 5 8 2 4 6 3 | 3 6 8 1 4 7 5 2 | 4 7 5 2 6 1 3 8 |
| 2 4 6 8 3 1 7 5 | 3 6 8 1 5 7 2 4 | 4 7 5 3 1 6 8 2 |
| 2 5 7 1 3 8 6 4 | 3 6 8 2 4 1 7 5 | 4 8 1 3 6 2 7 5 |
| 2 5 7 4 1 8 6 3 | 3 7 2 8 5 1 4 6 | 4 8 1 5 7 2 6 3 |
| 2 6 1 7 4 8 3 5 | 3 7 2 8 6 4 1 5 | 4 8 5 3 1 7 2 6 |
| 2 6 8 3 1 4 7 5 | 3 8 4 7 1 6 2 5 | 5 1 4 6 8 2 7 3 |
| 2 7 3 6 8 5 1 4 | 4 1 5 8 2 7 3 6 | 5 1 8 4 2 7 3 6 |
| 2 7 5 8 1 4 6 3 | 4 1 5 8 6 3 7 2 | 5 1 8 6 3 7 2 4 |
| 2 8 6 1 3 5 7 4 | 4 2 5 8 6 1 3 7 | 5 2 4 6 8 3 1 7 |
| 3 1 7 5 8 2 4 6 | 4 2 7 3 6 8 1 5 | 5 2 4 7 3 8 6 1 |
| 3 5 2 8 1 7 4 6 | 4 2 7 3 6 8 5 1 | 5 2 6 1 7 4 8 3 |
| 3 5 2 8 6 4 7 1 | 4 2 7 5 1 8 6 3 | 5 2 8 1 4 7 3 6 |
| 3 5 7 1 4 2 8 6 | 4 2 8 5 7 1 3 6 | 5 3 1 6 8 2 4 7 |
| 3 5 8 4 1 7 2 6 | 4 2 8 6 1 3 5 7 | 5 3 1 7 2 8 6 4 |
| 3 6 2 5 8 1 7 4 | 4 6 1 5 2 8 3 7 | 5 3 8 4 7 1 6 2 |
| 3 6 2 7 1 4 8 5 | 4 6 8 2 7 1 3 5 | 5 7 1 3 8 6 4 2 |

5 7 1 4 2 8 6 3
5 7 2 4 8 1 3 6
5 7 2 6 3 1 4 8
5 7 2 6 3 1 8 4
5 7 4 1 3 8 6 2
5 8 4 1 3 6 2 7
5 8 4 1 7 2 6 3
6 1 5 2 8 3 7 4
6 2 7 1 3 5 8 4
6 2 7 1 4 8 5 3
6 3 1 7 5 8 2 4
6 3 1 8 4 2 7 5
6 3 1 8 5 2 4 7
6 3 5 7 1 4 2 8
6 3 5 8 1 4 2 7
6 3 7 2 4 8 1 5
6 3 7 2 8 5 1 4
6 3 7 4 1 8 2 5
6 4 1 5 8 2 7 3
6 4 2 8 5 7 1 3
6 4 7 1 3 5 2 8
6 4 7 1 8 2 5 3
6 8 2 4 1 7 5 3
7 1 3 8 6 4 2 5
7 2 4 1 8 5 3 6
7 2 6 3 1 4 8 5
7 3 1 6 8 5 2 4
7 3 8 2 5 1 6 4
7 4 2 5 8 1 3 6
7 4 2 8 6 1 3 5
7 5 3 1 6 8 2 4
8 2 4 1 7 5 3 6
8 2 5 3 1 7 4 6
8 3 1 6 2 5 7 4
8 4 1 3 6 2 7 5
92

运行结果的截屏：

5 2 4 7 3 8 6 1
5 2 6 1 7 4 8 3
5 2 8 1 4 7 3 6
5 3 1 6 8 2 4 7
5 3 1 7 2 8 6 4
5 3 8 4 7 1 6 2
5 7 1 3 8 6 4 2
5 7 1 4 2 8 6 3
5 7 2 4 8 1 3 6
5 7 2 6 3 1 4 8
5 7 2 6 3 1 8 4
5 7 4 1 3 8 6 2
5 8 4 1 3 6 2 7
5 8 4 1 7 2 6 3
6 1 5 2 8 3 7 4
6 2 7 1 3 5 8 4
6 2 7 1 4 8 5 3
6 3 1 7 5 8 2 4
6 3 1 8 4 2 7 5
6 3 1 8 5 2 4 7
6 3 5 7 1 4 2 8
6 3 5 8 1 4 2 7
6 3 7 2 4 8 1 5
6 3 7 2 8 5 1 4
6 3 7 4 1 8 2 5
6 4 1 5 8 2 7 3
6 4 2 8 5 7 1 3
6 4 7 1 3 5 2 8
6 4 7 1 8 2 5 3
6 8 2 4 1 7 5 3
7 1 3 8 6 4 2 5
7 2 4 1 8 5 3 6
7 2 6 3 1 4 8 5
7 3 1 6 8 5 2 4
7 3 8 2 5 1 6 4
7 4 2 5 8 1 3 6
7 4 2 8 6 1 3 5
7 5 3 1 6 8 2 4
8 2 4 1 7 5 3 6
8 2 5 3 1 7 4 6
8 3 1 6 2 5 7 4
8 4 1 3 6 2 7 5
92%

关于输出结果的解释(以 1 5 8 6 3 7 2 4)为例：
这样一个输出结果代表在第 1 列的第 1 行有一个皇后，在第 2 列的第 5 行有一个皇后。如果用 1 代表皇后，用 0 代表其他的点，那么这样一个输出结果可以表示为：
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
通过观察我们可以发现，上图中，在任何一列，任何一行，任何一条对角线上，都不存在两个皇后。

（5）小结
在本次实习的过程中，八皇后问题的存储结构设计让我印象尤其深刻。如果直接存储八皇后问题的图，则需要对行列和对角线均进行判断，但是如果先生成全排列，再再全排列中进行判断，只需要判断对角线即可，大大减少了程序的计算量，提高运算效率。

# 6. 运动会分数统计

**任务**

（1）需求分析
分析总结逻辑结构：

分析总结运算集合：

（2）概要设计
存储结构设计：

算法设计（流程图）：

（3）详细设计
源代码（注释）

（4）调试分析
……
（5）小结

# 7. 订票系统

**任务**

（1）需求分析
分析总结逻辑结构：

分析总结运算集合：

（2）概要设计
存储结构设计：

算法设计（流程图）：

（3）详细设计
源代码（注释）

（4）调试分析
……
（5）小结

# 8. 文章编辑

**任务**

（1）需求分析
分析总结逻辑结构：

分析总结运算集合：

（2）概要设计
存储结构设计：

算法设计（流程图）：

（3）详细设计
源代码（注释）

（4）调试分析
……
（5）小结

# 9. 树形结构

**任务**

（1）需求分析
分析总结逻辑结构：

分析总结运算集合：

（2）概要设计
存储结构设计：

算法设计（流程图）：

（3）详细设计
源代码（注释）

（4）调试分析
……
（5）小结

# 10.哈夫曼编码

**任务**

（1）需求分析
分析总结逻辑结构：

分析总结运算集合：

（2）概要设计
存储结构设计：

算法设计（流程图）：

（3）详细设计
源代码（注释）

（4）调试分析
……
（5）小结

# 11. 并查集

**任务**

（1）需求分析
分析总结逻辑结构：

分析总结运算集合：

（2）概要设计
存储结构设计：

算法设计（流程图）：

（3）详细设计
源代码（注释）

（4）调试分析
……
（5）小结

# 12.最小代价生成树

**任务**

（1）需求分析
分析总结逻辑结构：

分析总结运算集合：

（2）概要设计
存储结构设计：

算法设计（流程图）：

（3）详细设计
源代码（注释）

（4）调试分析
……
（5）小结

# 13.拓补排序及关键路径

**任务**

（1）需求分析
分析总结逻辑结构：

分析总结运算集合：

（2）概要设计
存储结构设计：

算法设计（流程图）：

（3）详细设计
源代码（注释）

（4）调试分析
……
（5）小结

# 14.交通咨询系统

**任务**

（1）需求分析
分析总结逻辑结构：

分析总结运算集合：

（2）概要设计
存储结构设计：

算法设计（流程图）：

（3）详细设计
源代码（注释）

（4）调试分析
……
（5）小结

# 15.查找技术

**任务**

（1）需求分析
分析总结逻辑结构：

分析总结运算集合：

（2）概要设计
存储结构设计：

算法设计（流程图）：

（3）详细设计
源代码（注释）

（4）调试分析
……
（5）小结

# 16. 排序技术

**任务**

利用相关排序算法，将用户随机输入的一组整数(20<=个数<=50)按递增的顺序排好。

要求:

1. 输入的数据形式为整数。
2. 输出的形式:数字大小逐个递增的数列。
3. 比较不同排序算法的优缺点。

（1）需求分析

分析总结逻辑结构：

排序，用数组存储，将一些杂乱无章的数字依照升序排列即可

分析总结运算集合：

交换两个数的位置

（2）概要设计

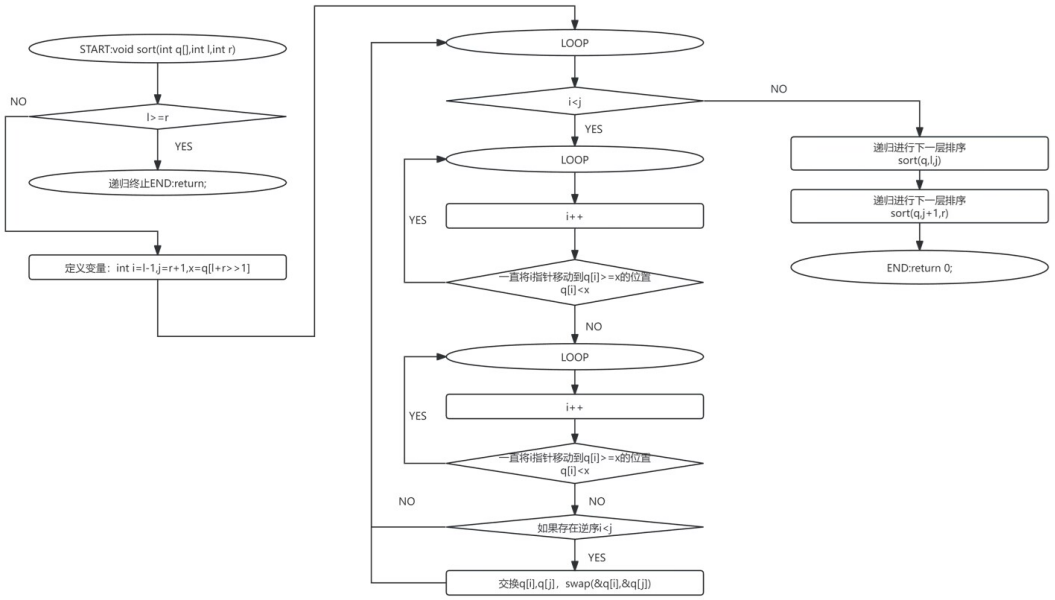
存储结构设计：

顺序存储：数据元素按顺序存放在一个连续的存储空间中，常用数组表示。适合直接访问，如快速排序、归并排序等。

辅助数组：在归并排序等算法中使用，用于暂存中间结果。

算法设计（流程图）：

以快速排序为例，快速排序利用双指针，从两边往中间靠，如果遇到了一个数字位于基准值错误的一边，那么停下来将两个值进行交换即可。若双指针相遇，则代表数组已经被分成了2边，一遍大于一个值，另一边小于一个值：



（3）详细设计

源代码（注释）

- 快速排序

```
#include<stdio.h>
// 交换两个整数的值
void swap(int *a, int *b)
{
    int tmp = *a;  // 临时变量存储 a 的值
    *a = *b;       // 将 b 的值赋给 a
    *b = tmp;      // 将临时变量的值（即原来的 a 值）赋给 b
}
// 实现快速排序的递归函数
void quick_sort(int q[], int l, int r)
{
    if(l >= r) return;
    // 如果子数组长度为 0 或 1，直接返回
    int mid = q[l + r >> 1];
    // 选择中间位置的元素作为基准值
    int i = l - 1, j = r + 1;
    // 初始化两个指针：i 指向左边界外，j 指向右边界外
    while(i < j)
    // 开始分区过程
    {
        do ++i; while(q[i] < mid);
        // 从左向右找到第一个大于等于基准值的元素
        do --j; while(q[j] > mid);
        // 从右向左找到第一个小于等于基准值的元素
        if(i < j) swap(&q[i], &q[j]);
        // 如果找到的一对元素的位置不正确，交换它们
    }
    // 递归地对分区后的左右部分进行快速排序
    quick_sort(q, l, i - 1);
    // 对左半部分排序
    quick_sort(q, j + 1, r);
    // 对右半部分排序
}
//主程序
int main()
{
    int n, num[100];
    // 声明变量：n 表示数组的大小，num 数组存储待排序的数
    scanf("%d", &n);
    // 读取数组的大小
    for(int i = 0; i < n; ++i) scanf("%d", &num[i]);
```

```
    // 读取数组中的 n 个元素
    quick_sort(num, 0, n - 1);
     // 对整个数组进行快速排序
    for(int i = 0; i < n; ++i) printf("%d ", num[i]);
    // 输出排序后的数组
    return 0;  // 程序结束
}
```

- 归并排序

```
#include<stdio.h>
// 声明全局变量
int num[100010], tmp[100010], n;
// 归并排序的递归函数
void merge_sort(int l, int r)
{
    if(l >= r) return;
    // 如果子数组长度为 0 或 1，直接返回，递归结束条件
    int mid=l+r>>1;
    // 计算中间位置，用于将数组分成两部分
    merge_sort(l, mid);
    // 递归排序左半部分
    merge_sort(mid + 1, r);
    // 递归排序右半部分
    int k = 1, i = l, j = mid + 1;
    // 初始化指针：k 用于临时数组，i 用于左半部分，j 用于右半部分
    // 合并两个有序的子数组
    while(i <= mid && j <= r)
    {
        if(num[i] < num[j]) tmp[k++] = num[i++];
        // 如果左半部分当前元素较小，放入临时数组
        else tmp[k++] = num[j++];
        // 否则放入右半部分当前元素
    }
    // 如果左半部分有剩余元素，继续放入临时数组
    while(i <= mid) tmp[k++] = num[i++];
    // 如果右半部分有剩余元素，继续放入临时数组
    while(j <= r) tmp[k++] = num[j++];
    // 将临时数组中的元素拷贝回原数组对应位置
    for(i = 1, j = l; j <= r; i++, j++) num[j] = tmp[i];
}
int main()
{
    scanf("%d", &n);
    // 读取数组大小 n
    for(int i = 1; i <= n; i++) scanf("%d", &num[i]);
```

```
    // 读取 n 个元素到 num 数组中
    merge_sort(1, n);
    // 对整个数组进行归并排序
    for(int i = 1; i <= n; i++) printf("%d ", num[i]);
    // 输出排序后的数组
    return 0;
}
```

- 桶排序

```
#include<stdio.h>
#include<string.h>
int main()
{
    int n,a[100],b;
    memset(a,0,sizeof(a));
    scanf("%d",&n);
    for(int i=1;i<=n;i++)
    {
        scanf("%d",&b);
        if(b>100||b<0)
        {
            printf("输入数据错误");
            return 0;
        }
        a[b]++;
    }
    for(int i=0;i<=99;i++)
    {
        while(a[i]>0)
        {
            printf("%d ",i);
            a[i]--;
        }
    }
    return 0;
}
```

- 冒泡排序

```
#include<stdio.h>
const int N=10;
int a[N];
int main()
{
    for(int i=0;i<10;i++) scanf("%d",&a[i]);
    for(int i=0;i<10;i++)
    {
```

```
        for(int j=1;j<10;j++)
        {
            if(a[j-1]>a[j])
            {
                int tmp=a[j-1];
                a[j-1]=a[j];
                a[j]=tmp;
            }
        }
    }
    for(int i=0;i<10;i++) printf("%d ",a[i]);
    return 0;
}
```

- 拓扑排序

```
#include<stdio.h>
#include<string.h>
const int N = 100010;
int e[N], ne[N], idx;
int h[N];
int q[N], hh = 0, tt = -1;
int n, m;
int d[N];
void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx++;
}
void topsort()
{
    for(int i = 1; i <= n; i++) if(d[i] == 0) q[++tt] = i;
    while(tt >= hh)
    {
        int a = q[hh++];
        for(int i = h[a]; i != -1; i = ne[i])
        {
            int b = e[i];
            d[b]--;
            if(d[b] == 0) q[++tt] = b;
        }
    }
    if(tt == n - 1)
    {
        for(int i = 0; i < n; i++) printf("%d ",q[i]);
    }
    else printf("-1");
```

```
}
int main(){
    scanf("%d%d",&n,&m);
    memset(h, -1, sizeof h);
    while (m -- )
    {
        int a, b;
        scanf("%d%d",&a,&b);
        d[b]++;
        add(a, b);
    }
    topsort();
    return 0;
}
```

- 选择排序

```
#include<stdio.h>
const int N=1e5+10;
int arr[N],n;
int main()
{
    scanf("%d",&n);
    for(int i=0;i<n;i++) scanf("%d",&arr[i]);
    for(int i=0;i<n;i++)
    {
        int tmp=i;
        for(int j=i+1;j<n;j++)
        {
            if(arr[j]<arr[tmp]) tmp=j;
        }
        int mid=arr[tmp];
        arr[tmp]=arr[i];
        arr[i]=mid;
    }
    for(int i=0;i<n;i++) printf("%d ",arr[i]);
    return 0;
}
```

- 插入排序

```
#include<stdio.h>
const int N=1e5+10;
int arr[N],n;
int main()
{
    scanf("%d",&n);
    for(int i=0;i<n;i++) scanf("%d",&arr[i]);
```

```
    for(int i=0;i<n;i++)
    {
        for(int j=i;j>=1;j--)
        {
            if(arr[j]<arr[j-1])
            {
                int tmp=arr[j];
                arr[j]=arr[j-1];
                arr[j-1]=tmp;
            }
        }
    }
    for(int i=0;i<n;i++) printf("%d ",arr[i]);
    return 0;
}
```

- 希尔排序

```
#include <stdio.h>
// 希尔排序函数
void shell_sort(int arr[], int n)
{
    // 选择初始步长，通常为数组长度的一半
    for (int gap = n / 2; gap > 0; gap /= 2)
    {
        // 对每个步长 gap 进行分组，并对每组进行插入排序
        for (int i = gap; i < n; i++)
        {
            int temp = arr[i];
            int j;
            // 插入排序部分，将 temp 插入到它在分组内的合适位置
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) arr[j] = arr[j - gap];
            arr[j] = temp;
        }
    }
}
// 主函数
int main() {
    int n;
    printf("请输入数组的元素数量: ");
    scanf("%d", &n);
    int arr[n];
    printf("请输入数组的元素:\n");
    for (int i = 0; i < n; i++) scanf("%d", &arr[i]);
    shell_sort(arr, n);
    printf("排序后的数组:\n");
```

```
    for (int i = 0; i < n; i++) printf("%d ", arr[i]);
    return 0;
}
```

（4）调试分析

所有排序算法的测试数据均相同：

测试数据 1:

20

23 23 28 29 19 29 39 20 10 9 18 29 57 97 45 99 26 38 28 39

测试数据 2:

50

64 13 78 52 16 89 28 4 85 66 12 19 73 45 9 39 82 96 6 35 57 7 24 61 92 77 41 33 22 69 3 83 94 25 48 59 1 30 53 98 74 2 50 88 34 5 87 17 31 46

测试数据 3:

40

77 15 42 89 61 37 81 23 54 68 10 90 28 34 76 2 55 92 43 60 4 99 7 21 85 17 72 39 57 63 30 18 11 74 3 86 25 50 96 8

对于桶排序，我给出了另一组特殊数据：

40

-24 58 -3 83 -71 48 -99 -53 100 -26 -16 -78 14 66 -34 -42 9 5 71 39 7 -11 -94 30 -17 24 4 93 -64 84 -81 50 -45 15 63 -18 80 -90 -7 12

测试数据 1:

```
20
23 23 28 29 19 29 39 20 10 9 18 29 57 97 45 99 26 38 28 39
9 10 18 19 20 23 23 26 28 28 29 29 29 38 39 39 45 57 97 99
```

测试数据 2:

```
50
64 13 78 52 16 89 28 4 85 66 12 19 73 45 9 39 82 96 6 35 57 7 24 61 92 77 41 33 22 69 3 83 94 25 48 59 1 30 53 98 74 2 50 88 34
 5 87 17 31 46
1 2 3 4 5 6 7 9 12 13 16 17 19 22 24 25 28 30 31 33 34 35 39 41 45 46 48 50 52 53 57 59 61 64 66 69 73 74 77 78 82 83 85 87 88
89 92 94 96 98
```

测试数据 3:

```
40
77 15 42 89 61 37 81 23 54 68 10 90 28 34 76 2 55 92 43 60 4 99 7 21 85 17 72 39 57 63 30 18 11 74 3 86 25 50 96 8
2 3 4 7 8 10 11 15 17 18 21 23 25 28 30 34 37 39 42 43 50 54 55 57 60 61 63 68 72 74 76 77 81 85 86 89 90 92 96 99
```

对于桶排序，测试数据 4:

```
40
-24 58 -3 83 -71 48 -99 -53 100 -26 -16 -78 14 66 -34 -42 9 5 71 39 7 -11 -94 30 -17 24 4 93 -64 84 -81 50 -45 15 63 -18 80 -9
 -7 12
输入数据错误
```

（5）小结

不同的排序方法在时间复杂度、空间复杂度上都各有优劣。比较类排序往往拥有更高的时间复杂度，其他的排序方法往往有较高的空间复杂度，没有一种排序方法是两全其美的。

冒泡排序：是一种简单的交换排序，通过重复地遍历数组，将相邻的未排序部分逐步推到数组的末尾。它的时间复杂度是 O(n^2)，且由于每次只涉及相邻元素的交换，空间复杂度为 O(1)。它是稳定的，但性能较差，不适合大型数字。

选择排序：主要思路是每次从未排序部分选择最小（或最大）的元素，放到已排序部分的末尾。它的时间复杂度为 O(n^2)，与冒泡排序类似，但它只进行 n 次交换，因此在某些情况下可能比冒泡排序更快。不过它同样不适合大规模数据，且不稳定。

插入排序：是通过将未排序的元素插入到已排序的部分中来实现的。它的时间复杂度为O(n^2)，但对几乎已排序的数据表现良好，达到O(n)的效率。空间复杂度为O(1)，且它是稳定的，常用于小规模或部分有序的数组。
希尔排序：是插入排序的改进版本，它通过选择不同的步长（gap）对数组进行分组，然后对每组进行插入排序。随着步长的减少，数组逐渐变得有序，最后一步是普通的插入排序。希尔排序的时间复杂度依赖于步长的选择，通常介于O(n^1.3)到O(n^2)之间，且不稳定。适合中等规模的数据集。
归并排序：是一种基于分治法的排序算法，它将数组递归地分成两半，分别进行排序，然后合并。归并排序的时间复杂度是O(n log n)，空间复杂度为O(n)，它是稳定的。由于其稳定性和时间复杂度，它适合处理大规模数据，但额外的空间开销较大。
快速排序：也是一种基于分治法的排序算法，通过选择一个基准元素将数组分成两部分，一部分比基准小，另一部分比基准大，然后递归地排序。它的平均时间复杂度为O(n log n)，但在最坏情况下退化为O(n^2)。快速排序通常是不稳定的，但由于其较小的常数因子和O(log n)的空间复杂度，它是实际中最常用的排序算法之一，尤其适合大数据集。
以下是通过一个表格直观反映各种排序方法优劣：

| 序号 | 排序名称 | 时间复杂度 | 空间复杂度 | 稳定性 | 缺点 |
|---|---|---|---|---|---|
| 1 | 快速排序 | O(nlogn) | 最优O(logn)，最坏O(n) | 不稳定 | 空间复杂度高 |
| 2 | 归并排序 | O(nlogn) | O(n) | 稳定 | 空间复杂度高 |
| 3 | 桶排序 | O(n) | 由值域决定 | 稳定 | 空间复杂度高 |
| 4 | 冒泡排序 | O(n^2) | O(1) | 稳定 | 时间复杂度高 |
| 5 | 拓扑排序 | O(n+e) | O(n+e) | 不稳定 | 空间复杂度高 |
| 6 | 选择排序 | O(n^2) | O(1) | 不稳定 | 时间复杂度高 |
| 7 | 插入排序 | O(n^2) | O(1) | 稳定 | 时间复杂度高 |
| 8 | 希尔排序 | O(n^1.3) | O(1) | 不稳定 | 特殊情况下时间复杂度高 |