手写代码必备手册

戴方勤 (soulmachine@gmail.com)

https://github.com/soulmachine/acm-cheat-sheet

最后更新 2013-7-19

版权声明

内容简介

本书包含了一些经典题目的范例代码，经过精心编写，编码规范良好，适合在纸上默写。

怎么样才算是经典的算法题？一般经典的题目都有约定俗成的名称，例如"八皇后问题"，"0-1 背包问题"等，这些名字已经固定下来了，类似于一个"成语"，一般说出名字，大家就都知道题目意思了，不用再解释题目内容。同时，本书的每一个题目，都至少在两本纸质书中出现过。

这本手册的定位，比 ACM 模板库的代码少，题型比 ACM 简单，对代码有一些讲解。ACM 代码库功能全，很多难度很高，且整本手册都是代码，没有讲解。

全书的代码，使用纯 C + STL 的风格。本书中的代码规范，跟在公司中的工程规范略有不同，为了使代码短（方便迅速实现）：

- 所有代码都是单一文件。这是因为一般 OJ 网站，提交代码的时候只有一个文本框，如果还是按照标准做法，比如分为头文件.h 和源代码.cpp，无法在网站上提交；
- 喜欢在全局定义一个最大整数，例如 MAX。一般的 OJ 题目，都会有数据规模的限制，所以定义一个常量 MAX 表示这个规模，可以不用动态分配内存，让代码实现更简单；
- 经常使用全局变量。比如用几个全局变量，定义某个递归函数需要的数据，减少递归函数的参数个数，就减少了递归时栈内存的消耗，可以说这几个全局变量是这个递归函数的"环境"。
- 不提倡防御式编程。不需要检查 malloc()/new 返回的指针是否为 NULL；不需要检查内部函数入口参数的有效性；使用纯 C 基于对象编程时，调用对象的成员方法，不需要检查对象自身是否为 NULL。

本手册假定读者已经学过《数据结构》[①]，《算法》[②] 这两门课，熟练掌握 C++ 或 Java。

本手册是开源的，项目地址：https://github.com/soulmachine/acm-cheatsheet

更新记录

2013-05-19 完成了一些重要章节

2013-04-14 创建 Repo，开始编写

[①]《数据结构》，严蔚敏等著，清华大学出版社，http://book.douban.com/subject/2024655/

[②]《Algorithms》，Robert Sedgewick, Addison-Wesley Professional, http://book.douban.com/subject/4854123/

目录

第 1 章 编程技巧

把较大的数组放在 main 函数外，作为全局变量，这样可以防止栈溢出，因为栈的大小是有限制的。

如果能够预估栈，队列的上限，则不要用 `std::stack`，`std::queue`，使用数组来模拟，这样速度最快。

输入数据一般放在全局变量，且在运行过程中不要修改这些变量。

在判断两个浮点数 a 和 b 是否相等时，不要用 `a==b`，应该判断二者之差的绝对值 `fabs(a-b)` 是否小于某个阈值，例如 `1e-9`。

第 2 章
线性表

线性表 (Linear List) 包含以下几种：

- 顺序存储：数组
- 链式存储：单链表，双向链表，循环单链表，循环双向链表
- 二者结合：静态链表

第 3 章 栈和队列

栈 (stack) 只能在表的一端插入和删除，先进后出（LIFO, Last In, First Out）。

队列 (queue) 只能在表的一端（队尾 rear）插入，另一端（队头 front）删除，先进先出（FIFO, First In, First Out）。

3.1 栈

3.1.1 栈的 C 语言实现

C++ 可以直接使用 `std::stack`。

stack.c

```c
/**
 * @file stack.c
 * @brief 栈，顺序存储.
 * @author soulmachine@gmail.com
 * @date 2010-7-31
 * @version 1.0
 */
#include <stdlib.h>  /* for malloc() */
#include <string.h>  /* for memcpy() */

typedef int stack_elem_t; // 元素的类型

/**
 * @struct
 * @brief 栈的结构体
 */
typedef struct stack_t {
    int     size;  /** 实际元素个数 */
    int     capacity; /** 容量，以元素为单位 */
    stack_elem_t  *elems;   /** 栈的数组 */
}stack_t;

/**
 * @brief 初始化栈.
 * @param[inout] s 栈对象的指针
 * @param[in] capacity 初始容量
 * @return 无
```

```
 */
void stack_init(stack_t *s, const int capacity) {
    s->size = 0;
    s->capacity = capacity;
    s->elems = (stack_elem_t*)malloc(capacity * sizeof(stack_elem_t));
}

/**
 * @brief 释放栈.
 * @param[inout] s 栈对象的指针
 * @return 无
 */
void stack_uninit(stack_t *s) {
    s->size = 0;
    s->capacity = 0;
    free(s->elems);
    s->elems = NULL;
}

/**
 * @brief 判断栈是否为空.
 * @param[in] s 栈对象的指针
 * @return 是空，返回 1，否则返回 0
 */
int stack_empty(const stack_t *s) {
    return s->size == 0;
}

/**
 * @brief 获取元素个数.
 * @param[in] s 栈对象的指针
 * @return 元素个数
 */
int stack_size(const stack_t *s) {
    return s->size;
}

/**
 * @brief 进栈.
 * @param[in] s 栈对象的指针
 * @param[in] x 要进栈的元素
 * @return 无
 */
void stack_push(stack_t *s, const stack_elem_t x)
{
    if(s->size == s->capacity) { /* 已满，重新分配内存 */
        stack_elem_t* tmp = (stack_elem_t*)realloc(s->elems,
            s->capacity * 2 * sizeof(stack_elem_t));
        s->capacity *= 2;
        s->elems = tmp;
    }
    s->elems[s->size++] = x;
}
```

```
/**
 * @brief 进栈.
 * @param[in] s 栈对象的指针
 * @return 无
 */
void stack_pop(stack_t *s) {
    s->size--;
}

/**
 * @brief 获取栈顶元素.
 * @param[in] s 栈对象的指针
 * @return 栈顶元素
 */
stack_elem_t stack_top(const stack_t *s) {
    return s->elems[s->size - 1];
}
```

stack.c

3.1.2 Hanoi 塔问题

n 阶 Hanoi 塔问题假设有三个分别命名为 X、Y 和 Z 的塔座，在塔座 X 上插有 n 个直径大小各不相同、从小到大编号为 1，2，…，n 的圆盘，如图 3-1所示。

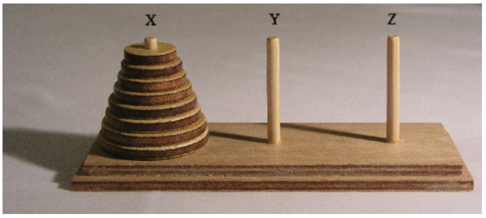


图 3-1 Hanoi 塔问题

现要求将 X 塔上的 n 个圆盘移动到 Z 上并仍按同样的顺序叠放，圆盘移动时必须遵循下列规则：

- 每次只能移动一个圆盘；
- 圆盘可以插在 X、Y 和 Z 中的任一塔座上；
- 任何时刻都不能将一个较大的圆盘压在较小的圆盘之上。

递归代码如下：

hanoi.c

```
/*
 * @brief 将塔座 x 上按直径有小到大且自上而下编号
 * 为 1 至 n 的 n 个圆盘按规则搬到塔座 z 上，y 可用做辅助塔座.
 * @param[in] n 圆盘个数
```

```
 * @param[in] x 源塔座
 * @param[in] y 辅助塔座
 * @param[in] z 目标塔座
 * @return 无
 * @note 无
 * @remarks 无
 */
void hanoi(int n, char x, char y, char z)
{
    if(n ==  1) {
        /* 移动操作 move(x,n,z) 可定义为 (c 是初始值为的全局
           变量, 对搬动计数):
           printf("%i. Move disk %i from %c to %c\n",
                                         ++c, n, x, z);
        */
        move(1, x, z); /* 将编号为 1 的圆盘从 x 移动到 z */
        return;
    } else {
        /* 将 x 上编号 1 至 n-1 的圆盘移到 y, z 作辅助塔 */
        hanoi(n-1, x, z, y);
        move(n,x,z);  /* 将编号为 n 的圆盘从 x 移到 z */
        /* 将 y 上编号至 n-1 的圆盘移到 z, x 作辅助塔 */
        hanoi(n-1, y, x, z);
    }
}
```

hanoi.c

3.1.3 数制转换

convert_base.cpp

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

 /**
  * @brief 数制转换, 将一个整数转化为 d 进制, d<=16.
  * @param[in] n 整数 n
  * @param[in] d d 进制
  * @return 无
  */
void convert_base(int n, const int d) {
        std::stack<int> s;
        int e;

        while(n != 0) {
                e = n % d;
                s.push(e);
                n /= d;
        }
        while(!s.empty()) {
                e = s.top();
                s.pop();
                printf("%x", e);
```

```
        }
        return;
}

#define MAX 64 // 栈的最大长度
int stack[MAX];
int top = -1;
 /**
  * @brief 数制转换，将一个整数转化为 d 进制，d<=16，更优化的版本.
  *
  * 如果可以预估栈的最大空间，则用数组来模拟栈，这时常用的一个技巧。
  * 这里，栈的最大长度是多少？假设 CPU 是 64 位，最大的整数则是 2^64，由于
  * 数制最小为 2，在这个进制下，数的位数最长，这就是栈的最大长度，最长为 64。
  *
  * @param[in] n 整数 n
  * @param[in] d d 进制
  * @return 无
  */
void convert_base2(int n, const int d) {
        int e;

        while(n != 0) {
                e = n % d;
                stack[++top] = e; // push
                n /= d;
        }
        while(top >= 0) {
                e = stack[top--]; // pop
                printf("%x", e);
        }
        return;
}
```

convert_base.cpp

3.2 队列

3.2.1 队列的 C 语言实现

C++ 可以直接使用 std::queue。

queue.c

```
/** @file queue.c
  * @brief 队列，顺序存储，循环队列.
  * @author soulmachine@gmail.com
  * @date 2010-7-30
  * @version 1.0
  */
#include <stdlib.h> /* for malloc(), free() */
#include <string.h> /* for memcpy() */

#ifndef __cplusplus
```

```
typedef char bool;
#define false 0
#define true 1
#endif

typedef int queue_elem_t; // 元素的类型

/*
  *@struct
  *@brief 队列的结构体定义.
  *@note 无
  */
typedef struct queue_t {
    int front;   /* 队头 */
    int rear;    /* 队尾 */
    int capacity; /* 容量大小, 以元素为单位 */
    queue_elem_t *elems; /* 存放数据的内存块 */
}queue_t;

/**
  * @brief 初始化队列.
  * @param[out] q 队列结构体的指针
  * @param[in] capacity 初始容量
  * @return 无
  */
void queue_init(queue_t *q, const int capacity) {
    q->front = 0;
    q->rear = 0;
    q->capacity = capacity;
    q->elems = (queue_elem_t*)malloc(capacity * sizeof(queue_elem_t));
}

/**
  * @brief 释放队列.
  * @param[inout] q 队列对象的指针
  * @return 无
  */
void queue_uninit(queue_t *q) {
    q->front = 0;
    q->rear = 0;
    q->capacity = 0;
    free(q->elems);
    q->elems = NULL;
}

/**
  * @brief 判断队列是否为空.
  * @param[in] q 队列结构体的指针
  * @return 是空, 返回 TRUE, 否则返回 FALSE
  */
bool queue_empty(const queue_t *q) {
    return q->front == q->rear;
}
```

```
/**
 * @brief 获取元素个数.
 * @param[in] s 栈对象的指针
 * @return 元素个数
 */
int queue_size(const queue_t *q) {
    return (q->rear - q->front + q->capacity) % q->capacity;
}

/**
  * @brief 在队尾添加元素.
  * @param[in] q 指向队列结构体的指针
  * @param[in] x 要添加的元素
  * @return 无
  */
void queue_push(queue_t *q, const queue_elem_t x) {
    if( (q->rear+1) % q->capacity == q->front) { // 已满, 重新分配内存
        queue_elem_t* tmp = (queue_elem_t*)malloc(
            q->capacity * 2 * sizeof(queue_elem_t));
        if(q->front < q->rear) {
            memcpy(tmp, q->elems + q->front,
                (q->rear - q->front) * sizeof(queue_elem_t));
            q->rear -= q->front;
            q->front = 0;
        } else if(q->front > q->rear) {
            /* 拷贝 q->front 到 q->capacity 之间的数据 */
            memcpy(tmp, q->elems + q->front,
                (q->capacity - q->front) * sizeof(queue_elem_t));
            /* 拷贝 q->data[0] 到 q->data[rear] 之间的数据 */
            memcpy(tmp +
                (q->capacity - q->front),
                q->elems, q->rear * sizeof(queue_elem_t));
            q->rear += q->capacity - q->front;
            q->front = 0;
        }
        free(q->elems);
        q->elems = tmp;
        q->capacity *= 2;
    }
    q->elems[q->rear] = x;
    q->rear = (q->rear + 1) % q->capacity;
}

/**
  * @brief 在队头删除元素.
  * @param[in] q 队列结构体的指针
  * @param[out] x 存放退出队列的元素
  * @return 无
  */
void queue_pop(queue_t *q) {
    q->front = (q->front + 1) % q->capacity;
}
```

```c
/**
 * @brief 获取队首元素.
 * @param[in] q 队列对象的指针
 * @return 队首元素
 */
queue_elem_t queue_front(const queue_t *q) {
    return q->elems[q->front];
}

/**
 * @brief 获取队首元素.
 * @param[in] q 队列对象的指针
 * @return 队首元素
 */
queue_elem_t queue_back(const queue_t *q) {
    return q->elems[q->rear - 1];
}
```

queue.c

3.2.2 打印杨辉三角

yanghui_triangle.cpp

```cpp
#include <queue>
/**
 * @brief 打印杨辉三角系数.
 *
 * 分行打印二项式 (a+b)^n 展开式的系数。在输出上一行
 * 系数的同时，将其下一行的系数预先计算好，放入队列中。
 * 在各行系数之间插入一个 0。
 *
 * @param[in] n (a+b)^n
 * @return 无
 */
void yanghui_triangle(const int n) {
    int i = 1;
    std::queue<int> q;
    /* 预先放入第一行的 1 */
    q.push(i);

    for(i = 0; i <= n; i++) {         /* 逐行处理 */
        int j;
        int s = 0;
        q.push(s);      /* 在各行间插入一个 0*/

        /* 处理第 i 行的 i+2 个系数（包括一个 0）*/
        for(j = 0; j < i+2; j++) {
            int t;
            int tmp;
            t = q.front();  /* 读取一个系数，放入 t*/
            q.pop();
            tmp = s + t;      /* 计算下一行系数，并进队列 */
```

```
            q.push(tmp);
            s = t;            /* 打印一个系数，第 i+2 个是 0*/
            if(j != i+1) {
                printf("%d ",s);
            }
        }
        printf("\n");
    }
}
```

yanghui_triangle.cpp

第 4 章 树

4.1 二叉树的遍历

在中序遍历中，一个节点的前驱，是其左子树的最右下角结点，后继，是其右子树的最左下角结点。

在后序遍历中，

- 若结点是根结点，则其后继为空；
- 若结点是双亲的右子树，或是左子树但双亲无右子树，则其后继为双亲结点；
- 若结点是双亲的左子树且双亲有右子树，则其后继为右子树按后序遍历的第一个结点

binary_tree.cpp

```
#include <stack>
#include <queue>

 /*
  *@struct
  *@brief 二叉树结点
  */
typedef struct binary_tree_node_t {
    binary_tree_node_t *lchild;   /* 左孩子 */
    binary_tree_node_t *rchild;   /* 右孩子 */
    void* data; /* 结点的数据 */
}binary_tree_node_t;

/**
  * @brief 先序遍历，递归.
  * @param[in] root 根结点
  * @param[in] visit 访问数据元素的函数指针
  * @return 无
  */
void pre_order_r(const binary_tree_node_t *root,
                 int (*visit)(void*)) {
    if(root != NULL) {
        (void)visit(root->data);
        pre_order_r(root->lchild, visit);
        pre_order_r(root->rchild, visit);
```

```
    }
}

/**
  * @brief 中序遍历，递归.
  */
void in_order_r(const binary_tree_node_t *root,
                int (*visit)(void*)) {
    if(root != NULL) {
        pre_order_r(root->lchild, visit);
        (void)visit(root->data);
        pre_order_r(root->rchild, visit);
    }
}

/**
  * @brief 后序遍历，递归.
  */
void post_order_r(const binary_tree_node_t *root,
                  int (*visit)(void*)) {
    if(root != NULL) {
        pre_order_r(root->lchild, visit);
        pre_order_r(root->rchild, visit);
        (void)visit(root->data);
    }
}

/**
 * @brief 先序遍历，非递归.
 */
void pre_order(const binary_tree_node_t *root,
               int (*visit)(void*)) {
    const binary_tree_node_t *p;
    std::stack<const binary_tree_node_t *> s;

    p = root;

    if(p != NULL) {
        s.push(p);
    }

    while(!s.empty()) {
        p = s.top();
        s.pop();
        visit(p->data);
        if(p->rchild != NULL) {
            s.push(p->rchild);
        }
        if(p->lchild != NULL) {
            s.push(p->lchild);
        }
    }
}
```

```
/**
 * @brief 中序遍历，非递归.
 */
void in_order(const binary_tree_node_t *root,
              int (*visit)(void*)) {
    const binary_tree_node_t *p;
    std::stack<const binary_tree_node_t *> s;

    p = root;

    while(!s.empty() || p!=NULL) {
        if(p != NULL) {
            s.push(p);
            p = p->lchild;
        } else {
            p = s.top();
            s.pop();
            visit(p->data);
            p = p->rchild;
        }
    }
}

/**
 * @brief 后序遍历，非递归.
 */
void post_order(const binary_tree_node_t *root,
                int (*visit)(void*)) {
    /* p，正在访问的结点，q，刚刚访问过的结点 */
    const binary_tree_node_t *p, *q;
    std::stack<const binary_tree_node_t *> s;

    p = root;

    do {
        while(p != NULL) { /* 往左下走 */
            s.push(p);
            p = p->lchild;
        }
        q = NULL;
        while(!s.empty()) {
            p = s.top();
            s.pop();
            /* 右孩子不存在或已被访问，访问之 */
            if(p->rchild == q) {
                visit(p->data);
                q = p; /* 保存刚访问过的结点 */
            } else {
                /* 当前结点不能访问，需第二次进栈 */
                s.push(p);
                /* 先处理右子树 */
                p = p->rchild;
```

```
                break;
            }
        }
    }while(!s.empty());
}

/**
 * @brief 层次遍历，也即 BFS.
 *
 * 跟先序遍历一模一样，唯一的不同是栈换成了队列
 */
void level_order(const binary_tree_node_t *root,
                 int (*visit)(void*)) {
    const binary_tree_node_t *p;
    std::queue<const binary_tree_node_t *> q;

    p = root;

    if(p != NULL) {
        q.push(p);
    }

    while(!q.empty()) {
        p = q.front();
        q.pop();
        visit(p->data);
        if(p->lchild != NULL) { /* 先左后右或先右后左无所谓 */
            q.push(p->lchild);
        }
        if(p->rchild != NULL) {
            q.push(p->rchild);
        }
    }
}
```

binary_tree.cpp

4.2 线索二叉树

二叉树中存在很多空指针，可以利用这些空指针，指向其前驱或者后继。这种利用起来的空指针称为线索，这种改进后的二叉树称为线索二叉树 (threaded binary tree)。

一棵 n 个结点的二叉树含有 n+1 个空指针。这是因为，假设叶子节点数为 n_0，度为 1 的节点数为 n_1，度为 2 的节点数为 n_2，每个叶子节点有 2 个空指针，每个度为 1 的节点有 1 个空指针，则空指针的总数为 $2n_0 + n_1$，又有 $n_0 = n_2 + 1$（留给读者证明），因此空指针总数为 $2n_0 + n_1 = n_0 + n_2 + 1 + n_1 = n_0 + n_1 + n_2 + 1 = n + 1$。

在二叉树线索化过程中，通常规定，若无左子树，令 lchild 指向前驱，若无右子树，令 rchild 指向后继。还需要增加两个标志域表示当前指针是不是线索，例如 ltag=1，表示

lchild 指向的是前驱，ltag=0，表示 lchild 指向的是左孩子，rtag 类似。

二叉树的线索化，实质上就是遍历一棵树，只是在遍历的过程中，检查当前节点的左右指针是否为空，若为空，将它们改为指向前驱或后继的线索。

以中序线索二叉树为例，指针 pre 表示前驱，succ 表示后继，如图 4-1所示。

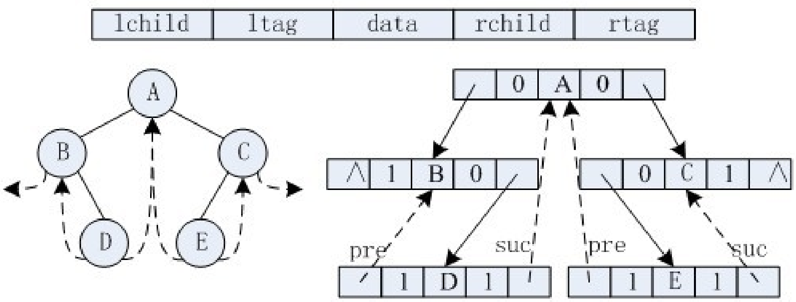


图 4-1 中序线索二叉树

在中序线索二叉树中，一个节点的前驱，是其左子树的最右下角结点，后继，是其右子树的最左下角结点。

中序线索二叉树的 C 语言实现如下。

theaded_binary_tree.c

```
/** @file threaded_binary_tree.c
  * @brief 线索二叉树.
  * @author soulmachine@gmail.com
  * @date 2013-06-16
  */
#include <stddef.h>    /* for NULL */
#include <stdio.h>

/* 结点数据的类型. */
typedef int elem_t;

 /**
  *@struct
  *@brief 线索二叉树结点.
  */
typedef struct tbt_node_t {
    int ltag; /** 1 表示是线索, 0 表示是孩子 */
    int rtag; /** 1 表示是线索, 0 表示是孩子 */
    struct tbt_node_t *lchild; /** 左孩子 */
    struct tbt_node_t *rchild; /** 右孩子 */
    elem_t data; /** 结点所存放的数据 */
}tbt_node_t;

/* 内部函数 */
static void in_thread(tbt_node_t *p, tbt_node_t **pre);
static tbt_node_t *first(tbt_node_t *p);
```

```
static tbt_node_t *next(const tbt_node_t *p);

 /**
  * @brief 建立中序线索二叉树.
  * @param[in] root 树根
  * @return 无
  */
void create_in_thread(tbt_node_t *root) {
    /* 前驱结点指针 */
        tbt_node_t *pre=NULL;
    if(root != NULL) { /* 非空二叉树, 线索化 */
        /* 中序遍历线索化二叉树 */
        in_thread(root, &pre);
        /* 处理中序最后一个结点 */
        pre->rchild = NULL;
        pre->rtag = 1;
    }
}


/**
  * @brief 在中序线索二叉树上执行中序遍历.
  * @param[in] root 树根
  * @param[in] visit 访问结点的数据的函数
  * @return 无
  */
void in_order(tbt_node_t *root, int(*visit)(elem_t*)) {
        tbt_node_t *p;
    for(p = first(root); p != NULL; p = next(p)) {
        (void)visit(&(p->data));
    }
}


 /*
  * @brief 中序线索化二叉树的主过程.
  * @param[in] p 当前要处理的结点
  * @param[inout] pre 当前结点的前驱结点
  * @return 无
  */
static void in_thread(tbt_node_t *p, tbt_node_t **pre) {
    if(p != NULL) {
        in_thread(p->lchild, pre); /* 线索化左子树 */
        if(p->lchild == NULL) {  /* 左子树为空, 建立前驱 */
            p->lchild = *pre;
            p->ltag = 1;
        }
        /* 建立前驱结点的后继线索 */
        if((*pre) != NULL &&
            (*pre)->rchild == NULL) {
            (*pre)->rchild = p;
            (*pre)->rtag = 1;
        }
```

```
        *pre = p; /* 更新前驱 */
        in_thread(p->rchild, pre); /* 线索化右子树 */
    }
}

 /*
  * @brief 寻找线索二叉树的中序下的第一个结点.
  * @param[in] p 线索二叉树中的任意一个结点
  * @return 此线索二叉树的第一个结点
  */
static tbt_node_t *first(tbt_node_t *p) {
    if(p == NULL)  return NULL;

    while(p->ltag == 0) {
        p = p->lchild;  /* 最左下结点，不一定是叶结点 */
    }
    return p;
}

 /*
  * @brief 求中序线索二叉树中某结点的后继.
  * @param[in] p 某结点
  * @return p 的后继
  */
static tbt_node_t *next(const tbt_node_t *p) {
    if(p->rtag == 0) {
        return first(p->rchild);
    } else {
        return p->rchild;
    }
}
```
theaded_binary_tree.c

中序线索二叉树最简单，在中序线索的基础上稍加修改就可以实现先序，后续就要再费点心思了。

4.3 Morris Traversal

通过前面第 §4.1节，我们知道，实现二叉树的前序（preorder）、中序（inorder）、后序（postorder）遍历有两个常用的方法，一是递归 (recursive)，二是栈 (stack+iterative)。这两种方法都是 O(n) 的空间复杂度。

而 Morris Traversal 只需要 O(1) 的空间复杂度。这种算法跟线索二叉树很像，不过 Morris Traversal 一边建线索，一边访问数据，访问完后销毁线索，保持二叉树不变。

4.3.1 Morris 中序遍历

Morris 中序遍历的步骤如下：

1. 初始化当前节点 cur 为 root 节点

2. 如果 cur 没有左孩子，则输出当前节点并将其右孩子作为当前节点，即 cur = cur->rchild。

3. 如果 cur 有左孩子，则寻找 cur 的前驱，即 cur 的左子树的最右下角结点。
   a) 如果前驱节点的右孩子为空，将它的右孩子指向当前节点，当前节点更新为当前节点的左孩子。
   b) 如果前驱节点的右孩子为当前节点，将它的右孩子重新设为空（恢复树的形状），输出当前节点，当前节点更新为当前节点的右孩子。

4. 重复 2、3 步骤，直到 cur 为空。

如图 4-2所示，cur 表示当前节点，深色节点表示该节点已输出。

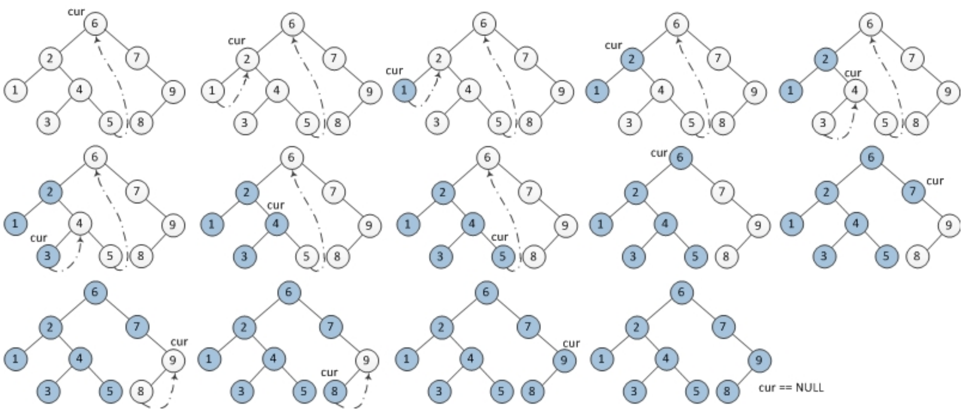


图 4-2 Morris 中序遍历

C 语言实现见第 §4.3.4节。

相关的题目

- Leet Code - Binary Tree Inorder Traversal, http://leetcode.com/onlinejudge#question_94

4.3.2 Morris 先序遍历

Morris 先序遍历的步骤如下：

1. 初始化当前节点 cur 为 root 节点

2. 如果 cur 没有左孩子，则输出当前节点并将其右孩子作为当前节点，即 cur = cur->rchild。

3. 如果 cur 有左孩子，则寻找 cur 的前驱，即 cur 的左子树的最右下角结点。
   a) 如果前驱节点的右孩子为空，将它的右孩子指向当前节点，**输出当前节点（在这里输出，这是与中序遍历唯一的不同点）**当前节点更新为当前节点的左孩子。
   b) 如果前驱节点的右孩子为当前节点，将它的右孩子重新设为空（恢复树的形状），~~输出当前节点，~~当前节点更新为当前节点的右孩子。

4. 重复 2、3 步骤，直到 cur 为空。

如图 4-3所示。

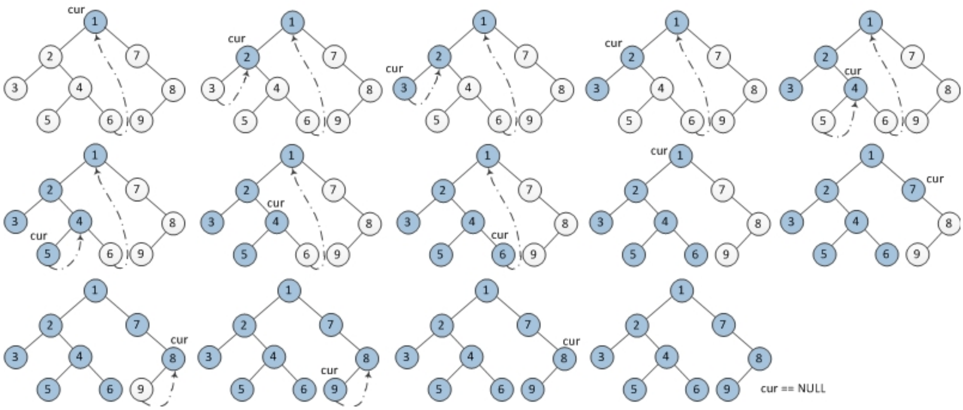


图 4-3 Morris 先序遍历

C 语言实现见第 §4.3.4节。

4.3.3 Morris 后序遍历

Morris 后续遍历稍微复杂，需要建立一个临时节点 dump，令其左孩子是 root，并且还需要一个子过程，就是倒序输出某两个节点之间路径上的所有节点。

Morris 后序遍历的步骤如下：

1. 初始化当前节点 cur 为 root 节点

2. 如果 cur 没有左孩子，则~~输出当前节点并~~将其右孩子作为当前节点，即 cur = cur->rchild。

3. 如果 cur 有左孩子，则寻找 cur 的前驱，即 cur 的左子树的最右下角结点。

   a) 如果前驱节点的右孩子为空，将它的右孩子指向当前节点，当前节点更新为当前节点的左孩子。

   b) 如果前驱节点的右孩子为当前节点，将它的右孩子重新设为空（恢复树的形状），~~输出当前节点，~~**倒序输出从当前节点的左孩子到该前驱节点这条路径上的所有节点。**当前节点更新为当前节点的右孩子。

4. 重复 2、3 步骤，直到 cur 为空。

如图 4-4所示。

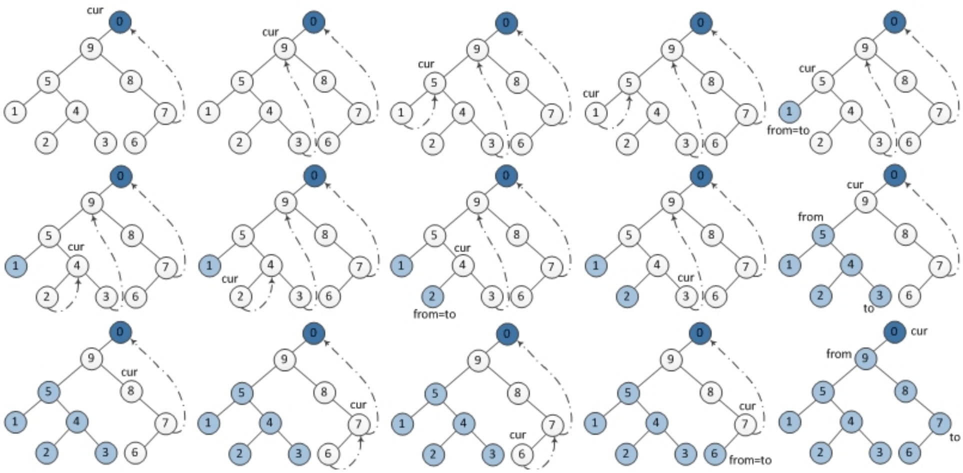


图 4-4 Morris 后序遍历

C 语言实现见第 §4.3.4节。

4.3.4 C 语言实现

morris_traversal.cpp

```
/** @file morris_traversal.c
 * @brief Morris 遍历算法.
 * @author soulmachine@gmail.com
 * @date 2013-06-16
 */
#include<stdio.h>
#include<stdlib.h>

/* 结点数据的类型. */
typedef int elem_t;

/**
```

```
 *@struct
 *@brief 二叉树结点.
 */
typedef struct bt_node_t {
    elem_t data; /* 节点的数据 */
    struct bt_node_t *lchild; /* 左孩子 */
    struct bt_node_t *rchild; /* 右孩子 */
} bt_node_t;

/**
 * @brief 中序遍历, Morris 算法.
 * @param[in] root 根节点
 * @param[in] visit 访问函数
 * @return 无
 */
void in_order_morris(bt_node_t *root, int(*visit)(elem_t*)) {
    bt_node_t *cur, *pre;

    cur = root;
    while (cur != NULL ) {
        if (cur->lchild == NULL ) {
            visit(cur->data);
            cur = cur->rchild;
        } else {
            /* 查找前驱 */
            pre = cur->lchild;
            while (pre->rchild != NULL && pre->rchild != cur)
                pre = pre->rchild;

            if (pre->rchild == NULL ) {    /* 还没线索化, 则建立线索 */
                pre->rchild = cur;
                cur = cur->lchild;
            } else {    /* 已经线索化, 则访问节点, 并删除线索  */
                visit(cur->data);
                pre->rchild = NULL;
                cur = cur->rchild;
            }
        }
    }
}

/**
 * @brief 先序遍历, Morris 算法.
 * @param[in] root 根节点
 * @param[in] visit 访问函数
 * @return 无
 */
void pre_order_morris(bt_node_t *root, int (*visit)(elem_t*)) {
    bt_node_t *cur = root, *pre = NULL;
    while (cur != NULL ) {
        if (cur->lchild == NULL ) {
            visit(cur->data);
            cur = cur->rchild;
```

```
        } else {
            pre = cur->lchild;
            while (pre->rchild != NULL && pre->rchild != cur)
                pre = pre->rchild;

            if (pre->rchild == NULL ) {
                visit(cur->data);  // 仅这一行与中序不同
                pre->rchild = cur;
                cur = cur->lchild;
            } else {
                pre->rchild = NULL;
                cur = cur->rchild;
            }
        }
    }
}

static void reverse(bt_node_t *from, bt_node_t *to);
static void visit_reverse(bt_node_t* from, bt_node_t *to, int (*visit)(elem_t*));
/**
 * @brief 后序遍历, Morris 算法.
 * @param[in] root 根节点
 * @param[in] visit 访问函数
 * @return 无
 */
void post_order_morris(bt_node_t *root, int (*visit)(elem_t*)) {
    bt_node_t dump = { 0, NULL, NULL };
    dump.lchild = root;
    bt_node_t *cur = &dump, *pre = NULL;
    while (cur != NULL ) {
        if (cur->lchild == NULL ) {
            cur = cur->rchild;
        } else {
            pre = cur->lchild;
            while (pre->rchild != NULL && pre->rchild != cur)
                pre = pre->rchild;

            if (pre->rchild == NULL ) {
                pre->rchild = cur;
                cur = cur->lchild;
            } else {
                visit_reverse(cur->lchild, pre, visit);  // call print
                pre->rchild = NULL;
                cur = cur->rchild;
            }
        }
    }
}

/*
 * @brief 逆转路径.
 * @param[in] from from
 * @param[to] to to
```

```
 * @return 无
 */
static void reverse(bt_node_t *from, bt_node_t *to) {
    if (from == to) return;
    bt_node_t *x = from, *y = from->rchild, *z;
    while (x != to) {
        z = y->rchild;
        y->rchild = x;
        x = y;
        y = z;
    }
}

/*
 * @brief  访问逆转后的路径上的所有结点.
 * @param[in] from from
 * @param[to] to to
 * @return 无
 */
static void visit_reverse(bt_node_t* from, bt_node_t *to, int (*visit)(elem_t*)) {
    reverse(from, to);

    bt_node_t *p = to;
    while (1) {
        visit(p->data);
        if (p == from)
            break;
        p = p->rchild;
    }

    reverse(to, from);
}

/*
 * @brief 分配一个新节点.
 * @param[in] data 新节点的数据
 * @return 新节点
 */
bt_node_t* new_node(int data) {
    bt_node_t* node = (bt_node_t*) malloc(sizeof(bt_node_t));
    node->data = data;
    node->lchild = NULL;
    node->rchild = NULL;

    return (node);
}

static int print(const elem_t *data) {
    printf(" %d ", data);
    return 0;
}

/* test */
```

```
int main() {
    /* 构造的二叉树如下
       1
     /   \
    2     3
  /  \
4     5
     */
    bt_node_t *root = new_node(1);
    root->lchild = new_node(2);
    root->rchild = new_node(3);
    root->lchild->lchild = new_node(4);
    root->lchild->rchild = new_node(5);

    in_order_morris(root, print);
    printf("\n");
    pre_order_morris(root, print);
    printf("\n");
    post_order_morris(root, print);

    return 0;
}
```

morris_traversal.cpp

4.4 重建二叉树

binary_tree_rebuild.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stddef.h>
/**
 * @brief 给定前序遍历和中序遍历，输出后序遍历.
 *
 * @param[in] pre 前序遍历的序列
 * @param[in] in 中序遍历的序列
 * @param[in] n 序列的长度
 * @param[out] post 后续遍历的序列
 * @return 无
 */
void build_post(const char * pre, const char *in, const int n, char *post) {
    if(n <= 0) return;
    int left_len = strchr(in, pre[0]) - in;
    build_post(pre + 1, in, left_len, post);
    build_post(pre + left_len + 1, in + left_len + 1,
            n - left_len - 1, post + left_len);
    post[n - 1] = pre[0];
}

#define MAX  64
// 测试
// BCAD CBAD, 输出 CDAB
```

```
// DBACEGF ABCDEFG, 输出 ACBFGED
void build_post_test() {
    char pre[MAX] = {0};
    char in[MAX] = {0};
    char post[MAX] = {0};
    scanf("%s%s", pre, in);

    const int n = strlen(pre);
    build_post(pre, in, n, post);
    printf("%s\n", post);
}

/* 结点数据的类型. */
typedef char elem_t;

/**
 *@struct
 *@brief 二叉树结点.
 */
typedef struct bt_node_t {
    elem_t data; /* 节点的数据 */
    struct bt_node_t *lchild; /* 左孩子 */
    struct bt_node_t *rchild; /* 右孩子 */
} bt_node_t;

/**
 * @brief 给定前序遍历和中序遍历，重建二叉树.
 *
 * @param[in] pre 前序遍历的序列
 * @param[in] in 中序遍历的序列
 * @param[in] n 序列的长度
 * @param[out] root 根节点
 * @return 无
 */
void rebuild(const char *pre, const char *in, int n, bt_node_t **root) {
    // 检查终止条件
    if (n <= 0 || pre == NULL || in == NULL)
        return;
    //获得前序遍历的第一个结点
    *root = (bt_node_t*) malloc(sizeof(bt_node_t));
    (*root)->data = *pre;
    (*root)->lchild = NULL;
    (*root)->rchild = NULL;

    int left_len = strchr(in, pre[0]) - in;
    //重建左子树
    rebuild(pre + 1, in, left_len, &((*root)->lchild));
    //重建右子树
    rebuild(pre + left_len + 1, in + left_len + 1, n - left_len - 1,
            &((*root)->rchild));
}

void print_post_order(const bt_node_t *root) {
```

```
    if(root != NULL) {
        print_post_order(root->lchild);
        print_post_order(root->rchild);
        printf("%c", root->data);
    }
}

void rebuild_test() {
    char pre[MAX] = { 0 };
    char in[MAX] = { 0 };
    scanf("%s%s", pre, in);
    const int n = strlen(pre);

    bt_node_t *root;
    rebuild(pre, in, n, &root);
    print_post_order(root);
}

int main() {
    build_post_test();
    rebuild_test();
    return 0;
}
```

binary_tree_rebuild.c

4.5 堆

4.5.1 堆的 C 语言实现

C++ 可以直接使用 `std::priority_queue`。

heap.c

```
/** @file heap.c
 * @brief 堆，默认为小根堆，即堆顶为最小.
 * @author soulmachine@gmail.com
 * @date 2010-8-1
 * @version 1.0
 */
#include <stdlib.h>  /* for malloc() */
#include <string.h>  /* for memcpy() */

typedef int heap_elem_t; // 元素的类型

/**
 * @struct
 * @brief 堆的结构体
 */
typedef struct heap_t {
    int     size;   /** 实际元素个数 */
    int     capacity; /** 容量，以元素为单位 */
    heap_elem_t  *elems;   /** 堆的数组 */
```

```
    int (*cmp)(const heap_elem_t*, const heap_elem_t*);   /** 元素的比较函数 */
}heap_t;


/** 基本类型（如 int, long, float, double）的比较函数 */
int cmp_int(const int *x, const int *y) {
    const int sub = *x - *y;
    if(sub > 0) {
        return 1;
    } else if(sub < 0) {
        return -1;
    } else {
        return 0;
    }
}

/**
 * @brief 堆的初始化.
 * @param[out] h 堆对象的指针
 * @param[out] capacity 初始容量
 * @param[in] cmp cmp 比较函数，小于返回-1，等于返回 0
 * 大于返回 1，反过来则是大根堆
 * @return 成功返回 0，失败返回错误码
 */
int heap_init(heap_t *h, const int capacity,
              int (*cmp)(const heap_elem_t*, const heap_elem_t*)) {
    h->size = 0;
    h->capacity = capacity;
    h->elems = (heap_elem_t*)malloc(capacity * sizeof(heap_elem_t));
    h->cmp = cmp;

    return 0;
}

/**
 * @brief 释放堆.
 * @param[inout] h 堆对象的指针
 * @return 成功返回 0，失败返回错误码
 */
int heap_uninit(heap_t *h) {
    h->size = 0;
    h->capacity = 0;
    free(h->elems);
    h->elems = NULL;
    h->cmp = NULL;

    return 0;
}


/**
 * @brief 判断堆是否为空.
 * @param[in] h 堆对象的指针
```

```
 * @return 是空, 返回 1, 否则返回 0
 */
int heap_empty(const heap_t *h) {
    return h->size == 0;
}

/**
 * @brief 获取元素个数.
 * @param[in] s 堆对象的指针
 * @return 元素个数
 */
int heap_size(const heap_t *h) {
    return h->size;
}

/*
 * @brief 小根堆的自上向下筛选算法.
 * @param[in] h 堆对象的指针
 * @param[in] start 开始结点
 * @return 无
 */
void heap_sift_down(const heap_t *h, const int start) {
    int i = start;
    int j;
    const heap_elem_t tmp = h->elems[start];

    for(j = 2 * i + 1; j < h->size; j = 2 * j + 1) {
        if(j < (h->size - 1) &&
            // h->elems[j] > h->elems[j + 1]
            h->cmp(&(h->elems[j]), &(h->elems[j + 1])) > 0) {
                j++; /* j 指向两子女中小者 */
        }
        // tmp <= h->data[j]
        if(h->cmp(&tmp, &(h->elems[j])) <= 0) {
            break;
        } else {
            h->elems[i] = h->elems[j];
            i = j;
        }
    }
    h->elems[i] = tmp;
}

/*
 * @brief 小根堆的自下向上筛选算法.
 * @param[in] h 堆对象的指针
 * @param[in] start 开始结点
 * @return 无
 */
void heap_sift_up(const heap_t *h, const int start) {
    int j = start;
    int i= (j - 1) / 2;
    const heap_elem_t tmp = h->elems[start];
```

```
    while(j > 0) {
        // h->data[i] <= tmp
        if(h->cmp(&(h->elems[i]), &tmp) <= 0) {
            break;
        } else {
            h->elems[j] = h->elems[i];
            j = i;
            i = (i - 1) / 2;
        }
    }
    h->elems[j] = tmp;
}

/**
 * @brief 添加一个元素.
 * @param[in] h 堆对象的指针
 * @param[in] x 要添加的元素
 * @return 无
 */
void heap_push(heap_t *h, const heap_elem_t x) {
    if(h->size == h->capacity) { /* 已满，重新分配内存 */
        heap_elem_t* tmp =
            (heap_elem_t*)realloc(h->elems, h->capacity * 2 * sizeof(heap_elem_t));
        h->elems = tmp;
        h->capacity *= 2;
    }

    h->elems[h->size] = x;
    h->size++;

    heap_sift_up(h, h->size - 1);
}

/**
 * @brief 弹出堆顶元素.
 * @param[in] h 堆对象的指针
 * @return 无
 */
void heap_pop(heap_t *h) {
    h->elems[0] = h->elems[h->size - 1];
    h->size --;
    heap_sift_down(h, 0);
}

/**
 * @brief 获取堆顶元素.
 * @param[in] h 堆对象的指针
 * @return 堆顶元素
 */
heap_elem_t heap_top(const heap_t *h) {
    return h->elems[0];
}
```

heap.c

第 5 章
查找

5.1 折半查找

binary_search.c

```
/** 数组元素的类型 */
typedef int elem_t;
/**
  * @brief 有序顺序表的折半查找算法.
  *
  * @param[in] a 存放数据元素的数组, 已排好序
  * @param[in] n 数组的元素个数
  * @param[in] x 要查找的元素
  * @return 查找成功则返回元素所在下标, 否则返回-1
  */
int binary_search(const elem_t a[], const int n, const elem_t x) {
    int left = 0, right = n -1, mid;
    while(left <= right) {
        mid = left + (right - left) / 2;
        if(x > a[mid]) {
            left = mid + 1;
        } else if(x < a[mid]) {
            right = mid - 1;
        } else {
            return mid;
        }
    }
    return -1;
}
```

binary_search.c

第 6 章 排序

6.1 插入排序

6.1.1 直接插入排序

直接插入排序 (Straight Insertion Sort) 的基本思想是：把数组 a[n] 中待排序的 n 个元素看成为一个有序表和一个无序表，开始时有序表中只包含一个元素 a[0]，无序表中包含有 n-1 个元素 a[1] a[n-1]，排序过程中每次从无序表中取出第一个元素，把它插入到有序表中的适当位置，使之成为新的有序表，这样经过 n-1 次插入后，无序表就变为空表，有序表中就包含了全部 n 个元素，至此排序完毕。在有序表中寻找插入位置是采用从后向前的顺序查找的方法。

直接插入排序的 C 语言实现如下。

straight_insertion_sort.c

```c
/** 数组元素的类型 */
typedef int elem_t;

/**
  * @brief 直接插入排序，时间复杂度 O(n^2).
  *
  * @param[inout] a 待排序元素序列
  * @param[in] start 开始位置
  * @param[in] end 结束位置，最后一个元素后一个位置，即左闭右开区间
  * @return 无
  */
void straight_insertion_sort(elem_t a[], const int start, const int end) {
    elem_t tmp;
    int i, j;

    for (i = start + 1; i < end; i++) {
        tmp = a[i];
        for (j = i - 1; tmp < a[j] && j >= start; j--) {
            a[j + 1] = a[j];
        }
        a[j + 1] = tmp;
    }
}
```

straight_insertion_sort.c

6.1.2 折半插入排序

在查找插入位置时，若改为折半查找，就是**折半插入排序** (Binary Insertion Sort)。

折半插入排序的 C 语言实现如下。

binary_insertion_sort.c

```
/**
  * @brief 折半插入排序，时间复杂度 O(nlog2n).
  *
  * @param[inout] a 待排序元素序列
  * @param[in] start 开始位置
  * @param[in] end 结束位置，最后一个元素后一个位置
  * @return 无
  */
void binary_insertion_sort(elem_t a[], const int start, const int end) {
    elem_t tmp;
    int i, j, left, right, mid;

    for (i = start + 1; i < end; i++) {
        tmp = a[i];
        left = start;
        right = i - 1;
        while (left <= right) {
            mid = (left + right) / 2;
            if (tmp < a[mid]) {
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        for (j = i - 1;j >= left; j--) {
            a[j + 1] = a[j];
        }
        a[left] = tmp;
    }
}
```

binary_insertion_sort.c

6.1.3 希尔 (Shell) 插入排序

从对直接插入排序的分析得知，其算法时间复杂度为 $O(n^2)$，但是，若待排序记录序列为“正序”时，其时间复杂度可提高至 $O(n)$。由此可设想，若待排序记录序列按关键字“基本有序”，即序列中具有下列特性

$$R_i.key < \max\{R_j.key\},\ j < i$$

的记录较少时，直接插入排序的效率就可大大提高，从另一方面来看，由于直接插入排序算法简单，则在 n 值很小时效率也比较高。希尔排序正是从这两点分析出发对直接插入排序进行改进得到的一种插入排序方法。

希尔排序 (Shell Sort) 的基本思想是：设待排序元素序列有 n 个元素，首先取一个整数 $gap = \lfloor \frac{n}{3} \rfloor + 1$ 作为间隔，将全部元素分为 gap 个子序列，所有距离为 gap 的元素放在同一个子序列中，在每一个子序列中分别施行直接插入排序。然后缩小间隔 gap，取 $gap = \lfloor \frac{gap}{3} \rfloor + 1$，重复上述的子序列划分和排序工作，直到最后取 $gap = 1$，将所有元素放在同一个序列中排序为止。

图 6-1展示了希尔排序的过程。

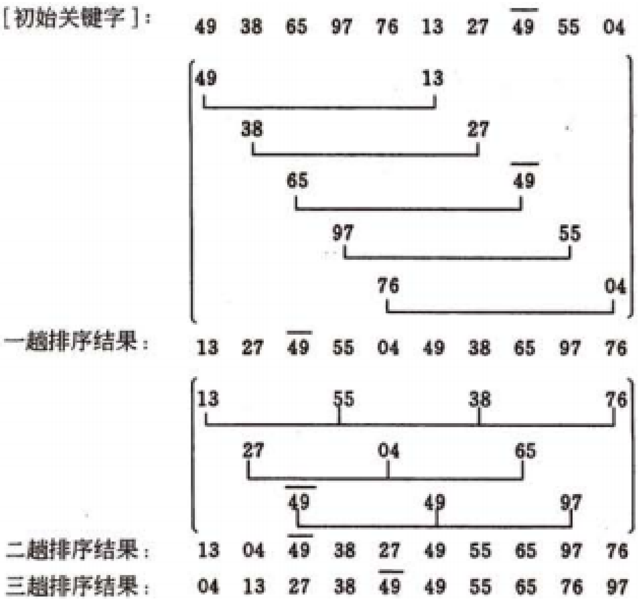


图 6-1 希尔排序

希尔排序的 C 语言实现如下。

shell_sort.c

```
/*
  * @brief 一趟希尔插入排序.
  *
  * 和一趟直接插入排序相比，仅有一点不同，就是前后元素的间距是 gap 而不是 1
  *
  * @param[inout] a 待排序元素序列
  * @param[in] start 开始位置
  * @param[in] end 结束位置，最后一个元素后一个位置，即左闭右开区间
  * @param[in] gap 间隔
  * @return 无
  */
static void shell_insert(elem_t a[], const int start, const int end, const int gap) {
    elem_t tmp;
    int i, j;
    for (i = start + gap; i < end; i++) {
```

```
        tmp = a[i];
        for (j = i - gap; tmp < a[j] && j >= start; j -= gap) {
            a[j + gap] = a[j];
        }
        a[j + gap] = tmp;
    }
}

 /*
  * @brief 希尔排序.
  * @param[inout] a 待排序元素序列
  * @param[in] start 开始位置
  * @param[in] end 结束位置，最后一个元素后一个位置，即左闭右开区间
  * @return 无
  */
void shell_sort(elem_t a[], const int start, const int end) {
    int gap = end - start;
    while (gap > 1) {
        gap = gap / 3 + 1;
        shell_insert(a, start, end, gap);
    }
}
```
shell_sort.c

6.2 交换排序

6.2.1 冒泡排序

冒泡排序 (Bubble Sort) 的基本方法是：设待排序元素序列的元素个数为 n，从后向前两两比较相邻元素的值，如果发生逆序（即前一个比后一个大），则交换它们，直到序列比较完。我们称它为一趟冒泡，结果是最小的元素交换到待排序序列的第一个位置，其他元素也都向排序的最终位置移动。下一趟冒泡时前一趟确定的最小元素不参加比较，待排序序列减少一个元素，一趟冒泡的结果又把序列中最小的元素交换到待排序序列的第一个位置。这样最多做 n-1 趟冒泡就能把所有元素排好序。

冒泡排序的 C 语言实现如下。

bubble_sort.c
```
/** 数组元素的类型 */
typedef int elem_t;

/**
  * @brief 冒泡排序.
  * @param[inout] a 待排序元素序列
  * @param[in] start 开始位置
  * @param[in] end 结束位置，最后一个元素后一个位置，即左闭右开区间
  * @return 无
  * @note 无
  * @remarks 无
```

```
  */
void bubble_sort(elem_t a[], const int start, const int end) {
    int exchange; /* 是否发生交换 */
    elem_t tmp;
    int i, j;

    for (i = start; i < end - 1; i++) {
        exchange = 0;
        for (j = end - 1; j > i; j--) { /* 发生逆序，交换 */
            if (a[j - 1] > a[j]) {
                tmp = a[j - 1];
                a[j - 1] = a[j];
                a[j] = tmp;
                exchange = 1;
            }
        }
        if (exchange == 0) return; /* 本趟无逆序，停止处理 */
    }
}
```

bubble_sort.c

6.2.2 快速排序

快速排序 (Quick sort) 的基本思想是任取待排序元素序列中的某个元素（例如取第一个元素）作为基准，按照该元素的关键字大小，将整个元素序列划分为左右两个子序列：左侧子序列中所有元素的关键字都小于基准元素的关键字，右侧子序列中所有元素的关键字都大于或等于基准元素的关键字，基准元素则排在这两个子序列中间（这也是该元素最终应该安放的位置)。然后分别对这两个子序列重复施行上述算法，直到所有的元素都排在相应位置为止。

一趟快排的过程如图 6-2 (a) 所示。整个快速排序的过程可递归，如图 6-2 (b) 所示。

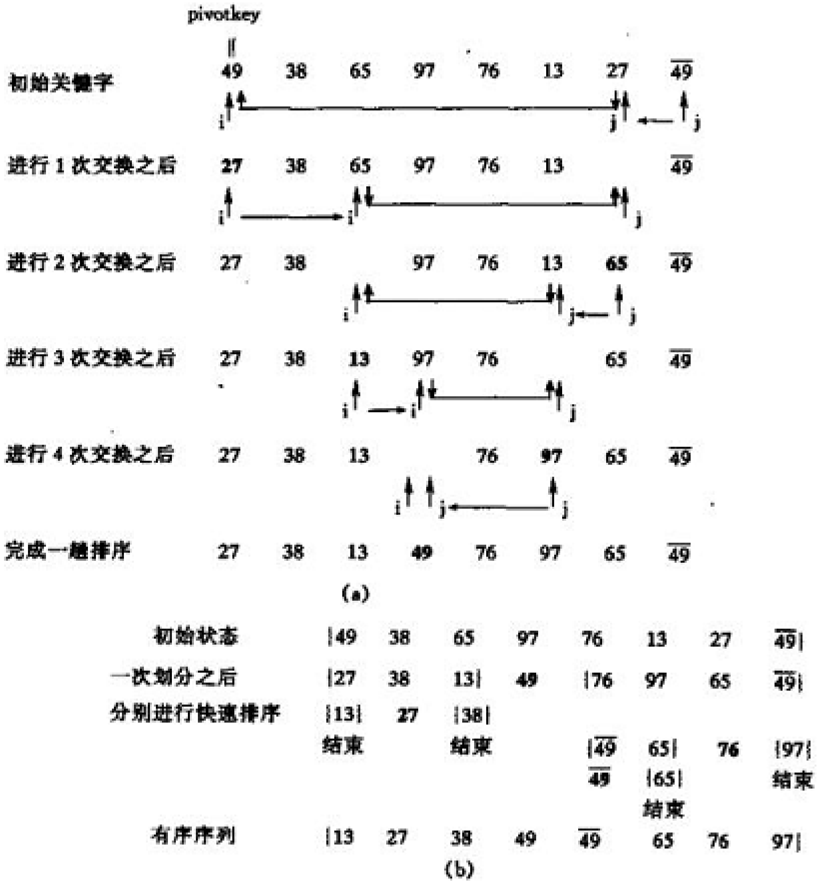


图 6-2 快速排序示例

更多详细解释请参考本项目的 wiki，https://github.com/soulmachine/acm-cheatsheet/wiki/快速排序

快速排序的 C 语言实现如下。

quick_sort.c

```
/** 数组元素的类型 */
typedef int elem_t;
 /*
  * @brief 一趟划分.
  * @param[inout] a 待排序元素序列
  * @param[in] start 开始位置
  * @param[in] end 结束位置，最后一个元素后一个位置，即左闭右开区间
  * @return 基准元素的新位置
  */
int partition(elem_t a[], const int start, const int end) {
    int i = start;
```

```c
    int j = end - 1;
    const elem_t pivot = a[i];

    while(i < j) {
        while(i < j && a[j] >= pivot) j--;
        a[i] = a[j];
        while(i < j && a[i] <= pivot) i++;
        a[j] = a[i];
    }
    a[i] = pivot;
    return i;
}

/**
  * @brief 快速排序.
  * @param[inout] a 待排序元素序列
  * @param[in] start 开始位置
  * @param[in] end 结束位置, 最后一个元素后一个位置
  * @return 无
  */
void quick_sort(elem_t a[], const int start, const int end) {
    if(start < end - 1) { /* 至少两个元素 */
        const int pivot_pos = partition(a, start, end);
        quick_sort(a, start, pivot_pos);
        quick_sort(a, pivot_pos + 1, end);
    }
}
```

quick_sort.c

6.3 选择排序

选择排序 (Selection sort) 的基本思想是：每一趟在后面 n-i(i=1, 2, ..., n-2) 个元素中选取最小的元素作为有序序列的第 i 个元素。

6.3.1 简单选择排序

简单选择排序 (simple selection sort) 也叫直接选择排序 (straight selection sort)，其基本步骤是：

- 在一组元素 a[i] a[n-1] 中选择最小的元素；
- 若它不是这组元素中的第一个元素，则将它与这组元素的第一个元素对调；
- 在剩下的 a[i+1] a[n-1] 中重复执行以上两步，直到剩余元素只有一个为止。

简单选择排序的 C 语言实现如下。

simple_selection_sort.c

```c
/** 数组元素的类型 */
typedef int elem_t;
```

```c
/**
  * @brief 简单选择排序.
  * @param[inout] a 待排序元素序列
  * @param[in] start 开始位置
  * @param[in] end 结束位置，最后一个元素后一个位置，即左闭右开区间
  * @return 无
  */
void simple_selection_sort(elem_t a[], int start, int end) {
    elem_t tmp;
    int i, j, k;

    for (i = start; i < end; i++) {
        k = i;
        /* 在 a[i] 到 a[end-1] 中寻找最小元素 */
        for (j = i + 1; j < end; j++)
            if(a[j] < a[k]) k = j;
        /* 交换 */
        if (k != i) {
            tmp = a[i];
            a[i] = a[k];
            a[k]= tmp;
        }
    }
}
```

simple_selection_sort.c

6.3.2 堆排序

堆排序的 C 语言实现如下。

heap_sort.c

```c
#include "heap.c"
/**
  * @brief 堆排序.
  * @param[inout] a 待排序元素序列
  * @param[in] n 元素个数
  * @param[in] cmp cmp 比较函数，小于返回-1，等于，大于
  * @return 无
  */
void heap_sort(heap_elem_t *a, const int n,
               int (*cmp)(const heap_elem_t*, const heap_elem_t*)) {
    int i;
    heap_t h;
    heap_elem_t tmp;

    /* 替代 heap_init() */
    h.size = n;
    h.capacity = n;
    h.elems = a;
    h.cmp = cmp;
```

```
    i = (h.size - 2)/2;   /* 找最初调整位置: 最后分支结点 */
    while (i >= 0) {  /* 自底向上逐步扩大形成堆 */
        heap_sift_down(&h, i);
        i--;
    }

    for (i = h.size - 1; i > 0; i--) {
        tmp = h.elems[i];
        h.elems[i] = h.elems[0];
        h.elems[0] = tmp;
        h.size = i; /* 相当于 h.size -- */
        heap_sift_down(&h, 0);
    }
}
```

heap_sort.c

6.4 归并排序

所谓“归并”，就是将两个或两个以上的有序序列合并成一个有序序列。我们先从最简单的二路**归并排序** (Merge sort) 入手。

图 6-3是一个二路归并排序的例子。

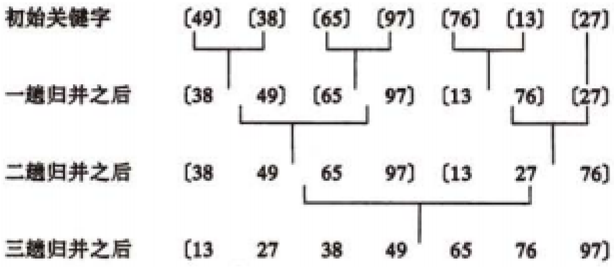


图 6-3 二路归并排序示例

二路归并排序的 C 语言实现如下。

heap_sort.c

```
/** 数组元素的类型 */
typedef int elem_t;

 /*
  * @brief 将两个有序表合并成一个新的有序表
  * @param[inout] a 待排序元素序列, 包含两个有序表
  * @param[in] tmp 与 a 等长的辅助数组
  * @param[in] start a[start]~a[mid-1] 为第一个有序表
  * @param[in] mid 分界点
  * @param[in] end a[mid]~a[end-1] 为第二个有序表
  * @return 无
  */
```

```
static void merge(elem_t a[], elem_t tmp[], const int start, const int mid, const int end) {
    int i, j, k;
    for (i = 0; i < end; i++) tmp[i] = a[i];

    /* i, j 是检测指针，k 是存放指针 */
    for (i = start, j = mid, k = start; i < mid && j < end; k++) {
        if (tmp[i] < tmp[j]) {
            a[k] = tmp[i++];
        } else {
            a[k] = tmp[j++];
        }
    }
    /* 若第一个表未检测完，复制 */
    while (i < mid) a[k++] = tmp[i++];
    /* 若第二个表未检测完，复制 */
    while (j < end) a[k++] = tmp[j++];
}

 /*
  * @brief 归并排序.
  * @param[inout] a 待排序元素序列
  * @param[in] tmp 与 a 等长的辅助数组
  * @param[in] start 开始位置
  * @param[in] end 结束位置，最后一个元素后一个位置，即左闭右开区间
  * @return 无
  * @note 无
  * @remarks 无
  */
void merge_sort(elem_t a[], elem_t tmp[], const int start, const int end) {
    if (start < end - 1) {
        const int mid = (start + end) / 2;
        merge_sort(a, tmp, start, mid);
        merge_sort(a, tmp, mid, end);
        merge(a, tmp, start, mid, end);
    }
}
```

heap_sort.c

6.5 基数排序

利用多关键字实现对单关键字排序的算法就称为**基数排序** (Radix sort)。

有两种顺序，最高位优先 MSD(Most Significant Digit first) 和最低位优先 LSD(Least Significant Digit first)。

下面介绍“LSD 链式基数排序”。首先以静态链表存储 n 个待排元素，并令表头指针指向第一个元素，即 A[1] 到 A[n] 存放元素，A[0] 为表头结点，这样元素在重排时不必移动元素，只需要修改各个元素的 link 指针即可，如图 6-4(a) 所示。每个位设置一个桶（跟散列桶一样），桶采用静态链表结构，同时设置两个数组 f[RADIX] 和 r[RADIX]，记录每

个桶的头指针和尾指针。排序过程就是 d（关键字位数）趟“分配”、“收集”的过程，如图 6-4所示。

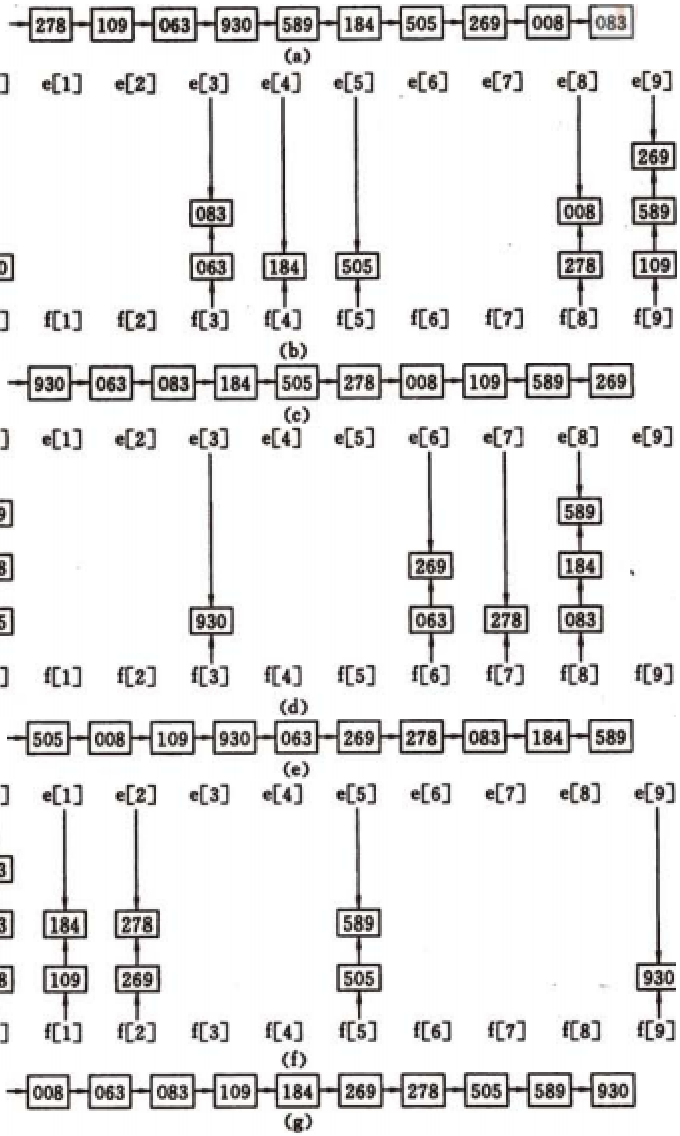


图 6-4 LSD 链式基数排序示例

LSD 链式基数排序的 C 语言实现如下。

radix_sort.c

```
/** @file radix_sort.c
 * @brief LSD 链式基数排序.
```

```
  * @author soulmachine@gmail.com
  * @date 2013-05-18
  */
#include <stdio.h>  /* for printf() */

/* 关键字基数，此时是十进制 */
#define R 10  /*Radix*/

/**
  *@struct
  *@brief 静态链表结点.
  */
typedef struct static_list_node_t {
    int key; /** 关键字 */
    int link; /** 下一个节点 */
}static_list_node_t;

 /*
  * @brief 打印静态链表.
  * @param[in] a 静态链表数组
  * @return 无
  */
static void static_list_print(const static_list_node_t a[]) {
    int i = a[0].link;
    while (i != 0) {
        printf("%d ", a[i].key);
        i = a[i].link;
    }
}

 /*
  * @brief 获取十进制整数的某一位数字.
  * @param[in] n 整数
  * @param[in] i 第 i 位
  * @return 整数 n 第 i 位的数字
  */
static int get_digit(int n, const int i) {
    int j;
    for(j = 1; j < i; j++) {
        n /= 10;
    }

    return n % 10;
}

/**
  * @brief LSD 链式基数排序.
  * @param[in] a 静态链表，a[0] 是头指针
  * @param[in] n 待排序元素的个数
  * @param[in] d 最大整数的位数
  * @return 无
  * @note 无
  * @remarks 无
```

```c
 */
void radix_sort(static_list_node_t a[], const int n, const int d) {
    int i, j, k, current, last;
    int rear[R], front[R];

    for(i = 0; i < n; i++) a[i].link = i + 1;
    a[n].link = 0;
    for(i = 0; i < d; i++) {
        /* 分配 */
        for(j = 0; j < R; j++) front[j] = 0;
        for(current = a[0].link; current != 0;
            current = a[current].link) {
            k = get_digit(a[current].key, i + 1);
            if(front[k] == 0) {
                front[k] = current;
                rear[k] = current;
            } else {
                a[rear[k]].link = current;
                rear[k] = current;
            }
        }

        /* 收集 */
        j = 0;
        while(front[j] == 0) j++;
        a[0].link = current = front[j];
        last = rear[j];
        for(j = j + 1; j < R; j++) {
            if(front[j] != 0) {
                a[last].link = front[j];
                last = rear[j];
            }
        }
        a[last].link = 0;
    }
}

void radix_sort_test(void) {
    static_list_node_t a[] = {{0,0}/* 头指针 */, {278,0}, {109,0},
    {63,0}, {930,0}, {589,0}, {184,0}, {505,0}, {269,0},
    {8,0}, {83,0}};
    radix_sort(a, 10, 3);
    static_list_print(a);
}
```

radix_sort.c

6.6 总结和比较

表 6-1 各种排序算法的总结和比较

排序方法	**平均时间**	**最坏情况**	**辅助存储**	**是否稳定**
直接插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	是
折半插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	是
希尔排序	N/A	N/A	$O(1)$	否
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	是
快速排序	$O(n\log_2 n)$	$O(n^2)$	$O(\log_2 n)$	否
简单选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	否
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(1)$	否
二路归并	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n)$	是
基数排序	$O(d\times(n+R))$	$O(d\times(n+R))$	$O(R)$	是

假设在数组中有两个元素 $A_i, A_j, i<j$，即 A_i 在 A_j 之前，且 $A_i = A_j$，如果在排序之后，A_i 仍然在 A_j 的前面，则称这个排序算法是**稳定**的，否则称这个排序算法是**不稳定**的。

排序方法根据在排序过程中数据是否完全在内存，分为两大类：**内部排序**和**外部排序**。内部排序是指在排序期间数据全部存放在内存；外部排序是指在排序期间所有数据不能同时存放在内存，在排序过程中需要不断在内、外存之间交换。一般说到排序，默认是指内部排序。

第 7 章
暴力枚举法

7.1 枚举排列

枚举排列，即输出某个集合的所有排列。例如，集合 1,2,3 的所有排列是 (1,2,3),(1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)。

7.1.1 生成 1 到 n 的全排列

描述

给定一个正整数 n，输出 1 到 n 的所有排列。

分析

我们尝试用递归的思路：先输出以 1 开头的排列（这一步是递归调用），然后输出以 2 开头的排列（又是递归调用），...，最后才是以 n 开头的排列。

以 1 开头的排列，第一位是 1，后面是 2 到 9 的排列，这是一个子问题，可以接着递归。

伪代码如下：

```
void print_permutation(序列 P, 集合 S) {
    if (S 为空) 输出序列 P
    else {
        for(按照从小到大的顺序依次考虑 S 中的每个元素 e) {
            print_permutation(在 A 的末尾添加 e 后得到的新序列, S-{e});
        }
    }
}
// 调用
print_permutation({}, S);
```

代码

下面考虑用 C 语言实现。不难想到用数组表示 P 和 S。由于 P 和 S 是互补的，它们二者知道其中给一个，另一个就完全确定了，因此不用保存 P。

C 语言实现如下：

print_permutation_n.c

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

/*
 * @brief 输出 1 到 n 的全排列
 * @param[in] n n
 * @param[in] cur 当前进行到哪个位置
 * @param[out] 存放一个排列
 * @return 无
 */
static void print_permutation_r(int n, int cur, int P[]) {
    int i, j;
    if (cur == n) { // 递归边界
        for (i = 0; i < n; i++)
            printf("%d", P[i]);
        printf("\n");
    } else {
        // 尝试在 A[cur] 中填各种整数 i，按从小到大的顺序
        for (i = 1; i <= n; i++) {
            int ok = 1;
            for (j = 0; j < cur; j++)
                if (P[j] == i)
                    ok = 0; // 如果 i 已经在 A[0]~A[cur-1] 出现过，则不能再选
            if (ok) {
                P[cur] = i;
                print_permutation_r(n, cur + 1, P); // 递归调用
            }
        }
    }
}

/**
 * @brief 输出 1 到 n 的全排列
 * @param[in] n n
 * @return 无
 */
void print_permutation(int n) {
    int *P = (int*)malloc(n * sizeof(int));
    print_permutation_r(n, 0, P);
    free(P);
    return;
}
//test
int main() {
    print_permutation(3);
```

```
    return 0;
}
```

print_permutation_n.c

7.1.2 生成可重集的排列

描述

如果把问题改成，给定一个数组 S，按字典序输出所有排列。

分析

此时 S 不再仅限于整数，可以是字符等。

首先想到可以利用上一题的思路，先输出 1 到 n 的全排列，然后把打印语句改成 `printf("%c", S[P[i]-1])` 即可。

但是这个方法有点问题，当集合中有重复元素时，它不能正确处理。例如 S=”AAA”，它会打印出 6 个 AAA。见下面的代码。

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

/*
 * @brief 输出 1 到 n 的全排列，把数字当做下标
 * @param[in] S 字符集合
 * @param[in] n n
 * @param[in] cur 当前进行到哪个位置
 * @param[out] 存放一个排列
 * @return 无
 */
static void print_permutation_r(char S[], int n, int cur, int P[]) {
    int i, j;
    if (cur == n) { // 递归边界
        for (i = 0; i < n; i++)
            printf("%c", S[P[i]-1]);
        printf("\n");
    } else {
        // 尝试在 A[cur] 中填各种整数 i，按从小到大的顺序
        for (i = 1; i <= n; i++) {
            int ok = 1;
            for (j = 0; j < cur; j++)
                if (P[j] == i)
                    ok = 0; // 如果 i 已经在 A[0]~A[cur-1] 出现过，则不能再选
            if (ok) {
                P[cur] = i;
                print_permutation_r(S, n, cur + 1, P); // 递归调用
            }
        }
    }
```

```
}

/**
 * @brief 输出字符集合的全排列
 * @param[in] S 字符集合
 * @param[in] n n
 * @return 无
 */
void print_permutation(char S[], int n) {
    int *P = (int*)malloc(n * sizeof(int));
    print_permutation_r(S, n, 0, P);
    free(P);
    return;
}

//test
int main() {
    char *S="ABC";
    char *S1="AAA";
    print_permutation(S,3);
    print_permutation(S1,3); // 不能正确处理可重集
    return 0;
}
```

再换一个思路，还是在上一节的代码上进行修改，把 `if(P[j]==i)` 和 `P[cur]=i` 分别改成 `if(P[j]==S[i])` 和 `P[cur]=S[i]`。这样，只要把 S 中的所有元素按从小到大的顺序排序，然后调用 `print_permutation_r(S, n, 0, P)` 即可。

这个方法看上去不错，可惜还是有个小问题。例如 `S="AAA"`，程序什么也不输出（正确答案应该是唯一的全排列 AAA），原因在于，我们禁止 S 数组中出现重复，而在 S 中本来就有重复元素，这个“禁令”是错误的。

解决方法是，统计 P[0] 到 P[cur-1] 中 S[i] 出现的次数 c1，以及 S 数组中 S[i] 的出现次数 c2，只要 c1<c2，就能继续选择 S[i]。

结果又如何呢？这次有输出了，可是输出了 27 个 AAA。遗漏是没有了，可是出现了重复。程序先把第 1 个 A 作为开头，递归调用结束后用第 2 个 A 作为开头，递归调用结束后用第 3 个 A 作为开头。每次输出 3 个排列，共 27 个。

枚举时应该**不重不漏**地取遍集合的所有元素。由于数组已经排序，所以只需要检查当前元素和前一个元素不相同，就可以做到不重不漏了。即只需要在 `for (i = 1; i <= n; i++)` 和其后的花括号之间加上 `if(i==0 && S[i] != S[i-1])`。见下面的代码。

代码

print_permutation.c

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

```
/*
 * @brief 输出集合的全排列
 * @param[in] S 字符集合
 * @param[in] n n
 * @param[in] cur 当前进行到哪个位置
 * @param[out] 存放一个排列
 * @return 无
 */
static void print_permutation_r(char S[], int n, int cur, char P[]) {
    int i, j;
    if (cur == n) { // 递归边界
        for (i = 0; i < n; i++)
            printf("%c", P[i]);
        printf("\n");
    } else {
        // 尝试在 A[cur] 中填各种元素，按从小到大的顺序
        for (i = 0; i < n; i++) if(i == 0 || S[i] != S[i-1]){
            int c1 = 0, c2 = 0;
            for (j = 0; j < cur; j++) if (P[j] == S[i]) ++c1;
            for (j = 0; j < n; j++) if (S[j] == S[i]) ++c2;
            if (c1 < c2) {
                P[cur] = S[i];
                print_permutation_r(S, n, cur + 1, P); // 递归调用
            }
        }
    }
}

/**
 * @brief 输出字符集合的全排列
 * @param[in] S 字符集合
 * @param[in] n n
 * @return 无
 */
void print_permutation(char S[], int n) {
    char *P = (char*)malloc(n * sizeof(char));
    print_permutation_r(S, n, 0, P);
    free(P);
    return;
}

//test
int main() {
    char *S="ABC";
    char *S1="AAA";
    print_permutation(S,3);
    print_permutation(S1,3); // 可以正确处理可重集
    return 0;
}
```
print_permutation.c

7.1.3 下一个排列

还可以利用 STL 中的 `std::next_permutation()` 或第 §16.1节的 `next_permutation()`。

代码

print_permutation_next.c

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

/**
 * @brief 输出字符集合的全排列，利用 std::next_permutation
 * @param[in] S 字符集合
 * @param[in] n n
 * @return 无
 */
void print_permutation(char S[], int n) {
    std::sort(&S[0], &S[n]);
    do {
        for(int i = 0; i < n; i++) printf("%c", S[i]);
        printf("\n");
    }while(std::next_permutation(&S[0], &S[n]));
    return;
}

/* 等价于复制粘贴，这里为了节约篇幅，使用 include，在 OJ 上提交时请用复制粘贴 */
#include "next_permutation.c"

/**
 * @brief 输出字符集合的全排列，利用第 15 章的 next_permutation
 * @param[in] S 字符集合
 * @param[in] n n
 * @return 无
 */
void print_permutation1(char S[], int n) {
    std::sort(&S[0], &S[n]);
    int *N = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; ++i) N[i]=S[i]-'A';
    do {
        for(int i = 0; i < n; i++) printf("%c", S[N[i]]);
        printf("\n");
    }while(std::next_permutation(&N[0], &N[n]));
    return;
}

//test
int main() {
    char S[]="ABC";
    char S1[]="AAA";
    print_permutation(S,3);
```

```
    print_permutation(S1,3);
    printf("\n\n");
    print_permutation1(S,3);
    print_permutation1(S1,3);
    return 0;
}
```

print_permutation_next.c

7.2 子集生成

给定一个集合，输出它所有的子集。为了简单起见，本节讨论的集合中没有重复元素。

7.2.1 增量构造法

一次选出一个元素，放或者不放到集合中。

代码

subset.c

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

/**
 * @brief 增量构造法
 * @param[in] S 输入集合
 * @param[in] n 集合大小
 * @param[inout] P 某个子集
 * @param[in] cur p 的当前位置
 * @param[in] ed S 的当前位置，前面的元素已经选过了
 * @return 无
 */
void print_subset1(int *S, int n, int *P, int cur, int ed) {
    int i, j;
    for (i = ed; i < n; i++) {
        // 选择 S[i]
        P[cur] = S[i];
        for (j = 0; j <= cur; j++) printf("%d ", P[j]);
        printf("\n");
        // 不选择 S[i]
        print_subset1(S, n, P, cur + 1, i + 1);
    }
}
```

subset.c

7.2.2 位向量法

开一个位向量 B，B[i]=1 表示选择 S[i], B[i]=0 表示不选择。

代码

subset.c

```
/**
 * @brief 位向量法
 * @param[in] S 输入集合
 * @param[in] n 集合大小
 * @param[in] B 位向量
 * @param[in] cur B 的当前位置
 * @return 无
 */
void print_subset2(int *S, int n, char *B, int cur) {
    int i;
    if (cur == n) {
        for (i = 0; i < n; i++) if (B[i]) printf("%d ", S[i]);
        printf("\n");
        return;
    }
    B[cur] = 1;
    print_subset2(S, n, B, cur + 1);
    B[cur] = 0;
    print_subset2(S, n, B, cur + 1);
}
```

subset.c

7.2.3 二进制法

前提：集合的元素不超过 int 位数。用一个 int 整数表示位向量，第 i 位为 1，则表示选择 S[i]，为 0 则不选择。例如 S={A,B,C,D}，则 0110=6 表示子集 {B,C}。

这种方法最巧妙。因为它不仅能生成子集，还能方便的表示集合的并、交、差等集合运算。设两个集合的位向量分别为 B1 和 B2，则 B1|B2, B1&B2, B1^B2 分别对应集合的并、交、对称差。

代码

subset.c

```
/**
 * @brief 二进制法
 * @param[in] S 输入集合
 * @param[in] n 集合大小
 * @param[in] B 位向量
 * @param[in] cur B 的当前位置
 * @return 无
 */
```

```
void print_subset3(int *S, int n) {
    int i, j;
    for (i = 1; i < (1 << n); i++) {
        for (j = 0; j < n; j++)
            if (i & (1 << j)) printf("%d ", S[j]);
        printf("\n");
    }
}

int main() {
    int n, i;

    while(scanf("%d",&n) > 0) {
        int *S = (int*)malloc(n * sizeof(int));
        int *P = (int*)malloc(n * sizeof(int));
        char *B = (char*)malloc(n * sizeof(char));

        for(i = 0; i < n; i++) scanf("%d",&S[i]);

        print_subset1(S, n, P, 0, 0); putchar('\n');
        print_subset2(S, n, B, 0); putchar('\n');
        print_subset3(S, n);

        free(S);
        free(P);
        free(B);
    }
    return 0;
}
```

subset.c

第 8 章 分治法

二分查找，快速排序，归并排序，都属于分治法 (Divide and Conquer)。

8.1 棋盘覆盖

描述

在一个 $2^k \times 2^k (1 \leq k \leq 100)$ 的棋盘中，恰有一个方格被黑色覆盖，其他为白色。用黑色的 L 型牌（如图 8-1所示为 4 种 L 型牌），去覆盖棋盘中所有的白色方格，黑色方格不能被覆盖，且任意两个 L 型牌不能重叠（即不重不漏）。求所需 L 型牌的总数。

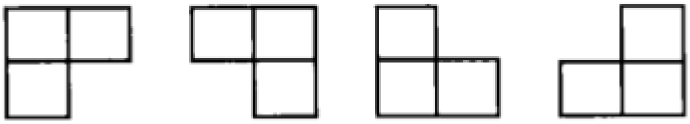


图 8-1　4 种 L 型牌

输入

第一行包含一个整数 T，表示有 T 组测试用例。

每一组测试用例占用一行，包含一个整数 k。

输出

所需 L 型牌的总数

样例输入

```
3
1
2
3
```

样例输出

```
1
5
21
```

分析

本题的棋盘是 $2^k \times 2^k$，很容易想到用分治法。把棋盘切成 4 块，则每一块都是 $2^{k-1} \times 2^{k-1}$ 的。有黑格的那一块可以递归解决，但其他 3 块并没有黑格子，应该怎么办呢？可以构造出一个黑格子，如图 8-2所示，在中心放一个 L 型牌，其它 3 块也变成了子问题。递归边界不难得出，当 $k = 1$ 时 1 块 L 型牌就够了。

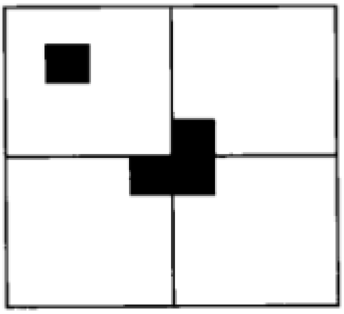


图 8-2 棋盘覆盖问题的递归解法

本题只需要求总数，不需要求具体怎么摆放，因此简化很多。根据上面的思路，设 $f(k)$ 表示棋盘是 $2^k \times 2^k$ 时所需 L 型牌的总数，可得递推公式 $f(k) = 4f(k-1) + 1$。

注意，2^100 是一个很大的数，本题需要处理大数，见 §14.3节。

代码

chessboard_cover.c

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

#define MAXK 100

/* 一个数组元素表示 4 个十进制位，即数组是万进制的 */
#define BIGINT_MOD 10000
#define MOD_LEN 4
#define MAX_LEN (61/MOD_LEN+1)  /* 整数的最大位数，10^x > 4^100 */

int  d[MAXK][MAX_LEN * 2];  /* d[k-1] = f(k) */

/**
 * @brief 打印大整数.
```

```
 * @param[in] x 大整数，用数组表示，低位在低地址
 * @param[in] n 数组 x 的长度
 * @return 无
 */
void bigint_print(const int x[], const int n) {
    int i;
    int start_output = 0;  /* 用于跳过前导 0 */
    for (i = n - 1; i >= 0; --i) {
        if (start_output) {  /* 如果多余的 0 已经都跳过，则输出 */
            printf("%04d", x[i]);
        } else if (x[i] > 0) {
            printf("%d", x[i]);  /* 本题输出比较坑爹，最高位数字有前导 0 */
            start_output = 1; /* 碰到第一个非 0 的值，就说明多余的 0 已经都跳过 */
        }
    }

    if(!start_output) printf("0");  /* 当 x 全为 0 时 */
}

/**
 * @brief 计算 f(k) = 4f(k-1)+1，与大整数乘法很类似.
 * @param[in] x x
 * @param[in] y y
 * @param[out] z z=x*y+1
 * @return 无
 */
void bigint_mul_small(const int x[], const int y, int z[]) {
    int i;
    for (i = 0; i < MAX_LEN * 2; i++) z[i] = 0;

    z[0] = 1;

    for (i = 0; i < MAX_LEN; i++) z[i] += x[i] * y;

    for (i = 0; i < MAX_LEN * 2; i++) {  /* 统一处理进位问题 */
        if (z[i] >= BIGINT_MOD) {  /* 看是否要进位 */
            z[i+1] += z[i] / BIGINT_MOD;  /* 进位 */
            z[i] %= BIGINT_MOD;
        }
    }
}

int main() {
    int k, T;
    d[0][0] = 1;
    for (k = 2; k <= 100; k++) bigint_mul_small(d[k-2], 4, d[k-1]);

    scanf("%d", &T);
    while(T-- > 0) {
        scanf("%d", &k);
        bigint_print(d[k - 1], MAX_LEN * 2);
        printf("\n");
    }
```

```
    return 0;
}
```
chessboard_cover.c

相关的题目

与本题相同的题目:

- 《算法竞赛入门经典》[①]第 148 页 8.3.1 节
- 《计算机算法设计与分析 (第 3 版)》[②]第 19 页 2.6 节
- NYOJ 45 棋盘覆盖, http://acm.nyist.net/JudgeOnline/problem.php?pid=45

与本题相似的题目:

- POJ 2495 Incomplete chess boards, http://poj.org/problem?id=2495

8.2 循环赛日程表

描述

有 2^k 个运动员参加循环比赛，需要设计比赛日程表。要求如下:

- 每个选手必须与其他 n-1 个选手各赛一次
- 每个选手一天只能赛一次
- 比赛一共进行 n-1 天

按此要求设计一张比赛日程表，它有 n 行和 n-1 列，第 i 行第 j 列为第 i 个选手第 j 天遇到的对手。

输入

只有一个数 k，$0 < k < 9$，且 k 为自然数。

输出

一张比赛日程表，它有 n 行和 n-1 列（不算第一列，第一列表示选手的编号），第 i 行第 j 列为第 i 个选手第 j 天遇到的对手。相邻的两个整数用空格隔开。

样例输入

```
1
```

[①]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009
[②]王晓东, 计算机算法设计与分析 (第 3 版), 电子工业出版社, 2007

样例输出

```
1 2
2 1
```

分析

根据分而治之的思想，可从其中一半选手 (2^{k-1} 位) 的比赛日程，推导出全体选手的日程，最终细分到只有两位选手的比赛日程。

图 8-2所示是 k=3 时的一个可行解，它是由 4 块拼起来的。左上角是 k=2 时的一组解，左下角是由左上角每个数加 4 得到，而右上角、右下角分别由左下角、左上角复制得到。

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

图 8-3 循环赛日程表问题 k=3 时的解

代码

roundrobin_scheduling.c

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

#define MAXN 512   /* N=2^k, 0<k9 */
short schedule[MAXN][MAXN];

void dc(const int k) {
    int i, j, t;
    int n, n2; /* 当前的 n，即将扩展的 n */

    /* k=1，即两个人时，日程表可以直接写出 */
    n=2;
    schedule[0][0]=1; schedule[0][1]=2;
    schedule[1][0]=2; schedule[1][1]=1;

     /* 迭代处理，依次处理 2^2....2^k 个选手的比赛日程 */
    for(t = 1; t < k; t++, n *= 2) {
        n2 = n * 2;
        //填左下角元素
        for(i = n; i < n2; i++)
            for(j = 0; j < n; j++)
```

```
                schedule[i][j] = schedule[i-n][j] + n;

        //将左下角元素抄到右上角
        for(i = 0; i < n; i++)
            for(j = n; j < n2; j++)
                schedule[i][j] = schedule[i+n][j-n];

        //将左上角元素抄到右下角
        for(i = n; i < n2; i++)
            for(j = n;j < n2; j++)
                schedule[i][j] = schedule[i-n][j-n];
    }
}

/* 另一个版本 */
void dc2(const int k) {
    int i, j, r;
    int n;
    const int N = 1 << k;

    /* 第一列是选手的编号 */
    for(i = 0; i < N; i++) schedule[i][0] = i + 1;
    schedule[0][1] = 1;  /* 当 k=0 时，只有一个人 */

    for (n = 2; n <= N; n *= 2) {  /* 方块大小，2, 4, 8 */
        const int half = n / 2;
        for (r = 0; r < N; r += n) { /* 方块所在行 */
            for (i = r; i <= r + half -1; i++) {  /* 左上角小方块的所有行 */
                for (j = 0; j < half; j++) {  /* 左上角小方块的所有行 */
                    /* 右下角 <-- 左上角 */
                    schedule[i + half][j + half] = schedule[i][j];
                    /* 右上角 <-- 左下角 */
                    schedule[i][j + half] = schedule[i + half][j];
                }
            }
        }
    }
}


int main(){
    int k, N;
    int i,j;

    scanf("%d",&k);
    N = 1 << k;

    dc(k);
    // dc2(k);

    // 输出日程表
    for(i = 0; i < N; i++) {
        for(j = 0; j < N; j++) printf("%d ", schedule[i][j]);
```

```
        printf("\n");
    }
    return 0;
}
```

roundrobin_scheduling.c

相关的题目

与本题相同的题目:

- 《算法竞赛入门经典》[①]第 149 页 8.3.2 节
- 《计算机算法设计与分析 (第 3 版)》[②]第 34 页 2.11 节
- NKOJ 1437 校长杯, http://acm.nankai.edu.cn/p1437.html

与本题相似的题目:

- SPOJ 2826 Round-Robin Scheduling, http://www.spoj.com/problems/RRSCHED/
- UVa OJ 678 Schedule of Taiwan Baseball League, http://t.cn/zHJD9TQ

[①]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009
[②]王晓东, 计算机算法设计与分析 (第 3 版), 电子工业出版社, 2007

第 9 章
贪心法

我们前面见过的一些算法，比如单源最短路径、最小生成树等都属于贪心法 (greedy algorithm)。

如果一个问题具有以下两个要素：

- 最优子结构 (optimal substructure)
- 贪心选择性质 (greedy-choice property)

则可以用贪心法求最优解。

9.1 最优装载

9.2 哈弗曼编码

描述

给定一个英文字符串，使用 0 和 1 对其进行编码，求最优前缀编码，使其所需要的比特数最少。

分析

题目很长，不过就是哈弗曼编码。

代码

poj1521_entropy.cpp

```
// 本题考查哈弗曼编码，但只需要统计哈弗曼编码后的总码长即可，
// 没必要建哈弗曼树得出哈弗曼编码
#include <stdio.h>
#include <string.h>
#include <queue>
#include <functional>

const int LINE_MAX = 256; // 一行最大字符数
```

```
const int MAX_ASCII = 128; // ASCII 码最大值

int main_entropy() {
    char    s[LINE_MAX];
    int     count[MAX_ASCII] = {0}; // count[i] 记录 ASCII 码为 i 的字符的出现次数
    int     sum;
    // 小根堆，队列头为最小元素
    std::priority_queue<int, std::vector<int>, std::greater<int> >    pq;

    while (scanf("%s", s) > 0) {
        sum = 0; // 清零
        const int len = strlen(s);

        if (strcmp(s,"END") == 0) {
            break;
        }

        for (int i = 0; i < len; i++) {
            count[s[i]]++;
        }

        for (int i = 0;i < MAX_ASCII; i++) {
            if (count[i] > 0) {
                pq.push(count[i]);
                count[i] = 0;
            }
        }
        while (pq.size() > 1) {
            const int a = pq.top(); pq.pop();
            const int b = pq.top(); pq.pop();
            sum += a + b;
            pq.push(a + b);
        }
        if (sum == 0) {
            sum = len; // 此时 pq 中只有一个元素
        }

        while (!pq.empty()) { // clear
            pq.pop();
        }
        // 注意精度设置
        printf("%d %d %.1f\n", 8 * len, sum, ((double)8 * len) / sum);
    }
    return 0;
}
```

poj1521_entropy.cpp

相关的题目

与本题相同的题目：

- POJ 1521 Entropy, http://poj.org/problem?id=1521

与本题相似的题目：

- POJ 3253 Fence Repair, http://poj.org/problem?id=3253
- 《算法竞赛入门经典》[①] 第 155 页例题 8-5
- 《Introduction to Algorithms》[②] 第 16.3 节
- 《算法设计与分析 (第 3 版)》[③] 第 109 页 4.4 节

9.3 部分背包问题

[①]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009
[②]CLRS,Introduction to Algorithms(3rd Edition), 2009
[③]王晓东, 计算机算法设计与分析 (第 3 版), 2007

第 10 章 动态规划

如果一个问题具有以下两个要素：

- 最优子结构 (optimal substructure)
- 重叠子问题 (overlap subproblem)

则可以用动态规划求最优解。

动态规划分为 4 个步骤：

- 描述最优解的结构。即抽象出一个状态来表示最优解。
- 递归的定义最优解的值。找出状态转移方程，然后递归的定义
- 计算最优解的值。典型的做法是自底向上，当然也可以自顶向下。
- 根据计算过程中得到的信息，构造出最优解。如果我们只需要最优解的值，不需要最优解本身，则可以忽略第 4 步。当执行第 4 步时，我们需要在第 3 步的过程中维护一些额外的信息，以便我们能方便的构造出最优解。

在第 1 步中，我们需要抽象出一个“状态”，在第 2 步中，我们要找出“状态转移方程”，然后才能递归的定义最优解的值。第 3 步和第 4 步就是写代码实现了。

写代码实现时有两种方式，“递归 (recursive)+ 自顶向下 (top-down)+ 表格 (memoization)”和“自底向上 (bottom-up)+ 表格”。自顶向下也称为记忆化搜索，自底向上也称为递推（不是递归）。

动规用表格将各个子问题的最优解存起来，避免重复计算，是一种空间换时间。

动规与贪心的相同点：最优子结构。

不同点：1、动规的子问题是重叠的，而贪心的子问题是不重叠的 (disjoint subproblems)；2、动规不具有贪心选择性质；3、贪心的前进路线是一条线，而动规是一个 DAG。

分治和贪心的相同点：disjoint subproblems。

10.1 最长公共子序列

描述

一个序列的子序列 (subsequence) 是指在该序列中删去若干（可以为 0 个）元素后得到的序列。准确的说，若给定序列 $X = (x_1, x_2, \cdots, x_m)$，则另一个序列 $Z = (z_1, z_2, \cdots, z_k)$，是 X 的子序列是指存在一个严格递增下标序列 $(i_1, i_2, \cdots, i_k)$ 使得对于所有 $j = 1, 2, \cdots, k$ 有 $z_j = x_{i_j}$。例如，序列 $Z = (B, C, D, B)$ 是序列 $X = (A, B, C, B, D, A, B)$ 的子序列，相应的递增下标序列为 $(1, 2, 4, 6)$。

给定两个序列 X 和 Y，求 X 和 Y 的最长公共子序列 (longest common subsequence)。

输入

输入包括多组测试数据，每组数据占一行，包含两个字符串（字符串长度不超过 200），代表两个序列。两个字符串之间由若干个空格隔开。

输出

对每组测试数据，输出最大公共子序列的长度，每组一行。

样例输入

```
abcfbc abfcab
programming contest
abcd mnp
```

样例输出

```
4
2
0
```

分析

最长公共子序列问题具有最优子结构性质。设序列 $X = (x_1, x_2, \cdots, x_m)$ 和 $Y = (y_1, y_2, \cdots, y_n)$ 的最长公共子序列为 $Z = (z_1, z_2, \cdots, z_k)$，则

- 若 $x_m = y_n$，则 $z_k = x_m = y_n$，且 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的最长公共子序列。
- 若 $x_m \neq y_n$ 且 $z_k \neq x_m$，则 Z 是 X_{m-1} 和 Y 的最长公共子序列。
- 若 $x_m \neq y_n$ 且 $z_k \neq y_n$，则 Z 是 X 和 Y_{n-1} 的最长公共子序列。

其中，$X_{m-1}=(x_1,x_2,\cdots,x_{m-1}), Y_{n-1}=(y_1,y_2,\cdots,y_{n-1}), Z_{k-1}=(z_1,z_2,\cdots,z_{k-1})$。

设状态为 d[i][j]，表示序列 X_i 和 Y_j 的最长公共子序列的长度。由最优子结构可得状态转移方程如下：

$$d[i][j]=\begin{cases}0 & i=0,j=0\\ d[i-1][j-1]+1 & i,j>0;x_i=y_i\\ \max\left\{d[i][j-1],d[i-1][j]\right\} & i,j>0;x_i\neq y_i\end{cases}$$

如果要打印出最长公共子序列，需要另设一个数组 p，p[i][j] 记录 d[i][j] 是由哪个子问题得到的。

代码

lcs.c

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

#define MAX 201  /* 字符串最大长度为 200 */

int d[MAX][MAX]; /* d[i][j] 表示序列 Xi 和 Yj 的最长公共子序列的长度 */
char x[MAX], y[MAX];  /* 字符串末尾有个'0' */

void lcs(const char *x, const int m, const char *y, const int n) {
    int i, j;

    for (i = 0; i <= m; i++) d[i][0] = 0;  /* 边界初始化 */
    for (j = 0; j <= n; j++) d[0][j] = 0;

    for (i = 1; i <= m; i++) {
        for (j = 1; j <= n; j++) {
            if (x[i-1] == y[j-1]) {
                d[i][j] = d[i-1][j-1] + 1;
            } else {
                d[i][j] = d[i-1][j] > d[i][j-1] ? d[i-1][j] : d[i][j-1];
            }
        }
    }
}

void lcs_extend(const char *x, const int m, const char *y, const int n);
void lcs_print(const char *x, const int m, const char *y, const int n);

int main() {
    /* while (scanf ("%s%s", a, b)) { /* TLE */
    /* while (scanf ("%s%s", a, b) == 2) {  /* AC */
    while (scanf ("%s%s", x, y) != EOF) { /* AC */
        const int lx = strlen(x);
        const int ly = strlen(y);
```

```
        lcs(x, lx, y, ly);
        printf ("%d\n", d[lx][ly]);
        /*
        lcs_extend(x, lx, y, ly);
        lcs_print(x, lx, y, ly);
        printf("\n"); */
    }
    return 0;
}

int p[MAX][MAX]; /* p[i][j] 记录 d[i][j] 是由哪个子问题得到的 */

void lcs_extend(const char *x, const int m, const char *y, const int n) {
    int i, j;

    memset(p, 0, sizeof(p));
    for (i = 0; i <= m; i++) d[i][0] = 0;  /* 边界初始化 */
    for (j = 0; j <= n; j++) d[0][j] = 0;

    for (i = 1; i <= m; i++) {
        for (j = 1; j <= n; j++) {
            if (x[i-1] == y[j-1]) {
                d[i][j] = d[i-1][j-1] + 1;
                p[i][j] = 1;
            } else {
                if (d[i-1][j] >= d[i][j-1]) {
                    d[i][j] = d[i-1][j];
                    p[i][j] = 2;
                } else {
                    d[i][j] = d[i][j-1];
                    p[i][j] = 3;
                }
            }
        }
    }
}

void lcs_print(const char *x, const int m, const char *y, const int n) {
    if (m == 0 || n == 0) return;

    if (p[m][n] == 1) {
        lcs_print(x, m - 1, y, n - 1);
        printf("%c", x[m - 1]);
    } else if (p[m][n] == 2) {
        lcs_print(x, m - 1, y, n);
    } else {
        lcs_print(x, m, y, n - 1);
    }
}
```

lcs.c

相关的题目

与本题相同的题目：

- 《计算机算法设计与分析 (第 3 版)》[①]第 56 页 3.3 节
- POJ 1458 Common Subsequence, http://poj.org/problem?id=1458
- HDOJ 1159 Common Subsequence, http://acm.hdu.edu.cn/showproblem.php?pid=1159
- 《程序设计导引及在线实践》[②]第 203 页 10.5 节
- 百练 2806 公共子序列, http://poj.grids.cn/practice/2806/

与本题相似的题目：

- HDOJ 1080 Human Gene Functions, http://acm.hdu.edu.cn/showproblem.php?pid=1080
- HDOJ 1503 Advanced Fruits, http://acm.hdu.edu.cn/showproblem.php?pid=1503

10.2 最大连续子序列和

描述

给定一个整数序列 $S_1, S_2, \cdot, S_n (1 \leq n \leq 1,000,000, -32768 \leq S_i \leq 32768)$，定义函数 $sum(i, j) = S_i + \cdot + S_j (1 \leq i \leq j \leq n)$。

求最大的 $sum(i, j)$，即**最大连续子序列和** (maximum continuous subsequence sum)。

输入

每个测试用例由正整数 n 开头，接着是 n 个整数。

输出

每行输出一个最大和。

样例输入

```
3 1 2 3
6 -1 4 -2 3 -2 3
```

样例输出

```
6
6
```

[①] 王晓东, 计算机算法设计与分析 (第 3 版), 电子工业出版社, 2007

[②] 李文新, 程序设计导引及在线实践, 清华大学出版社, 2007

分析

设状态为 d[j]，表示以 S[j] 结尾的最大连续子序列和，则状态转移方程如下:

$$
\begin{aligned}
d[j] &= \max\left\{d[j-1]+S[j], S[j]\right\}, \text{其中} 1 \leq j \leq n \\
target &= \max\left\{d[j]\right\}, \text{其中} 1 \leq j \leq n
\end{aligned}
$$

解释如下:

- 情况一，S[j] 不独立，与前面的某些数组成一个连续子序列，则最大连续子序列和为 $d[j-1]+S[j]$。
- 情况二，S[j] 独立划分成为一段，即连续子序列仅包含一个数 S[j]，则最大连续子序列和为 $S[j]$。

代码

mcss.c

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

inline static int max(int a , int b) {
   return (a > b) ? a : b ;
}

inline static int min(int a , int b) {
   return (a < b) ? a : b ;
}

/**
 * @brief 最大连续子序列和，思路四
 * @param[in] S 数列
 * @param[in] n 数组的长度
 * @return 最大连续子序列和
 */
int mcss(int S[], int n) {
    int i, maxs, mins;
    int *sum = (int*) malloc((n + 1) * sizeof(int));  // 前 n 项和
    sum[0] = 0;
    maxs = 0, mins = 0;
    for (i = 1; i <= n; i++) {
        sum[i] = sum[i - 1] + S[i - 1];
        maxs = max(maxs, sum[i]);
        mins = min(mins, sum[i]);
    }
    free(sum);
    return maxs - mins;
}

/**
```

```
 * @brief 最大连续子序列和，动规
 * @param[in] S 数列
 * @param[in] n 数组的长度
 * @return 最大连续子序列和
 */
int mcss_dp(int S[], int n) {
    int i, result;
    int *d = (int*) malloc(n * sizeof(int));
    d[0] = S[0];
    result = d[0];
    for (i = 1; i < n; i++) {
        d[i] = max(S[i], d[i - 1] + S[i]);
        if (result < d[i])
            result = d[i];
    }
    free(d);
    return result;
}

int main() {
    int n, *S;
    while (scanf("%d", &n) > 0) {
        int i;
        S = (int*) malloc(n * sizeof(int));
        for (i = 0; i < n; i++)
            scanf("%d", &S[i]);

        printf("%d\n", mcss_dp(S, n));
        free(S);
    }
    return 0;
}
```
mcss.c

其他解法

- 思路一：直接在 i 到 j 之间暴力枚举，复杂度是 $O(n^3)$
- 思路二：处理后枚举，连续子序列的和等于两个前缀和之差，复杂度 $O(n^2)$。
- 思路三：分治法，把序列分为两段，分别求最大连续子序列和，然后归并，复杂度 $O(n \log n)$
- 思路四：把 $O(n^2)$ 的代码稍作处理，得到 $O(n)$ 的算法
- 思路五：M=1 的最大 M 子段和，见第 §10.3 节

感兴趣的读者请参考这篇博客，http://www.cnblogs.com/gj-Acit/archive/2013/02/12/2910332.html

相关题目

与本题相同的题目:

- TODO

与本题相似的题目:

- HDOJ 1024 Max Sum Plus Plus, http://acm.hdu.edu.cn/showproblem.php?pid=1024

10.3 最大 M 子段和

描述

给定一个整数序列 $S_1, S_2, \cdot, S_n (1 \leq n \leq 1,000,000, -32768 \leq S_i \leq 32768)$，定义函数 $sum(i,j) = S_i + ... + S_j (1 \leq i \leq j \leq n)$。

现给定一个正整数 m，找出 m 对 i 和 j，使得 $sum(i_1,j_1) + sum(i_2,j_2) + ... + sum(i_m,j_m)$ 最大。这就是**最大 M 子段和** (maximum m segments sum)。

输入

每个测试用例由两个正整数 m 和 n 开头，接着是 n 个整数。

输出

每行输出一个最大和。

样例输入

```
1 3 1 2 3
2 6 -1 4 -2 3 -2 3
```

样例输出

```
6
8
```

分析

设状态为 d[i,j]，表示前 j 项分为 i 段的最大和，且第 i 段必须包含 S[j]，则状态转移方程如下:

$$\begin{aligned} d[i,j] &= \max\left\{d[i,j-1]+S[j], \max\left\{d[i-1,t]+S[j]\right\}\right\}, \text{其中} i \leq j \leq n, i-1 \leq t < j \\ target &= \max\left\{d[m,j]\right\}, \text{其中} m \leq j \leq n \end{aligned}$$

分为两种情况：

- 情况一，S[j] 包含在第 i 段之中，$d[i, j-1] + S[j]$。
- 情况二，S[j] 独立划分成为一段，$\max\{d[i-1, t] + S[j]\}$。

观察上述两种情况可知 d[i,j] 的值只和 d[i,j-1] 和 d[i-1,t] 这两个值相关，因此不需要二维数组，可以用滚动数组，只需要两个一维数组，用 d[j] 表示现阶段的最大值，即 $d[i, j-1] + S[j]$，用 prev[j] 表示上一阶段的最大值，即 $\max\{d[i-1, t] + S[j]\}$。

代码

mmss.c

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

/**
 * @brief 最大 m 段子序列和
 * @param[in] S 数组
 * @param[in] n 数组长度
 * @param[in] m m 段
 * @return 最大 m 段子序列和
 */
int mmss(int S[], int n, int m) {
    int max_sum, i, j;

    /* d[i] 表示现阶段最大值，prev[i] 表示上阶段最大值 */
    /* d[0], prev[0] 未使用 */
    int *d = (int*) calloc(n + 1, sizeof(int));
    int *prev = (int*) calloc(n + 1, sizeof(int));
    S--;  // 因为 j 是从 1 开始，而 S 从 0 开始，这里要减一

    for (i = 1; i <= m; ++i) {
        max_sum = INT_MIN;
        for (j = i; j <= n; ++j) {
            // 状态转移方程
            if (d[j - 1] < prev[j - 1])
                d[j] = prev[j - 1] + S[j];
            else
                d[j] = d[j - 1] + S[j];

            prev[j - 1] = max_sum;  // 存放上阶段最大值
            if (max_sum < d[j])
                max_sum = d[j];  // 更新 max_sum
        }
        prev[j - 1] = max_sum;
    }

    free(d);
    free(prev);
    return max_sum;
```

```
}

int main() {
    int n, m, i, *S;
    while (scanf("%d%d", &m, &n) == 2) {
        S = (int*) malloc(sizeof(int) * n);
        for (i = 0; i < n; ++i)
            scanf("%d", &S[i]);
        printf("%d\n", mmss(S, n, m));
        free(S);
    }
    return 0;
}
```
mmss.c

相关题目

与本题相同的题目:

- HDOJ 1024 Max Sum Plus Plus, http://acm.hdu.edu.cn/showproblem.php?pid=1024

与本题相似的题目:

- LeetCode Best Time to Buy and Sell Stock III, http://leetcode.com/onlinejudge#question_123, 参考代码 https://gist.github.com/soulmachine/5906637

10.4 DAG 上的动态规划

10.4.1 数字三角形

描述

有一个由非负整数组成的三角形，第一行只有一个数，除了最下一行之外每个数的左下角和右下角各有一个数，如图 10-1所示。

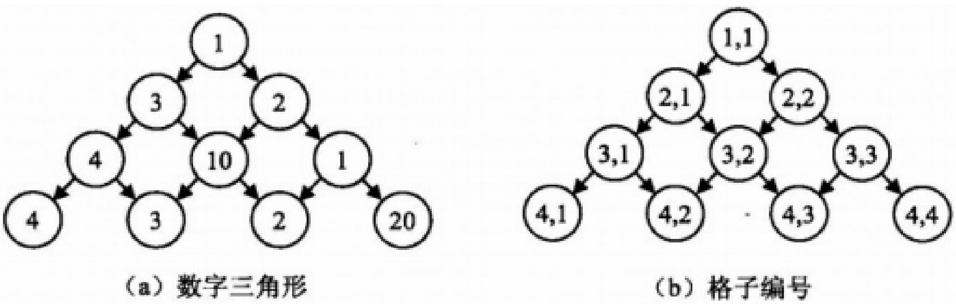


图 10-1 数字三角形问题

从第一行的数开始，每次可以往左下或右下走一格，直到走到最下行，把沿途经过的数全部加起来。如何走才能使得这个和最大？

输入

第一行是一个整数 $N(1 \leq N \leq 100)$，给出三角形的行数。接下来的 N 行给出数字三角形。三角形中的数全部是整数，范围在 0 到 100 之间。

输出

输出最大的和。

样例输入

```
5
7
3 8
8 1 0
2 7 4 4
4 5 2 6 5
```

样例输出

```
30
```

分析

这是一个动态决策问题，在每层有两种选择，左下或右下，因此一个 n 层的数字三角形有 2^n 条路线。

可以用回溯法，用回溯法求出所有可能的路线，就可以从中选出最优路线。但是由于有 2^n 条路线，回溯法很慢。

本题可以用动态规划来求解 (具有最有子结构和重叠子问题两个要素，后面会看到)。把当前位置 (i,j) 看成一个状态，然后定义状态 d[i][j] 为从位置 (i,j) 出发时能得到的最大和 (包括格子 (i,j) 本身的值 a[i][j])。在这个状态定义下，原问题的解是 d[0][0]。

下面来看看不同状态之间是怎样转移的。从位置 (i,j) 出发有两种决策，如果往左走，则走到 (i+1,j) 后需要求“从 (i+1,j) 出发后能得到的最大和”这一子问题，即 d[i+1][j]，类似地，往右走之后需要求 d[i+1][j+1]。应该选择 d[i+1][j] 和 d[i+1][j+1] 中较大的一个，因此可以得到如下的状态转移方程：

$$d[i][j] = a[i][j] + \max\{d[i+1][j], d[i+1][j+1]\}$$

代码

版本 1，自顶向下。

numbers_triangle1.c

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

#define MAXN 100

int n, a[MAXN][MAXN], d[MAXN][MAXN];

int max(const int x, const int y) {
    return x > y ? x : y;
}

/**
 * @brief 求从位置 (i,j) 出发时能得到的最大和
 * @param[in] i 行
 * @param[in] j 列
 * @return 最大和
 */
int dp(const int i, const int j) {
    if(d[i][j] >= 0) {
        return d[i][j];
    } else {
        return d[i][j] = a[i][j] + (i == n-1 ? 0 : max(dp(i+1, j+1), dp(i+1, j)));
    }
}

int main() {
    int i, j;
    memset(d, -1, sizeof(d));

    scanf("%d", &n);
    for(i = 0; i < n; i++)
      for (j = 0; j <= i; j++) scanf("%d", &a[i][j]);

    printf("%d\n", dp(0, 0));
    return 0;
}
```

numbers_triangle1.c

版本 2，自底向上。

numbers_triangle2.c

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

#define MAXN 100

int n, a[MAXN][MAXN], d[MAXN][MAXN];

int max(const int x, const int y) {
```

```c
    return x > y ? x : y;
}

/**
 * @brief 自底向上计算所有子问题的最优解
 * @return 无
 */
void dp() {
    int i, j;
    for (i = 0; i < n; ++i) {
        d[n-1][i] = a[n-1][i];
    }
    for (i = n-2; i >= 0; --i)
      for (j = 0; j <= i; ++j)
        d[i][j] = a[i][j] + max(d[i+1][j], d[i+1][j+1]);
}

int main() {
    int i, j;
    memset(d, -1, sizeof(d));

    scanf("%d", &n);
    for(i = 0; i < n; i++)
      for (j = 0; j <= i; j++)
          scanf("%d", &a[i][j]);

    dp();

    printf("%d\n", d[0][0]);
    return 0;
}
```

numbers_triangle2.c

相关的题目

与本题相同的题目：

- 《算法竞赛入门经典》[1]第 159 页 9.1.1 节
- POJ 1163 The Triangle, http://poj.org/problem?id=1163
- 百练 2760 数字三角形, http://poj.grids.cn/practice/2760/

与本题相似的题目：

- TODO

[1]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

10.4.2 嵌套矩形

描述

有 n 个矩形，每个矩形可以用 a,b 来描述，表示长和宽。矩形 $X(a,b)$ 可以嵌套在矩形 $Y(c,d)$ 中当且仅当 $a<c,b<d$ 或者 $b<c,a<d$（相当于旋转 X90 度)。例如（1,5）可以嵌套在（6,2）内，但不能嵌套在（3,4）中。你的任务是选出尽可能多的矩形排成一行，使得除最后一个外，每一个矩形都可以嵌套在下一个矩形内。

输入

第一行是一个正整数 $N(0<N<10)$，表示测试数据组数，每组测试数据的第一行是一个正正数 n，表示该组测试数据中含有矩形的个数 $(n<=1000)$ 随后的 n 行，每行有两个数 $a,b(0<a,b<100)$，表示矩形的长和宽

输出

每组测试数据都输出一个数，表示最多符合条件的矩形数目，每组输出占一行

样例输入

```
1
10
1 2
2 4
5 8
6 10
7 9
3 1
5 8
12 10
9 7
2 2
```

样例输出

```
5
```

分析

本题实质上是求 DAG 中不固定起点的最长路径。

设 d[i] 表示从结点 i 出发的最长长度，状态转移方程如下:

$$d[i]=\max\left\{d[j]+1|(i,j)\in E\right\}$$

其中，E 为边的集合。最终答案是 d[i] 中的最大值。

代码

embedded_rectangles.c

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

#define MAXN 1000 // 矩形最大个数

int n; // 矩形个数
int G[MAXN][MAXN]; // 矩形包含关系
int d[MAXN]; // 表格

/**
 * @brief 动规，自顶向下.
 * @param[in] i 起点
 * @return 以 i 为起点，能达到的最长路径
 */
int dp(const int i) {
    int j;
    int *ans= &d[i];
    if(*ans > 0) return *ans;

    *ans = 1;
    for(j = 0; j < n; j++) if(G[i][j]) {
        const int next = dp(j) + 1;
        if(*ans < next) *ans = next;
    }
    return *ans;
}

/**
 * @brief 按字典序打印路径.
 *
 * 如果多个 d[i] 相等，选择最小的 i。
 *
 * @param[in] i 起点
 * @return 无
 */
void print_path(const int i) {
    int j;
    printf("%d ", i);
    for(j = 0; j < n; j++) if(G[i][j] && d[i] == d[j] + 1) {
        print_path(j);
        break;
    }
}

int main() {
    int N, i, j;
    int max, maxi;
```

```
    int a[MAXN],b[MAXN];

    scanf("%d", &N);
    while(N--) {
        memset(G, 0, sizeof(G));
        memset(d, 0, sizeof(d));

        scanf("%d", &n);
        for(i = 0; i < n; i++) scanf("%d%d", &a[i], &b[i]);

        for(i = 0; i < n; i++)
            for(j = 0; j < n; j++)
                if((a[i] > a[j] && b[i] > b[j]) ||
                    (a[i] > b[j] && b[i] > a[j])) G[i][j] = 1;

        max = 0;
        maxi = -1;
        for(i = 0; i < n; i++) if(dp(i) > max) {
            max = dp(i);
            maxi = i;
        }
        printf("%d\n", max);
        // print_path(maxi);
    }
    return 0;
}
```

embedded_rectangles.c

相关的题目

与本题相同的题目:

- 《算法竞赛入门经典》[①]第 161 页 9.2.1 节
- NYOJ 16 嵌套矩形, http://acm.nyist.net/JudgeOnline/problem.php?pid=16

与本题相似的题目:

- TODO

10.4.3 硬币问题

描述

有 n 种硬币，面值为别为 $v_1, v_2, v_3, \cdots, v_n$，每种都有无限多。给定非负整数 S，可以选取多少个硬币，使得面值和恰好为 S？输出硬币数目的最小值和最大值。$1 \leq n \leq 100, 1 \leq S \leq 10000, 1 \leq v_i \leq S$。

[①]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

输入

第 1 行 n，第 2 行 S，第 3 到 $n+2$ 行为 n 种不同的面值。

输出

第 1 行为最小值，第 2 行为最大值。

样例输入

```
3
6
1
2
3
```

样例输出

```
2
6
```

分析

本题实质上是求 DAG 中固定终点的最长路径和最短路径。

把每种面值看作一个点，表示“还需要凑足的面值”，则初始状态为 S，目标状态为 0。若当前状态为 i，每使用一个硬币 j，状态便转移到 $i-v_j$。

设状态为 d[i]，表示从节点 i 出发的最长路径长度，则原问题的解是 d[S]。状态转移方程如下:

$$d[i]=\max\left\{d[j]+1,(i,j)\in E\right\}$$

本题还可以看作是完全背包问题（见 §10.5.2节）：背包容量为 S，背包必须要装满，物品即硬币，每个硬币的费用为面值 v_i，价值均为 1。求背包中物品的最小价值和最大价值。

代码

版本 1，自顶向下。

coin_change.c

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

#define MAXN 100
#define MAXV 10000

/** 硬币面值的种类. */
```

```
int n;
/** 要找零的数目. */
int S;
/** 硬币的各种面值. */
int v[MAXN];
/** min[i] 表示面值之和为 i 的最短路径的长度, max 则是最长. */
int min[MAXV + 1], max[MAXV + 1];

/**
 * @brief 最短路径.
 * @param[in] s 面值
 * @return 最短路径长度
 */
int dp1(const int s) { // 最小值
    int i;
    int *ans = &min[s];
    if(*ans != -1) return *ans;
    *ans = 1<<30;
    for(i = 0; i < n; ++i) if(v[i] <= s) {
        const int tmp = dp1(s-v[i])+1;
        *ans = *ans < tmp ? *ans : tmp;
    }
    return *ans;
}

int visited[MAXV + 1];
/**
 * @brief 最长路径.
 * @param[in] s 面值
 * @return 最长路径长度
 */
int dp2(const int s) { //最大值
    int i;
    int *ans = &max[s];

    if(visited[s]) return max[s];
    visited[s] = 1;

    *ans = -1<<30;
    for(i = 0; i < n; ++i) if(v[i] <= s) {
        const int tmp = dp2(s-v[i])+1;
        *ans = *ans > tmp ? *ans : tmp;
    }
    return *ans;
}

void print_path(const int* d, const int s);

int main() {
    int i;
    scanf("%d%d", &n, &S);
    for(i = 0; i < n; ++i) scanf("%d", &v[i]);
```

```
    memset(min, -1, sizeof(min));
    min[0] = 0;
    printf("%d\n", dp1(S));
    // print_path(min, S);

    memset(max, -1, sizeof(max));
    memset(visited, 0, sizeof(visited));
    max[0] = 0; visited[0] = 1;
    printf("%d\n", dp2(S));
    // print_path(max, S);

    return 0;
}

/**
 * @brief 打印路径.
 * @param[in] d 上面的 min 或 min
 * @param[in] s 面值之和
 * @return 无
 */
void print_path(const int* d, const int s) {//打印的是边
    int i;
    for(i = 0; i < n; ++i) if(v[i] <= s && d[s-v[i]] + 1 == d[s]) {
        printf("%d ",i);
        print_path(d, s-v[i]);
        break;
    }
    printf("\n");
}
```

coin_change.c

版本 2，自底向上。

coin_change2.c

```
#include<stdio.h>

#define MAXN 100
#define MAXV 10000

int n, S, v[MAXN], min[MAXV + 1], max[MAXV + 1];
int min_path[MAXV], max_path[MAXV];

void dp() {
    int i, j;

    min[0] = max[0] = 0;
    for(i = 1; i <= S; ++i) {
        min[i] = MAXV;
        max[i] = -MAXV;
    }

    for(i = 1; i <= S; ++i) {
        for(j = 0; j < n; ++j) if(v[j] <= i) {
            if(min[i-v[j]] + 1 < min[i]) {
```

```
                min[i] = min[i-v[j]] + 1;
                min_path[i] = j;
            }
            if(max[i-v[j]] + 1 > max[i]) {
                max[i] = max[i-v[j]] + 1;
                max_path[i] = j;
            }
        }
    }
}

void print_path(const int *d, int s);

int main() {
    int i;
    scanf("%d%d", &n, &S);
    for(i = 0; i < n; ++i) scanf("%d", &v[i]);

    dp();
    printf("%d\n", min[S]);
    // print_path(min_path, S);
    printf("%d\n", max[S]);
    // print_path(max_path, S);
    return 0;
}

/**
 * @brief 打印路径.
 * @param[in] d 上面的 min_path 或 min_path
 * @param[in] s 面值之和
 * @return 无
 */
void print_path(const int *d, int s) {
    while(s) {
        printf("%d ", d[S]);
        S -= v[d[s]];
    }
    printf("\n");
}
```

coin_change2.c

版本 3，当作完全背包问题。

coin_change3.c

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

#define MAXN 100
#define MAXW 10000
 /* 无效值, 不要用 0x7FFFFFFF, 执行加运算后会变成负数 */
const int INF = 0x0FFFFFFF;

int N, W;
int w[MAXN], v[MAXN];
```

```
int min[MAXW + 1], max[MAXW + 1]; /* 滚动数组 */

int min_path[MAXW + 1], max_path[MAXW + 1];
void print_path(const int *d, int s);

/**
 * @brief 完全背包问题中，处理单个物品.
 * @param[in] d 滚动数组
 * @param[in] i 该物品的下标
 * @return 无
 */
void unbounded_knapsack(int min[], int max[], const int i) {
    int j;
    for(j = w[i]; j <= W; ++j) {
        if(min[j - w[i]] + v[i] < min[j]) {
            min[j] = min[j - w[i]] + v[i];
            // min_path[j] = i;
        }
        if(max[j - w[i]] + v[i] > max[j]) {
            max[j] = max[j - w[i]] + v[i];
            // max_path[j] = i;
        }
    }
}

void dp() {
    int i, j;
    min[0] = 0;
    max[0] = 0;
    for(j = 1; j <= W; ++j) { /* 背包要装满 */
        min[j] = INF;
        max[j] = -INF;
    }

    for(i = 0; i < N; ++i) unbounded_knapsack(min, max, i);
}

int main() {
    int i;
    for (i = 0; i < MAXN; ++i) v[i] = 1;
    scanf("%d%d", &N, &W);
    for(i = 0; i < N; ++i) scanf("%d", &w[i]);

    dp();
    printf("%d\n", min[W]);
    // print_path(min_path, W);
    printf("%d\n", max[W]);
    // print_path(max_path, W);
    return 0;
}

/**
```

```
 * @brief 打印路径.
 * @param[in] d 上面的 min_path 或 min_path
 * @param[in] j 面值之和
 * @return 无
 */
void print_path(const int *d, int j) {
    while(j) {
        printf("%d ", d[j]);
        j -= w[d[j]];
    }
    printf("\n");
}
```

coin_change3.c

相关的题目

与本题相同的题目:

- 《算法竞赛入门经典》[1]第 162 页例题 9-3
- tyvj 1214 硬币问题, http://www.tyvj.cn/problem_show.aspx?id=1214

与本题相似的题目:

- TODO

10.4.4 最长上升子序列

描述

当一个序列严格递增时，我们称这个序列是上升的。对于一个给定的序列 $(a_1, a_2, ..., a_N)$，我们可以得到一些上升的子序列 $(a_{i1}, a_{i2}, ..., a_{iK})$，这里 $1 \leq i1 < i2 < ... < iK \leq N$。例如，对于序列 (1, 7, 3, 5, 9, 4, 8)，有它的一些上升子序列，如 (1, 7), (3, 4, 8) 等等，这些子序列中最长的长度是 4，比如子序列 (1, 3, 5, 8)。

对于给定的序列，求**最长上升子序列** (longest increasing subsequence) 的长度。

输入

第一行是序列的长度 $N(1 \leq N \leq 1000)$。第二行给出序列中的 N 个整数，这些整数的取值范围都在 0 到 10000。

输出

最长上升子序列的长度。

[1]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

样例输入

```
7
1 7 3 5 9 4 8
```

样例输出

```
4
```

分析

设状态为 d[j]，表示以 a_j 为终点的最长上升子序列的长度。状态转移方程如下;

$$d[j] = \begin{cases} 1 & j = 1 \\ \max\{d[i]\} + 1 & 1 < i < j, a_i < a_j \end{cases}$$

代码

lis.c

```c
#include<stdio.h>
#define MAXN 1001 // a[0] 未用

int N;
int a[MAXN];
int d[MAXN];

void dp() {
    int i, j;
    d[1] = 1;

    for (j = 2; j <= N; j++) { // 每次求以 aj 为终点的最长上升子序列的长度
        int max = 0;  // 记录 aj 左边的上升子序列的最大长度
        for (i = 1; i < j; i++)  if (a[i] <a[j] && max < d[i]) max = d[i];
        d[j] = max + 1;
    }
}

int main() {
    int i, max;
    scanf("%d",&N);
    for (i = 1; i <= N;i++) scanf("%d",&a[i]);

    dp();

    max = 0;
    for(i = 1; i <= N;i++) if (d[i] > max) max = d[i];
    printf("%d\n",max);
    return 0;
}
```

lis.c

相关的题目

与本题相同的题目：

- 《程序设计导引及在线实践》[1]第 198 页例题 10.3
- 百练 2757 最长上升子序列, http://poj.grids.cn/practice/2757/
- POJ 2533 Longest Ordered Subsequence, http://poj.org/problem?id=2533

与本题相似的题目：

- TODO

10.5 背包问题

背包问题 (Knapsack problem[2]) 有很多种版本，常见的是以下三种：

- 0-1 背包问题 (0-1 knapsack problem)：每种物品只有一个
- 完全背包问题 (UKP, unbounded knapsack problem)：每种物品都有无限个可用
- 多重背包问题 (BKP, bounded knapsack problem)：第 i 种物品有 c[i] 个可用

其他版本的背包问题请参考“背包问题九讲”，https://github.com/tianyicui/pack

背包问题是一种“多阶段决策问题”。

10.5.1 0-1 背包问题

描述

有 N 种物品，第 i 种物品的重量为 w_i，价值为 v_i，每种物品只有一个。背包能承受的重量为 W。将哪些物品装入背包可使这些物品的总重量不超过背包容量，且价值总和最大？

输入

第 1 行包含一个整数 T，表示有 T 组测试用例。每组测试用例有 3 行，第 1 行包含两个整数 $N, W (N \leq 1000, W \leq 1000)$ 分别表示物品的种数和背包的容量，第 2 行包含 N 个整数表示每种物品的价值，第 3 行包含 N 个整数表示每种物品的重量。

输出

每行一个整数，表示价值总和的最大值。

[1]李文新, 程序设计导引及在线实践, 清华大学出版社,2007

[2]Knapsack problem, http://en.wikipedia.org/wiki/Knapsack_problem

样例输入

```
1
5 10
1 2 3 4 5
5 4 3 2 1
```

样例输出

```
14
```

分析

由于每种物品仅有一个，可以选择装或者不装。

定义状态 f[i][j]，表示“把前 i 个物品装进容量为 j 的背包可以获得的最大价值”，则其状态转移方程便是：

$$f[i][j] = \max\{f[i-1][j], f[i-1][j-w[i]+v[i]\}$$

这个方程理解如下，把前 i 个物品装进容量为 j 的背包时，有两种情况：

- 第 i 个不装进去，这时所得价值为：$f[i-1][j]$
- 第 i 个装进去，这时所得价值为：$f[i-1][j-w[i]]+v[i]$

动规过程的伪代码如下：

```
f[0..N][0..W] = 0
for i=1..N
    for j=0..W
        f[i][j]=max{f[i-1][j],f[i-1][j-w[i]]+v[i]};
```

内循环从右向左也可以：

```
f[0..N][0..W] = 0
for i=1..N
    for j=W..0
        f[i][j]=max{f[i-1][j],f[i-1][j-w[i]]+v[i]};
```

内循环从右向左时，可以把二维数组优化成一维数组。伪代码如下：

```
for i=1..N
    for j=W..0
        d[j]=max{d[j],d[j-w[i]]+v[i]};
```

为什么呢？举个例子，测试数据如下：

```
1
3 10
4 5 6
3 4 5
```

f 是从上到下、从右到左计算的，如图 10-2所示。

最大容量M	物品个数N							j=0-m						
10	3		C	0	1	2	3	4	5	6	7	8	9	10
物品大小w	物品价值p	编号	0	0	0	0	0	0	0	0	0	0	0	0
3	4	i= 1	1	0	0	0	4	4	4	4	4	4	4	4
4	5	1-n 2	2	0	0	0	4	5	5	5	9	9	9	9
5	6	3	3	0	0	0	4	5	6	6	9	10	11	11

图 10-2　0-1 背包问题的计算顺序

当内循环是逆序时，且动规是用自底向上方式实现时，就可以保证同一行可以从右向左更新。

设一维数组为 d（又称为滚动数组[1]），在更新 d[j] 之前，d[j] 里保存的 f[i-1][j]，更新之后，d[j] 里保存的是 f[i][j]。

事实上，使用一维数组解 0-1 背包问题的程序在后面会被多次用到，所以这里抽象出一个处理单个物品的函数，以后的代码中直接调用不加说明。

```
def ZeroOneKnapsack(d[], i)
    for j = W..w[i]
        d[j] = max(d[j], d[j-w[i]] + v[i])
```

有了这个函数以后，0-1 背包问题的伪代码就可以这样写：

```
d[0..W] = 0
for i = 1..N
    ZeroOneKnapsack(d[], i)
```

代码

版本 1，自底向上。

01knapsack.c

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

#define MAXN 1000
#define MAXW 1000

int N, W;
int w[MAXN+1], v[MAXN+1]; /* 0 没有用 */

int f[MAXN + 1][MAXW + 1];

void dp() {
    int i, j;
    memset(f, 0, sizeof(f)); /* 背包不一定要装满 */
    for(i = 1; i <= N; ++i) {
        /* for(j = W; j >= 0; --j) { /* 也可以 */
        for(j = 0; j <= W; ++j) {
            f[i][j] = f[i-1][j];
```

[1] 刘汝佳, 算法竞赛入门经典，清华大学出版社，2009，第 169 页 9.3.3 节

```
            if(j >= w[i]) {
                const int tmp = f[i-1][j-w[i]] + v[i];
                if(tmp > f[i][j]) f[i][j] = tmp;
            }
        }
    }
}

int main() {
    int i, T;
    scanf("%d", &T);
    while(T--) {
        scanf("%d %d", &N, &W);
        for(i = 1; i <= N; ++i) scanf("%d", &v[i]);
        for(i = 1; i <= N; ++i) scanf("%d", &w[i]);

        dp();
        printf("%d\n", f[N][W]);
    }
    return 0;
}
```

01knapsack.c

版本 2，自底向上，滚动数组。

01knapsack2.c

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

#define MAXN 1000
#define MAXW 1000

int N, W;
int w[MAXN], v[MAXN];

int d[MAXW + 1]; /* 滚动数组 */

/**
 * @brief 0-1 背包问题中，处理单个物品.
 * @param[in] d 滚动数组
 * @param[in] i 该物品的下标
 * @return 无
 */
void zero_one_knapsack(int d[], const int i) {
    int j;
    for(j = W; j >= w[i]; --j) {
        const int tmp = d[j - w[i]] + v[i];
        if(tmp > d[j]) d[j] = tmp; /* 求最小用 <，求最大用 > */
    }
}

void dp() {
    int i;
    memset(d, 0, sizeof(d)); /* 背包不一定要装满 */
```

```c
    for(i = 0; i < N; ++i) zero_one_knapsack(d, i);
}

int main() {
    int i, T;
    scanf("%d", &T);
    while(T--) {
        scanf("%d %d", &N, &W);
        for(i = 0; i < N; ++i) scanf("%d", &v[i]);
        for(i = 0; i < N; ++i) scanf("%d", &w[i]);

        dp();
        printf("%d\n", d[W]);
    }
    return 0;
}
```

01knapsack2.c

相关的题目

与本题相同的题目:

- 《算法竞赛入门经典》[①]第 167 页例题 9-5
- HDOJ 2602 Bone Collector, http://acm.hdu.edu.cn/showproblem.php?pid=2602

与本题相似的题目:

- TODO

10.5.2 完全背包问题

描述

给你一个储钱罐 (piggy bank)，往里面存硬币。存入的过程是不可逆的，要想把钱拿出来只能摔碎储钱罐。因此，你想知道里面是否有足够多的钱，把它摔碎是值得的。

你可以通过储钱罐的重量来推测里面至少有多少钱。已知储钱罐空的时候的重量和装了硬币后的重量，还有每种硬币的重量和面值，每种硬币的数量不限。求在最坏情况下，储钱罐里最少有多少钱。

输入

第 1 行包含一个整数 T，表示有 T 组测试用例。每组测试用例，第一行是两个整数 E 和 F，分别表示空储钱罐的重量和装了硬币后的重量，以克 (gram) 为单位，储钱罐的重量

[①]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

不会超过 10kg，即 $1 \le E \le F \le 10000$。第二行是一个整数 $N(1 \le N \le 500)$，表示硬币的种类数目。接下来是 N 行，每行包含两个整数 v 和 w($1 \le v \le 50000, 1 \le w \le 10000$)，分别表示硬币的面值和重量。

输出

每个案例打印一行。内容是"The minimum amount of money in the piggy-bank is X."，其中 X 表示储钱罐里最少有多少钱。如果不能精确地达到给定的重量，则打印"This is impossible."。

样例输入

```
3
10 110
2
1 1
30 50
10 110
2
1 1
50 30
1 6
2
10 3
20 4
```

样例输出

```
The minimum amount of money in the piggy-bank is 60.
The minimum amount of money in the piggy-bank is 100.
This is impossible.
```

分析

每种物品有无限个可用，这是完全背包问题。

本题没有给出储钱罐的容量，但每个案例给出了，初始为空时的重量 E 和装了硬币后的重量 F，因此可以把储钱罐看作一个容量为 F-E 的背包，背包必须要装满。

这个问题非常类似于 0-1 背包问题，所不同的是每种物品有无限个。也就是从每种物品的角度考虑，与它相关的策略已并非取或不取两种，而是取 0 个、取 1 个、取 2 个……直至取 $W/w[i]$ 个。

一种好想好写的基本方法是转化为 0-1 背包问题：把第 i 种物品换成 $W/w[i]$ 个 0-1 背包问题中的物品，则得到了物品数为 $\sum \dfrac{W}{w[i]}$ 的 0-1 背包问题。时间复杂度是

$O(NW\sum\frac{W}{w[i]})$。

按照该思路，状态转移方程为:

$$f[i][j] = \max\{f[i-1][j-k*w[i]] + k*v[i], 0 \le k*w[i] \le j\}$$

伪代码如下:

```
for i = 1..N
    for j = W..w[i]
        for k = 1..j/w[i]
            d[j] = max{d[j], d[j-k*w[i]] + k*v[i]};
```

也可以写成:

```
for i = 1..N
    for k = 1..W/w[i]
        ZeroOneKnapsack(d[], w, v)
```

“拆分物品”还有更高效的拆分方法：把第 i 种物品拆分成重量为 $2^k*w[i]$、价值为 $2^k*v[i]$ 的若干物品，其中 k 取所有满足 $2^k*w[i] \le W$ 的非负整数。这是二进制的思想，因为闭区间 [1, W/w[i]] 中的任何整数都可以表示为 $1, 2, 4, ..., 2^k$ 中若干个的和。

这样处理单个物品的复杂度由 $O\left(\frac{W}{w[i]}\right)$ 降到了 $O\left(\log\frac{W}{w[i]}\right)$，伪代码如下:

```
def UnboundedKnapsack(d[], i)
    k=1
    while k*w[i] <= W
        ZeroOneKnapsack(d[], k*w[i], k*v[i])
        k=2*k
```

还存在更优化的算法，复杂度为 $O(NW)$，伪代码如下:

```
for i = 1..N
    for j = 0..W
        d[j] = max{d[j], d[j-w[i]] + v[i]};
```

与 0-1 背包问题相比，仅有一行代码不同，这里内循环是顺序的，而 0-1 背包是逆序的（在使用滚动数组的情况下)。

为什么这个算法可行呢？首先想想为什么 0-1 背包中内循环要逆序，逆序是为了保证每个物品只选一次，保证在“选择第 i 件物品”时，依赖的是一个没有选择第 i 件物品的子结果 $f[i-1][j-w[i]]$。而现在完全背包的特点却是每种物品可选无限个，没有了每个物品只选一次的限制，所以就可以并且必须采用 j 递增的顺序循环。

根据上面的伪代码，状态转移方程也可以写成这种形式:

$$f[i][j] = \max\{f[i-1][j], f[i][j-w[i]+v[i]\}$$

抽象出处理单个物品的函数:

```
def UnboundedKnapsack(d[], i)
    for j = w[i]..W
        d[j] = max(d[j], d[j-w[i]] + v[i])
```

代码

piggy_bank.c
```
#include <stdio.h>

#define MAXN 500
#define MAXW 10000
 /* 无效值，不要用 0x7FFFFFFF，执行加运算后会变成负数 */
const int INF = 0x0FFFFFFF;

int N, W;
int w[MAXN], v[MAXN];

int d[MAXW + 1]; /* 滚动数组 */

/**
 * @brief 完全背包问题中，处理单个物品.
 * @param[in] d 滚动数组
 * @param[in] i 该物品的下标
 * @return 无
 */
void unbounded_knapsack(int d[], const int i) {
    int j;
    for(j = w[i]; j <= W; ++j) {
        const int tmp = d[j - w[i]] + v[i];
        if(tmp < d[j]) d[j] = tmp; /* 求最小用 <，求最大用 > */
    }
}
/** c，物品的系数 */
void zero_one_knapsack(int d[], const int i, const int c) {
    int j;
    const int neww = c * w[i];
    const int newv = c * v[i];
    for(j = W; j >= neww; --j) {
        const int tmp = d[j - neww] + newv;
        if(tmp < d[j]) d[j] = tmp;  /* 求最小用 <，求最大用 > */
    }
}

void unbounded_knapsack1(int d[], const int i) {
    int k = 1;
    while(k * w[i] <= W) {
        zero_one_knapsack(d, i, k);
        k *= 2;
    }
}

void dp() {
    int i;
    for(i = 0; i <= W; ++i) d[i] = INF; /* 背包要装满 */
    d[0] = 0;

    for(i = 0; i < N; ++i) unbounded_knapsack(d, i);
```

```
}

int main() {
    int i, T;
    int E, F;

    scanf("%d", &T);
    while(T--) {
        scanf("%d %d", &E, &F);
        W = F - E;
        scanf("%d", &N);
        for(i = 0; i < N; ++i) scanf("%d %d", &v[i], &w[i]);

        dp();
        if(d[W] == INF) {
            printf("This is impossible.\n");
        } else {
            printf("The minimum amount of money in the piggy-bank is %d.\n",
                    d[W]);
        }
    }
    return 0;
}

/* 将第 i 种物品取 0 个, 1 个, ..., W/w[i] 个, 该版本不能 AC, 会 TLE */
void unbounded_knapsack2(int d[], const int w, const int v) {
    int j, k;
    for(j = W; j >= w; --j) {
        const int K = j / w;
        for(k = 1; k <= K; ++k) {
            const int tmp = d[j - k * w] + k * v;
            if(tmp < d[j]) d[j] = tmp; /* 求最小用 <, 求最大用 > */
        }
    }
}

/* 将第 i 种物品取 0 个, 1 个, ..., W/w[i] 个, 该版本不能 AC, 会 TLE */
void unbounded_knapsack3(int d[], const int w, const int v) {
    int k;
    const int K = W / w;
    for(k = 0; k < K; ++k){
        zero_one_knapsack(d, w, v);
    }
}
```
piggy_bank.c

相关的题目

与本题相同的题目:

- 《算法竞赛入门经典》[1]第 167 页例题 9-4

[1]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

- POJ 1384 Piggy-Bank, http://poj.org/problem?id=1384
- HDOJ 1114 Piggy-Bank, http://t.cn/zWXbXln

与本题相似的题目：

- POJ 2063 Investment, http://poj.org/problem?id=2063

10.5.3 多重背包问题

描述

某地发生地震，为了挽救灾区同胞的生命，心系灾区同胞的你准备自己采购一些粮食支援灾区，现在假设你一共有资金 W 元，而市场有 N 种大米，每种大米都是袋装产品，其价格不等，并且只能整袋购买。

请问：你用有限的资金最多能采购多少公斤粮食呢？

输入

第 1 行包含一个整数 T，表示有 T 组测试用例。每组测试用例的第一行是两个整数 W 和 N($1 \leq W \leq 100, 1 \leq N \leq 100$), 分别表示经费的金额和大米的种类，然后是 N 行数据，每行包含 3 个整数 w，v 和 c ($1 \leq w \leq 20, 1 \leq v \leq 200, 1 \leq c \leq 20$)，分别表示每袋的价格、每袋的重量以及对应种类大米的袋数。

输出

对于每组测试用例，输出能够购买大米的最大重量，你可以假设经费买不光所有的大米，并且经费你可以不用完。每个实例的输出占一行。

样例输入

```
1
8 2
2 100 4
4 100 2
```

样例输出

```
400
```

分析

第 i 种物品有 c[i] 个可用，这是多重背包问题。

与完全背包问题类似，也可以用“拆分物品”的思想把本问题转化为 0-1 背包问题：把第 i 种物品换成 $c[i]$ 个 0-1 背包问题中的物品，则得到了物品数为 $\sum c[i]$ 的 0-1 背包问题。时间复杂度是 $O(NW\sum c[i])$。状态转移方程为：

$$f[i][j]=\max\{f[i-1][j-k*w[i]]+k*v[i],0\le k\le c[i],0\le k*w[i]\le j\}$$

伪代码如下：

```
for i = 1..N
    for j = W..w[i]
        K = min{j/w[i], c[i]}
        for k = 1..K
            d[j] = max{d[j], d[j-k*w[i]] + k*v[i]};
```

也可以写成：

```
for i = 1..N
    for k = 1..c[i]
        ZeroOneKnapsack(d[], i)
```

拆分物品也可以使用二进制的技巧，把第 i 种物品拆分成若干物品，其中每件物品都有一个系数，这个新物品的重量和价值均是原来的重量和价值乘以这个系数。系数分别为 $1,2,2^2,...,2^{k-1},c[i]-2^k+1$，其中 k 是满足 $2^k-1<c[i]$ 的最大整数。例如，某种物品有 13 个，即 c[i]=13，则相应的 k=3，这种物品应该被拆分成系数分别 1,2,4,6 的四个物品。

这样处理单个物品的复杂度由 $O(c[i])$ 降到了 $O(\log c[i])$，伪代码如下：

```
// c，物品系数
def ZeroOneKnapsack(d[], i, c)
    for j = W..w[i]
        d[j] = max(d[j], d[j-c*w[i]] + c*v[i])
def BoundedKnapsack(d[], i)
    if c[i]*w[i] >= W
        unbounded_knapsack(d[], i);
        return;

    k = 1;
    while k < c[i]
        zero_one_knapsack(d[], i, k);
        c[i] -= k;
        k *= 2;

    zero_one_knapsack(d[], i, c);
```

代码

bkp.c

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

#define MAXN 100
#define MAXW 100

int N, W;
int w[MAXN], v[MAXN], c[MAXN];

int d[MAXW + 1]; /* 滚动数组 */

/** c，物品的系数 */
void zero_one_knapsack(int d[], const int i, const int c) {
    int j;
    const int neww = c * w[i];
    const int newv = c * v[i];
    for(j = W; j >= neww; --j) {
        const int tmp = d[j - neww] + newv;
        if(tmp > d[j]) d[j] = tmp;  /* 求最小用 <，求最大用 > */
    }
}

void unbounded_knapsack(int d[], const int i) {
    int j;
    for(j = w[i]; j <= W; ++j) {
        const int tmp = d[j - w[i]] + v[i];
        if(tmp > d[j]) d[j] = tmp; /* 求最小用 <，求最大用 > */
    }
}

/**
 * @brief 多重背包问题中，处理单个物品.
 * @param[in] d 滚动数组
 * @param[in] i 该物品的下标
 * @param[in] c 该物品的数量
 * @return 无
 */
void bounded_knapsack(int d[], const int i) {
    int k;
    for(k = 0; k < c[i]; ++k) {
        zero_one_knapsack(d, i, 1);
    }
}

/* 另一个版本，拆分物品更加优化 */
void bounded_knapsack1(int d[], const int i) {
    int k;
    if(c[i] * w[i] >= W) {
        unbounded_knapsack(d, i);
        return;
    }

    k = 1;
```

```
    while(k < c[i]) {
        zero_one_knapsack(d, i, k);
        c[i] -= k;
        k *= 2;
    }
    zero_one_knapsack(d, i, c[i]);
}

void dp() {
    int i;
    memset(d, 0, sizeof(d)); /* 背包不一定要装满 */

    for(i = 0; i < N; ++i) bounded_knapsack1(d, i);
}

int main() {
    int i;
    int T;
    scanf("%d", &T);
    while(T--) {
        scanf("%d %d", &W, &N);
        for(i = 0; i < N; ++i) scanf("%d %d %d", &w[i], &v[i], &c[i]);

        dp();
        printf("%d\n", d[W]);
    }
    return 0;
}
```
bkp.c

相关的题目

与本题相同的题目:

- HDOJ 2191 买大米, http://acm.hdu.edu.cn/showproblem.php?pid=2191

与本题相似的题目:

- TODO

10.6 递归结构中的动态规划

本节介绍一些递归结构中的动态规划。递归结构是指可以递归定义的数据结构，例如单链表，凸多边形、树和表达式等。

10.6.1 最优矩阵链乘

描述

一个 $m \times n$ 的矩阵乘以一个 $n \times p$ 的矩阵等于一个 $m \times p$ 的矩阵，运算量为 mnp。

矩阵乘法不满足分配律，但满足结合律，即 $A \times B \times C = (A \times B) \times C = A \times (B \times C)$。

假设 A、B、C 分别是 2×3、3×4、4×5 的矩阵，则 $(A \times B) \times C$ 的运算量为 $2 \times 3 \times 4 + 2 \times 4 \times 5 = 64$，$A \times (B \times C)$ 的运算量为 $3 \times 4 \times 5 + 2 \times 3 \times 5 = 90$，显然第一种运算顺序节省运算量。

给出 N 个矩阵组成的序列，设计一种方法把它们依次乘起来，使得总的运算量尽量小。

输入

对于每个矩阵序列，只给出它们的维度。每个序列由两个部分组成，第一行包含一个整数 N，表示矩阵的个数；接下来的 N 行，每行一对整数，分别表示矩阵的行数和列数。给出的顺序与矩阵链乘的顺序一致。最后一行 N 为 0，表示输入结束。N 不大于 10。

输出

假设矩阵命名为 $A_1, A_2, ..., A_N$。对每个测试用例输出一行，包含一个使用了小括号的表达式，清晰地指出乘法的先后顺序。每行输出以"Case x: " 为前缀，x 表示测试用例编号。如果有多个正确结果，只需要输出其中一个。

样例输入

```
3
1 5
5 20
20 1
3
5 10
10 20
20 35
6
30 35
35 15
15 5
5 10
10 20
20 25
0
```

样例输出

```
Case 1: (A1 x (A2 x A3))
Case 2: ((A1 x A2) x A3)
Case 3: ((A1 x (A2 x A3)) x ((A4 x A5) x A6))
```

分析

假设第 i 个矩阵 A_i 的维度是 $p_{i-1} \times p_i$, i=1..N。

设状态为 d[i][j]，表示子问题 $A_i, A_{i+1}, ..., A_j$ 的最优解，状态转移方程如下：

$$d[i][j] = \min\{d[i][k] + d[k+1][j] + p_{i-1}p_kp_j\}$$

代码

oams.c

```c
#include <stdio.h>
#include <memory.h>
#include <limits.h>

#define INF INT_MAX
#define MAXN 10

int N;  /** 矩阵的个数. */
int p[MAXN + 1];  /** 矩阵 Ai 的维度是 p[i-1]xp[i]. */
int d[MAXN][MAXN];  /** 状态, d[i][j] 表示子问题 Ai~Aj 的最优解. */
int s[MAXN][MAXN]; /** 子问题 Ai~Aj 应该在 s[i][j] 处断开 */

/**
 * @brief 打印子问题 Ai~Aj 的解
 * @param[in] i Ai
 * @param[in] j Aj
 * @return 无
 */
void print(const int i, const int j) {
        if (i == j) {
                printf("A%d",i);
        } else {  /* i < j */
                printf("(");
                print(i, s[i][j]);
                printf(" x ");
                print(s[i][j]+1, j);
                printf(")");
        }
}

void dp() {
    int i, j, k, l; /* l 表示区间长度 */
    for (i = 1;i <= N; ++i) d[i][i]=0;

    for (l = 2; l <= N; ++l) {
```

```
        for (i = 1; i <= N - l + 1; ++i) {
            j = i + l -1;
            d[i][j] = INF;
            for (k = i; k < j; ++k) {
                if (d[i][j] > d[i][k] + d[k+1][j] + p[i-1] * p[k] * p[j]) {
                    d[i][j] = d[i][k] + d[k+1][j] + p[i-1] * p[k] * p[j];
                    s[i][j] = k;
                }
            }
        }
    }
}

int main() {
    int i;
        int cas = 1;
        while (scanf("%d", &N) && N > 0) {
                memset(s, 0, sizeof(s));
                for (i = 0; i < N; ++i)  scanf("%d %d",&p[i],&p[i+1]);

                dp();

                printf("Case %d: ", cas++);
                print(1, N);
                printf("\n");
        }
        return 0;
}
```

oams.c

相关的题目

与本题相同的题目:

- UVa 348 Optimal Array Multiplication Sequence, http://t.cn/zH2pchp
- ZOJ 1276 Optimal Array Multiplication Sequence, http://t.cn/zH2gW1P

与本题相似的题目:

- Matrix67 - 十个利用矩阵乘法解决的经典题目, http://www.matrix67.com/blog/archives/276

10.7 最大子矩形

在一个给定的矩形网格中有一些障碍点，要找出网格内部不包含任何障碍点，且边界与坐标轴平行的最大子矩形。

遇到求矩形面积，一般把左上角设置为坐标原点，这与数学中的坐标系不同。

解决方法参考"浅谈用极大化思想解决最大子矩形问题", http://wenku.baidu.com/view/728cd5126edb6f1aff001fbb.html

方法一是一种暴力枚举法，方法二是一种动规法。

10.7.1 奶牛浴场

描述

由于 john 建造了牛场围栏，激起了奶牛的愤怒，奶牛的产奶量急剧减少。为了讨好奶牛，john 决定在牛场中建造一个大型浴场。但是 john 的奶牛有一个奇怪的习惯，每头奶牛都必须在牛场中的一个固定的位置产奶，而奶牛显然不能在浴场中产奶，于是，john 希望所建造的浴场不覆盖这些产奶点。这回，他又要求助于 clevow 了。你还能帮助 clevow 吗？

john 的牛场和规划的浴场都是矩形。浴场要完全位于牛场之内，并且浴场的轮廓要与牛场的轮廓平行或者重合。浴场不能覆盖任何产奶点，但是产奶点可以位于浴场的轮廓上。

clevow 当然希望浴场的面积尽可能大了，所以你的任务就是帮她计算浴场的最大面积。

输入

输入文件的第一行包含两个整数 L 和 W($1 \leq L, W \leq 30000$)，分别表示牛场的长和宽。文件的第二行包含一个整数 n($0 \leq n \leq 5000$)，表示产奶点的数量。以下 n 行每行包含两个整数 x 和 y，表示一个产奶点的坐标。所有产奶点都位于牛场内，即：$0 \leq x \leq l, 0 <\leq y \leq w$。

输出

输出文件仅一行，包含一个整数 S，表示浴场的最大面积。

样例输入

```
10 10
4
1 1
9 1
1 9
9 9
```

样例输出

```
80
```

分析

这里使用方法二，需要先做一定的预处理。由于第二种算法复杂度与牛场的面积有关，而题目中牛场的面积很大（30000×30000），因此需要对数据进行离散化处理。离散化后矩形的大小降为 S×S，所以时间复杂度为 O(S2)，空间复杂度为 O(S)。需要注意的是，为了保证算法能正确执行，把 (0,0) 和 (m,n) 设置为产奶点，相当于加上了一个“虚拟边界”。

代码

cow_bath.c

```c
/**
 * OJ: https://vijos.org/p/1055
 */
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#define MAXN 5001

#define max(a,b)  (a > b ? a : b)
#define min(a,b)  (a < b ? a : b)

int L, W; /* 长, 竖向 x 坐标, 宽, 横向 y 坐标 */
int n; /* n 个产奶点 */
int x[MAXN], y[MAXN]; /* 产奶点坐标 */

int int_cmp(const void *a, const void *b) {
    const int *ia = (const int*) a;
    const int *ib = (const int*) b;
    return *ia - *ib;
}

/* 等价于复制粘贴，这里为了节约篇幅，使用 include, 在 OJ 上提交时请用复制粘贴 */
#include "binary_search.c"

/**
 * @brief 求最大子矩形
 *
 * 参考 http://wenku.baidu.com/view/728cd5126edb6f1aff001fbb.html 的第二种方法
 *
 * @param[in] L 长度, 纵向
 * @param[in] W 宽度, 横向
 * @param[in] x 纵向 x 坐标
 * @param[in] y 横向 y 坐标
 * @param[in] n 障碍点的个数
 * @return 最大子矩形的面积
 */
int max_submatrix(const int L, const int W,
        const int x[], const int y[], int n) {
    int *a = (int*) malloc((n + 1) * sizeof(int));
    int *b = (int*) malloc((n + 1) * sizeof(int));
```

```
int la = n, lb = n;
int i, j;
int lm, rm; // 左边界, 右边界
int result = 0, temp;
int **v;  // 新的 01 矩阵, 1 表示障碍点
int *h;  // 高度
int *l;  // 左边界
int *r;  // 右边界

memcpy(a, x, n * sizeof(int));
memcpy(b, y, n * sizeof(int));
a[la++] = 0;
b[lb++] = 0;
a[la++] = L;
b[lb++] = W;

// 去重
qsort(a, la, sizeof(int), int_cmp);
qsort(b, lb, sizeof(int), int_cmp);
for (j = 1, i = 1; i < la; i++) { // 去重
    if (a[i] != a[i - 1])
        a[j++] = a[i];
}
la = j;
for (j = 1, i = 1; i < lb; i++) { // 去重
    if (b[i] != b[i - 1])
        b[j++] = b[i];
}
lb = j;
h = (int*) malloc(lb * sizeof(int));
l = (int*) malloc(lb * sizeof(int));
r = (int*) malloc(lb * sizeof(int));
//******************************//

// 计算 v 矩阵
v = (int**) malloc(la * sizeof(int*));
for (i = 0; i < la; i++) {
    v[i] = (int*) malloc(lb * sizeof(int));
    memset(v[i], 0, lb * sizeof(int));
}
for (i = 0; i < n; i++) {
    int ia, ib;
    ia = binary_search(a, la, x[i]);
    ib = binary_search(b, lb, y[i]);
    v[ia][ib] = 1; //标记障碍点
}

// 初始化
for (i = 0; i < lb; i++) {
    l[i] = 0;
    r[i] = W;
    h[i] = 0;
}
```

```
    for (i = 1; i < la; i++) { // 从上到下
        lm = 0;
        for (j = 0; j < lb; j++) { // 从左到右计算 l[j]
            if (!v[i - 1][j]) {  //如果上一个不是障碍点
                h[j] = h[j] + a[i] - a[i - 1]; //高度累加
                // l[i][j]=max(l[i-1][j] , 左边第一个障碍点 (i-1,j) 的位置)
                l[j] = max(l[j], lm);
            } else {  //如果上一个点是障碍点
                h[j] = a[i] - a[i - 1]; //高度重新计算
                l[j] = 0;
                r[j] = W;
                lm = b[j]; //更新 (i-1,j) 左边第一个障碍点的位置
            }
        }
        rm = W;
        for (j = lb - 1; j >= 0; j--) {  //  从右到左计算 r[j]
            //r[i][j]=min(r[i-1][j] , (i-1,j) 右边第一个障碍点的位置)
            r[j] = min(r[j], rm);
            temp = h[j] * (r[j] - l[j]);
            result = max(result, temp);  //计算最优解
            if (v[i - 1][j]) //如果该点是障碍点，更新 (i-1,j) 右边第一个障碍点的位置
                rm = b[j];
        }
    }
    // 计算横条的面积
    for (i = 1; i < la; i++) {
        temp = W * (a[i] - a[i - 1]);
        result = max(result, temp);
    }

    free(a);
    free(b);
    for (i = 0; i < la; i++) {
        free(v[i]);
    }
    free(v);
    free(h);
    free(l);
    free(r);
    return result;
}

int main() {
    int i;
    while (scanf("%d%d", &L, &W) == 2) {
        scanf("%d", &n);
        for (i = 0; i < n; i++) {
            scanf("%d%d", &x[i], &y[i]);
        }

        printf("%d\n", max_submatrix(L, W, x, y, n));
    }
```

```
    return 0;
}
```
cow_bath.c

相关的题目

与本题相同的题目:

- Vijos 1055 奶牛浴场, https://vijos.org/p/1055

与本题相似的题目:

- LeetCode Maximal Rectangle, http://leetcode.com/onlinejudge#question_85 ,
  参考代码 https://gist.github.com/soulmachine/b20a15009450016038d9
- POJ 3494 Largest Submatrix of All 1's, http://poj.org/problem?id=3494

10.7.2 最大全 1 子矩阵

描述

给定一个 $m \times n$ 的 01 矩阵，求最大的全 1 子矩阵。

输入

输入包含多组测试用例。每组测试用例第一行包含两个整数 m 和 n($1 \leq m, n \leq 2000$)，接下来是 m 行数据，每行 n 个元素。

输出

对每个测试用例，输出最大全 1 子矩阵的 1 的个数。如果输入的矩阵是全 0，则输出 0.

样例输入

```
2 2
0 0
0 0
4 4
0 0 0 0
0 1 1 0
0 1 1 0
0 0 0 0
```

样例输出

```
0
4
```

分析

注意，上一题算的是面积，这一题算的是个数，在某些细节上处理不同。

代码

largest_rectangle.c

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

#define max(a,b)  (a > b ? a : b)
#define min(a,b)  (a < b ? a : b)

#define  MAXN 2001

int matrix[MAXN][MAXN];

int lagest_rectangle(/*int **matrix, */int m, int n) {
    int i, j;
    int *H = (int*) malloc(n * sizeof(int));  // 高度
    int *L = (int*) malloc(n * sizeof(int));  // 左边界
    int *R = (int*) malloc(n * sizeof(int));  // 右边界
    int ret = 0;

    memset(H, 0, n * sizeof(int));
    memset(L, 0, n * sizeof(int));
    for (i = 0; i < n; i++) R[i] = n;

    for (i = 0; i < m; ++i) {
        int left = 0, right = n;
        // calculate L(i, j) from left to right
        for (j = 0; j < n; ++j) {
            if (matrix[i][j] == 1) {
                ++H[j];
                L[j] = max(L[j], left);
            } else {
                left = j + 1;
                H[j] = 0;
                L[j] = 0;
                R[j] = n;
            }
        }
        // calculate R(i, j) from right to left
        for (j = n - 1; j >= 0; --j) {
            if (matrix[i][j] == 1) {
                R[j] = min(R[j], right);
```

```
                ret = max(ret, H[j] * (R[j] - L[j]));
            } else {
                right = j;
            }
        }
    }

    return ret;
}

int main() {
    int m, n;
    int i, j;
    while (scanf("%d%d", &m, &n) > 0) {
        for (i = 0; i < m; i++) {
            for (j = 0; j < n; j++) {
                scanf("%d", &matrix[i][j]);
            }
        }

        printf("%d\n", lagest_rectangle(m, n));
    }
    return 0;
}
```

largest_rectangle.c

相关的题目

与本题相同的题目:

- POJ 3494 Largest Submatrix of All 1's, http://poj.org/problem?id=3494
- LeetCode Maximal Rectangle, http://leetcode.com/onlinejudge#question_85 , 参考代码 https://gist.github.com/soulmachine/b20a15009450016038d9

与本题相似的题目:

- Vijos 1055 奶牛浴场, https://vijos.org/p/1055

第 11 章 回溯法

11.1 算法思想

当把问题分成若干步骤并递归求解时，如果当前步骤没有合法选择，则函数将返回上一级递归调用，这种现象称为**回溯** (backtrack)。正是因为这个原因，枚举递归算法常被称为**回溯法**。

回溯法 = 深搜 + 剪枝。树的深搜或图的深搜都可以。深搜一般用递归来写，这样比较简洁。

回溯法比暴力枚举法快的原因，在于：暴力枚举法，是每生成一个完整的解答后，再来判断这个解答是否合法，而回溯法则在生成每一步中都进行判断，而不是等一个答案生成完毕后再来判断，这样，在每一步进行剪枝，减少了大量的废答案。

11.2 八皇后问题

描述

在 8×8 的棋盘上，放置 8 个皇后，使得她们互不攻击，每个皇后的攻击范围是同行、同列和同对角线，要求找出所有解。如图 11-1所示。

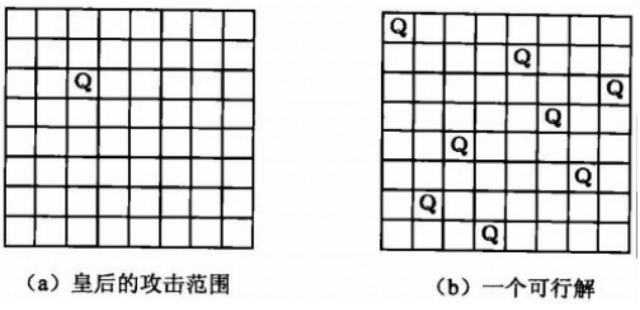


图 11-1 八皇后问题

分析

最简单的暴力枚举方法是，从 64 个格子中选一个子集，使得子集含有 8 个格子，且任意两个格子都不在同一行、同一列或同一个对角线上。这正是子集枚举问题，然而 64 个格子的子集有 2^{64} 个，太大了，这并不是一个很好的模型。

第二个思路是，从 64 个格子中选 8 个格子，这是组合生成问题。根据组合数学，有 $C_{64}^{8} \approx 4.426 \times 10^{9}$ 种方案，比第一种方案优秀，但仍然不够好。

经过思考不难发现，由于每一行只能放一个皇后，那么第一行有 8 种选择，第二行有 7 中选择，$\cdots$，第 8 行有 1 中选择，总共有 $8! = 40320$ 个方案。如果用 C[x] 表示第 x 行皇后的列编号，则问题变成了一个全排列生成问题，枚举量不会超过 8!。

代码

eight_queen.c

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

#define N 8 // 皇后的个数，也是棋盘的长和宽

int total = 0;  // 可行解的总数
int C[N];  // C[i] 表示第 i 行皇后所在的列编号

/**
 * @brief 输出所有可行的棋局，按列打印.
 *
 * http://poj.grids.cn/practice/2698/ ，这题需要按列打印
 *
 * @return 无
 */
void output() {
    int i, j;
    printf("No. %d\n", total);
    for (j = 0; j < N; ++j) {
        for (i = 0; i < N; ++i) {
            if (C[i] != j) {
                printf("0 ");
            } else {
                printf("1 ");
            }
        }
        printf("\n");
    }
}

/**
 * @brief 输出所有可行的棋局，按行打印.
 * @return 无
 */
```

```
void output1() {
    int i, j;
    printf("No. %d\n", total);
    for (i = 0; i < N; ++i) {
        for (j = 0; j < N; ++j) {
            if (j != C[i]) {
                printf("0 ");
            } else {
                printf("1 ");
            }
        }
        printf("\n");
    }
}

/**
 * @brief 检查当前位置 (row, column) 能否放置皇后.
 *
 * @param[in] row 当前行
 * @return 能则返回 1, 不能则返回 0
 */
int check(const int row, const int column) {
    int ok = 1;
    int i;
    for(i = 0; i < row; ++i) {
        // 两个点的坐标为 (row, column), (i, C[i])
        // 检查是否在同一列, 或对角线上
        if(column == C[i] || row - i == column - C[i] ||
            row - i == C[i] - column) {
            ok = 0;
            break;
        }
    }
    return ok;
}

/**
 * @brief 八皇后, 回溯法
 *
 * @param[in] row 搜索当前行, 该在哪一列上放一个皇后
 * @return 可行解的个数
 */
int search(const int row) {
    if(row == N) {  // 递归边界, 只要走到了这里, 意味着找到了一个可行解
        ++total;
        output();
    } else {
        int j;
        for(j = 0; j < N; ++j) {  // 一列一列的试
            const int ok = check(row, j);
            if(ok) {  // 如果合法, 继续递归
                C[row] = j;
                search(row + 1);
```

```
            }
        }
    }
    return total;
}

// 表示已经放置的皇后
// 占据了哪些列
int columns[N];
// 占据了哪些主对角线
int principal_diagonals[2 * N];
// 占据了哪些副对角线
int counter_diagonals[2 * N];

/**
 * @brief 检查当前位置 (row, column) 能否放置皇后.
 *
 * @param[in] row, 当前行
 * @return 能则返回 1, 不能则返回 0
 */
int check2(const int row, const int column) {
    return columns[column] == 0 && principal_diagonals[row + column] == 0
        && counter_diagonals[row - column + N] == 0;
}

/**
 * @brief 八皇后, 回溯法, 更优化的版本, 用空间换时间
 *
 * @param[in] row 搜索当前行, 该在哪一列上放一个皇后
 * @return 可行解的个数
 */
int search2(const int row) {
    if(row == N) {  // 递归边界, 只要走到了这里, 意味着找到了一个可行解
        ++total;
        output();
    } else {
        int j;
        for(j = 0; j < N; ++j) {  // 一列一列的试
            const int ok = check2(row, j);
            if(ok) {  // 如果合法, 继续递归
                C[row] = j;
                columns[j] = principal_diagonals[row + j] =
                    counter_diagonals[row - j + N] = 1;
                search2(row + 1);
                // 恢复环境
                columns[j] = principal_diagonals[row + j] =
                    counter_diagonals[row - j + N] = 0;
            }
        }
    }
    return total;
}
```

```
int main() {
    // search(0);
    search2(0);
    return 0;
}
```
eight_queen.c

相关的题目

与本题相同的题目:

- 《算法竞赛入门经典》[1] 第 123 页 7.4.1 节
- 百练 2698 八皇后问题, http://poj.grids.cn/practice/2698/
- wikioi 1295 N 皇后问题, http://www.wikioi.com/problem/1295/

与本题相似的题目:

- POJ 1321 棋盘问题, http://poj.org/problem?id=1321

11.3 还原 IP 地址

描述

本题是 LeetCode Online Judge 上的"Restore IP Addresses"。

给定一个只包含数字的字符串，还原出所有合法的 IP 地址。

例如：给定"25525511135"，返回 ["255.255.11.135", "255.255.111.35"]。(顺序无关紧要)

分析

这题很明显分为四步，有层次，因此可以尝试用回溯法解决。

代码

restore_ip_adresses.cpp

```
#include <iostream>
#include <vector>
using namespace std;

class Solution {
public:
    vector<string> restoreIpAddresses(string s) {
        vector<string> result;
        string ip;
        dfs(s, 0, 0, ip, result);
```

[1] 刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

```
        return result;
    }

    /**
     * @brief 解析字符串
     * @param[in] s 字符串
     * @param[in] startIndex 从 s 的哪里开始
     * @param[in] step 当前步骤编号，从 0 开始编号，取值为 0,1,2,3,4 表示结束了
     * @param[in] intermediate 当前解析出来的中间结果
     * @param[out] result 存放所有可能的 IP 地址
     * @return 无
     */
    void dfs(string s, int startIndex, int step, string intermediate,
            vector<string> &result) {
        if (s.size() - startIndex > (4 - step) * 3)
            return;  // 非法结果，剪枝
        if (s.size() - startIndex < (4 - step))
            return;  // 非法结果，剪枝

        if (startIndex == s.size() && step == 4) {  // 找到一个合法解
            intermediate.resize(intermediate.size() - 1);
            result.push_back(intermediate);
            return;
        }

        int num = 0;
        for (int i = startIndex; i < startIndex + 3; i++) {
            num = num * 10 + (s[i] - '0');

            if (num <= 255) {  // 当前结点合法，则继续往下递归
                intermediate += s[i];
                dfs(s, i + 1, step + 1, intermediate + '.', result);
            }
            if (num == 0) {  // 非法结果，剪枝
                break;
            }
        }
    }
};
```

restore_ip_adresses.cpp

11.4 Combination Sum

描述

本题是 LeetCode Online Judge 上的"Combination Sum"。

给定一个数的集合 (C) 和一个目标数 (T)，找到 C 中所有不重复的组合，让这些被选出来的数加起来等于 T。

每一个数可以被选无数次。

注意:

- 所有的数（包括目标）都是正整数
- 一个组合（$a_1, a_2, \cdot, a_k$）中的元素必须以非递减顺序排列
- 一个组合不能与另一个组合重复

例如，给定一组数 2,3,6,7, 和目标 7，则答案是

```
[7]
[2, 2, 3]
```

分析

这题没有固定的步骤数，但是步骤也是有限的，因此可以尝试用回溯法。

代码

combination_sum.cpp

```
vector<vector<int> > combinationSum(vector<int> &nums, int target) {
    std::sort(nums.begin(), nums.end());
    vector<vector<int> > result;
    vector<int> intermediate;
    dfs(nums, target, 0, intermediate, result);
    return result;
}
void dfs(vector<int>& nums, int gap, int level,
        vector<int>& intermediate, vector<vector<int> > &result) {
    if (gap == 0) {  // 找到一个合法解
        result.push_back(intermediate);
        return;
    }
    for (int i = level; i < nums.size(); i++) {
        if (gap < nums[i]) return; // 剪枝

        intermediate.push_back(nums[i]);
        dfs(nums, gap - nums[i], i, intermediate, result);
        intermediate.pop_back();  // 恢复环境
    }
}
```

combination_sum.cpp

11.5 Combination Sum II

描述

本题是 LeetCode Online Judge 上的”Combination Sum II”。

本题与上一题唯一不同的是，每个数只能使用一次。

分析

这题没有固定的步骤数，但是步骤也是有限的，因此可以尝试用回溯法。

代码

combination_sum2.cpp

```cpp
vector<vector<int> > combinationSum2(vector<int> &nums, int target) {
    sort(nums.begin(), nums.end());
    vector<vector<int> > result;
    vector<int> intermediate;
    dfs(nums, target, 0, intermediate, result);
    return result;
}
void dfs(vector<int> &nums, int gap, int index,
        vector<int> &intermediate, vector<vector<int> > &result) {
    if (gap == 0) {  //  找到一个合法解
        result.push_back(intermediate);
        return;
    }

    int previous = -1;
    for (int i = index; i < nums.size(); i++) {
        // 如果当前循环没有选 nums[i]，则下一次循环就不能再选 nums[i]
        if (previous == nums[i])
            continue;
        if (gap < nums[i]) return;  // 剪枝

        previous = nums[i];

        intermediate.push_back(nums[i]);
        dfs(nums, gap - nums[i], i + 1, intermediate, result);
        intermediate.pop_back();  // 恢复环境
    }
}
```

combination_sum2.cpp

第 12 章 图

在 ACM 竞赛中，图一般使用邻接矩阵表示，代码框架如下；

graph.c

```
#define MAXN 100  // 顶点最大个数

int n; // 顶点个数
int G[MAXN][MAXN]; // 邻接矩阵
int visited_edges[MAXN][MAXN]; // 边的访问历史记录
int visited_vertices[MAXN]; // 顶点的访问历史记录
```

graph.c

12.1 深度优先搜索

图的深度优先搜索的代码框架如下:

graph.c

```
/**
 * @brief 图的深度优先搜索代码框架，搜索边.
 * @param[in] u 出发顶点
 * @param[in] n 顶点个数
 * @param[in] G 图的临街举着
 * @param[in] visited 边的访问历史记录
 * @return 无
 * @remark 在使用的时候，为了降低递归的内存占用量，可以把
 * n, G, visited 抽出来作为全局变量
 */
void dfs(const int u,
                const int n, const int G[][MAXN], int visited[][MAXN]) {
    int v;
    for(v = 0;  v < n; v++) if(G[u][v] && !visited[u][v]) {
        visited[u][v] = visited[v][u] = 1; // 无向图用这句
        // visited_edges[u][v] = 1; // 有向图用这句
        dfs(v, n, G, visited);
        // 这里写逻辑代码
        // printf("%d %d\n", u, v);
    }
}

/**
 * @brief 图的深度优先搜索代码框架，搜索顶点.
```

```
 * @param[in] u 出发顶点
 * @param[in] n 顶点个数
 * @param[in] G 图的临街举着
 * @param[in] visited 顶点的访问历史记录
 * @return 无
 * @remark 在使用的时候，为了降低递归的内存占用量，可以把
 * n, G, visited 抽出来作为全局变量
 */
void dfs(const int u,
                const int n, const int G[][MAXN], int visited[MAXN]) {
    int v;
    visited[u] = 1;
    for(v = 0;  v < n; v++) if(G[u][v] && !visited[v]) {
        dfs(v, n, G, visited);
        // 这里写逻辑代码
        // printf("%d %d\n", u, v);
    }
}
```

graph.c

12.1.1　黑白图像

描述

输入一个 n*n 的黑白图像（1 表示黑丝，0 表示白色），任务是统计其中八连块的个数。如果两个黑格子有公共边或者公共定点，就说它们属于同一个八连块。如图 12-1所示的黑白图像中有 3 个八连块。

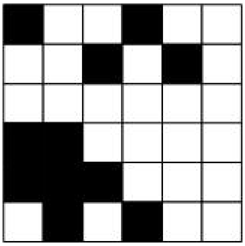


图 12-1　拥有 3 个八连块的黑白图

代码

blackwhite_image.c

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

#define MAXN 16

int n;
// 黑白图，1 表示黑色，0 表示白色，加一圈 0，用于判断出界
```

```
int G[MAXN + 1][MAXN + 1];
// 记录格子 (x,y) 是否已经被访问过
int visitied[MAXN][MAXN];

void dfs(const int x, const int y) {
    // 曾经访问过这个格子，或者当前格子是白色
    if(G[x][y] == 0 || visitied[x][y] == 1)  return;

    visitied[x][y] = 1; // 标记 (x,y) 已访问过
    // 递归访问周围的 8 个格子
    dfs(x - 1, y - 1); // 左上角
    dfs(x - 1, y); // 正上方
    dfs(x - 1, y + 1); // 右上角
    dfs(x, y - 1); // 左边
    dfs(x, y + 1); // 右边
    dfs(x + 1, y - 1); // 左下角
    dfs(x + 1, y); // 正下方
    dfs(x + 1, y + 1); // 右下角
}

/*
Sample Input
6
100100
001010
000000
110000
111000
010100
Sample Output
3
*/
int main() {
    int i, j;
    char s[MAXN]; // 矩阵的一行
    int count = 0; // 八连块的个数

    scanf("%d", &n);
    memset(G, 0, sizeof(G));
    memset(visitied, 0, sizeof(visitied));

    for(i = 0; i < n; ++i) {
        scanf("%s", s);
        for(j = 0; j < n; ++j) {
            G[i + 1][j + 1] = s[j] - '0'; // 把图像往中间挪一点，空出一圈白格子
        }
    }


    for(i = 1; i <= n; ++i) {
        for(j = 1; j <= n; ++j) {
            if(visitied[i][j] == 0 && G[i][j] == 1) {
                count++;
```

```
                dfs(i, j);
            }
        }
    }
    printf("%d\n", count);
    return 0;
}
```
blackwhite_image.c

相关的题目

与本题相同的题目：

- 《算法竞赛入门经典》[①] 第 107 页 6.4.1 节
- TODO

与本题相似的题目：

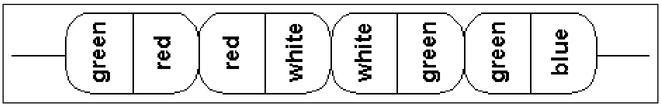
- TODO

12.1.2 欧拉回路

描述

本题是 UVA 10054 - The Necklace。

My little sister had a beautiful necklace made of colorful beads. Two successive beads in the necklace shared a common color at their meeting point. The figure below shows a segment of the necklace:



But, alas! One day, the necklace was torn and the beads were all scattered over the floor. My sister did her best to recollect all the beads from the floor, but she is not sure whether she was able to collect all of them. Now, she has come to me for help. She wants to know whether it is possible to make a necklace using all the beads she has in the same way her original necklace was made and if so in which order the bids must be put.

Please help me write a program to solve the problem.

[①] 刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

Input

The input contains T test cases. The first line of the input contains the integer T.

The first line of each test case contains an integer $N (5 \leq N \leq 1000)$ giving the number of beads my sister was able to collect. Each of the next N lines contains two integers describing the colors of a bead. Colors are represented by integers ranging from 1 to 50.

Output

For each test case in the input first output the test case number as shown in the sample output. Then if you apprehend that some beads may be lost just print the sentence "some beads may be lost" on a line by itself. Otherwise, print N lines with a single bead description on each line. Each bead description consists of two integers giving the colors of its two ends. For $1 \leq i \leq N_1$, the second integer on line i must be the same as the first integer on line i + 1. Additionally, the second integer on line N must be equal to the first integer on line 1. Since there are many solutions, any one of them is acceptable.

Print a blank line between two successive test cases.

Sample Input

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

Sample Output

```
Case \#1
some beads may be lost

Case \#2
2 1
1 3
3 4
4 2
2 2
```

分析

这题就是欧拉回路 + 打印路径。

如果能从图的某一顶点出发，每条边恰好经过一次，这样的路线称为**欧拉道路** (Eulerian Path)。如果每条边恰好经过一次，且能回到起点，这样的路线称为**欧拉回路** (Eulerian Circuit)。

对于无向图 G，当且仅当 G 是连通的，且最多有两个奇点，则存在欧拉道路。如果有两个奇点，则必须从其中一个奇点出发，到另一个奇点终止。

如果没有奇点，则一定存在一条欧拉回路。

对于有向图 G，当且仅当 G 是连通的，且每个点的入度等于出度，则存在欧拉回路。

如果有两个顶点的入度与出度不相等，且一个顶点的入度比出度小 1，另一个顶点的入度比出度大 1，此时，存在一条欧拉道路，以前一个顶点为起点，以后一个顶点为终点

代码

eulerian_circuit.c

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

#define MAXN 51  // 顶点最大个数

int G[MAXN][MAXN];
int visited_vertices[MAXN];
int visited_edges[MAXN][MAXN];
int count[MAXN]; // 顶点的度

void dfs(const int u) {
    int v;
    visited_vertices[u] = 1;
    for(v = 0;  v < MAXN; v++) if(G[u][v] && !visited_vertices[v]) {
        dfs(v);
    }
}

/*
 * @brief 欧拉回路，允许自环和重复边
 * @param[in] u 起点
 * @return 无
 */
void euler(const int u){
    int v;
    for(v = 0; v < MAXN; ++v) if(G[u][v]){
        --G[u][v]; --G[v][u]; // 这个技巧，即有 visited 的功能，又允许重复边
        euler(v);
        // 逆向打印，或者存到栈里再打印
        printf("%d %d\n", u, v);
    }
```

```
}

int main() {
    int T, N, a, b;
    int i;
    int cases=1;
    scanf("%d",&T);
    while(T--) {
        int flag = 1; // 结点的度是否为偶数
        int flag2 = 1; // 图是否是连通的

        memset(G, 0, sizeof(G));
        memset(count, 0, sizeof(count));

        scanf("%d",&N);
        for(i = 0; i < N; ++i){
            scanf("%d %d", &a, &b);
            ++G[a][b];
            ++G[b][a];
            ++count[a];
            ++count[b];
        }

        printf("Case #%d\n", cases++);

        // 欧拉回路形成的条件之一，判断结点的度是否为偶数
        for(i=0; i<MAXN; ++i) {
            if(count[i] & 1){
                flag = 0;
                break;
            }
        }
        // 检查图是否连通
        if(flag) {
            memset(visited_vertices, 0, sizeof(visited_vertices));
            memset(visited_edges, 0, sizeof(visited_edges));

            for(i=0; i< MAXN; ++i)
                if(count[i]) {
                    dfs(i);
                    break;
                }
            for(i=0; i< MAXN; ++i){
                if(count[i] && !visited_vertices[i]) {
                    flag2 = 0;
                    break;
                }
            }
        }
        if (flag && flag2) {
            for(i = 0; i < MAXN; ++i) if(count[i]){
                euler(i);
                break;
```

```
            }
        } else {
            printf("some beads may be lost\n");
        }

        if(T > 0) printf("\n");
    }
    return 0;
}
```

eulerian_circuit.c

相关的题目

与本题相同的题目:

- 《算法竞赛入门经典》[1] 第 111 页 6.4.4 节
- TODO

与本题相似的题目:

- UVa 10129 Play on Words, http://t.cn/zTInBDX

12.2 广度优先搜索

我们通常说的 BFS，默认指的是单向 BFS，此外还有双向 BFS。

12.2.1 走迷宫

描述

一个迷宫由 n 行 m 列的单元格组成，每个单元格要么是空地（用 0 表示），要么是障碍物（用 1 表示）。你的任务是找到一条从入口到出口的最短移动序列，其中 UDLR 分别表示上下左右四个方向。任何时候都不能再障碍物格子中，也不能走到迷宫之外。入口和出口保证是空地。$n, m \leq 100$。

分析

既然求的是“最短”，很自然的思路是用 BFS。举个例子，在如下图所示的迷宫中，假设入口是左上角 $(0, 0)$，我们就从入口开始用 BFS 遍历迷宫，就可以算出从入口到所有点的最短路径（如图 12-2(a) 所示），以及这些路径上每个节点的前驱（如图 12-2(b) 所示）。

[1] 刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

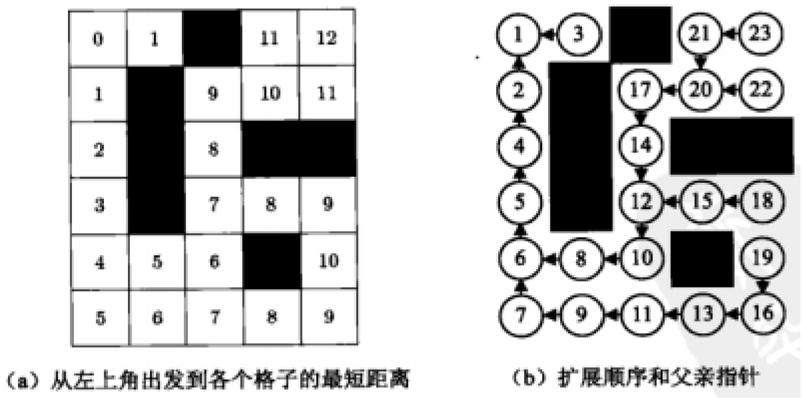


图 12-2 用 BFS 求迷宫中最短路径

代码

maze.c

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

#define MAXN 100

// 迷宫的行数，列数
int n, m;
// 迷宫，0 表示空地，1 表示障碍物
int G[MAXN][MAXN];
// 标记格子是否已访问过
int visited[MAXN][MAXN];
// 每个格子的前驱
int father[MAXN][MAXN];
// 前趋到该格子的前进方向
int last_direction[MAXN][MAXN];

// 四个方向
const char name[4] = {'U','R','D','L'};
const int dx[4] = {-1, 0, 1, 0}; // 行
const int dy[4] = {0, 1, 0, -1}; // 列

// 队列
int q[MAXN * MAXN];

/*
 * @brief 广搜
 *
 * @param[in] x 入口的 x 坐标
 * @param[in] y 入口的 y 坐标
 * @return 无
 */
void bfs(int x, int y) {
```

```
    int front = 0, rear = 0;
    int u = x * m + y;
    int d; // 方向

    father[x][y] = u; // 打印路径时的终止条件
    visited[x][y] = 1;// 千万别忘记了标记此处的访问记录
    q[rear++] = u;
    while (front < rear) {
        u = q[front++];
        x = u / m;        y = u % m;
        for(d = 0; d < 4; d++) { // 代表四个方向
            const int nx = x + dx[d];
            const int ny = y + dy[d];

            if (nx >= 0 && nx < n && ny >= 0 && ny < m &&  // //(nx, ny) 没有出界
                !G[nx][ny] && !visited[nx][ny]) { // 不是障碍且没被访问过
                const int v = nx * m + ny;
                q[rear++] = v;
                father[nx][ny] = u; // 记录 (nx, ny) 的前趋
                visited[nx][ny] = 1; // 访问记录
                last_direction[nx][ny] = d; // 记录从 (x, y) 到 (nx, ny) 的方向
            }
        }
    }
}

/*
 * @brief 递归实现路径输出
 *
 * 如果格子 (x, y) 有父亲 (fx, fy)，需要先打印出从入口到 (fx, fy) 的最短路径，然后再
 * 打印从 (fx, fy) 到 (x,y) 的移动方向。
 *
 * @param[in] x 目标点的 x 坐标
 * @param[in] y 目标点的 y 坐标
 * @return 无
 */
void print_path_r(const int x, const int y) {
    const int fx = father[x][y] / m;
    const int fy = father[x][y] % m;
    if (fx != x || fy != y) {
        print_path_r(fx, fy);
        putchar(name[last_direction[x][y]]);
    }
}

int direction[MAXN * MAXN];
/*
 * @brief 显式栈实现路径输出
 *
 * @param[in] x 目标点的 x 坐标
 * @param[in] y 目标点的 y 坐标
 * @return 无
 */
```

```c
void print_path(int x, int y) {
    int c = 0;
    while(1) {
        const int fx = father[x][y] / m;
        const int fy = father[x][y] % m;
        if (fx == x && fy == y) break;
        direction[c++] = last_direction[x][y];
        x = fx;
        y = fy;
    }
    while (c--) {
        putchar(name[direction[c]]);
    }
}

/*
Sample Input
6 5
00100
01000
01011
01000
00010
00000
Sample Output
(0,0)-->(0,4), DDDDRRUUURUR
*/
int main(void) {
    int i, j;
    char s[MAXN];

    scanf("%d%d", &n, &m);

    for(i = 0; i < n; i++) {
        scanf("%s", s);
        for(j = 0; j < m; j++) {
            G[i][j] = s[j] - '0';
        }
    }

    printf(" 从入口到出口迷宫路径: \n");
    bfs(0, 0);        // (0, 0) 是入口
    print_path(0, 4); // (0, 4) 是出口
    printf("\n");
    return 0;
}
```

maze.c

相关的题目

与本题相同的题目:

- 《算法竞赛入门经典》[1]第 108 页 6.4.2 节
- POJ 3984 迷宫问题, http://poj.org/problem?id=3984

与本题相似的题目：

- POJ 2049 Finding Nemo, http://poj.org/problem?id=2049

12.2.2 八数码问题

描述

编号为 1~8 的 8 个正方形滑块摆成 3 行 3 列，有一个格子空着，如图 12-3所示。

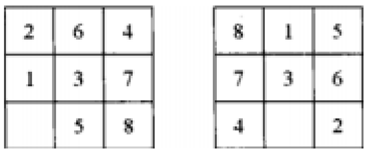


图 12-3 用 BFS 求迷宫中最短路径

每次可以把与空格相邻的滑块（有公共边才算相邻）移到空格中，而它原来的位置就成了新的空格。目标局面固定如下（用 x 表示空格）：

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

给定初始局面，计算出最短的移动路径。

输入

用一行表示一个局面，例如下面的这个局面：

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

可以表示为 1 2 3 x 4 6 7 5 8。

输出

如果有解答，输出一个由四个字母’r’,’l’,’u’,’d’ 组成的移动路径。如果没有，输出”unsolvable”。

[1]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

样例输入

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

样例输出

```
ullddrurdllurdruldr
```

分析

计算“最短”，很自然的想到 BFS。

如何表示一个状态？本题是一个 3*3 的棋盘，状态有 9! 个，可以用一个 32 位整数表示，但 15! 已经超过 32 位整数的范围，21！ 超过了 64 位整数的范围，因此 4*4 的棋盘可以用一个 64 位整数表示。超过 4*4 的棋盘，则无法用整数来表示了，可以用一个数组来表示。

怎么判断一个状态已经访问过？用哈希表或者集合。哈希表的话，由于 C++ STL 还没有 `std::hashset`，需要自己实现哈希表，然后由于本题的特殊性，存在一种完美哈希 (perfect hashing) 方案。集合可以直接使用 `std::set`。总结起来，有以下三个方法：

- 把排列变成整数，这是一种完美哈希，即不存在冲突
- 用普通的哈希表，这种方法通用一些，速度也略慢。手工实现哈希表，把哈希值相同的组成一个单链表，
- 用 `std::set` 实现判重，代码最短，速度也最慢（本题用这个方法会 TLE）。建议把该方法作为“跳板”，先写一个 STL 版的程序，确保主算法正确，然后把 `std::set` 替换成自己写的哈希表。

此题更优的解法还有双向 BFS（见 §12.3），A* 算法（见 §12.8）。

代码

eight_digits_bfs.c

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

#define DIGITS 9 // 棋盘中数字的个数，也是变进制数需要的位数
#define     MATRIX_EDGE 3       // 棋盘边长

// 3x3 的棋盘，状态最多有 9! 种
#define     MAX         362880

typedef int state_t[DIGITS]; // 单个状态

state_t q[MAX]; // 队列，也是哈希表
```

```
int front, rear;
int distance[MAX - 1]; // 由初始状态到本状态的最短步数
int father[MAX - 1]; // 父状态，初始状态无父状态
char move[MAX - 1]; // 父状态到本状态的移动方向

// 目标状态
const int goal[] = {1, 2, 3, 4, 5, 6, 7, 8, 0};
const int space_number = 0; // 空格对应着数字 0

// 上下左右四个方向
const int dx[] = {-1, 1, 0, 0};
const int dy[] = {0, 0, -1, 1};
const char dc[] = { 'u', 'd', 'l', 'r' };

/**
 * @brief 初始化哈希表.
 * @return 无
 */
void init_lookup_table(); // 版本 1

/**
 * @brief 插入到 visited 表中.
 * @param[in] index 状态在队列中的位置
 * @return 成功返回 1，失败返回 0
 */
int try_to_insert(const int index);

void init_lookup_table_hash(); // 版本 2
int try_to_insert_hash(const int index);

void init_lookup_table_stl(); // 版本 3
int try_to_insert_stl(const int index);

/**
 * @brief 单向 BFS.
 * @return 返回目标状态在队列 q 中的下标 ，失败则返回 0
 */
int bfs() {
    // 三个版本随意切换
    init_lookup_table();
    // init_lookup_table_hash();
    // init_lookup_table_stl(); // 这个版本会 Time Limit Exceeded

    while (front < rear) {
        int x, y, z, d;
        const state_t*s = &(q[front]);
        if (memcmp(goal, *s,sizeof(state_t)) == 0) {
            return front;  // 找到目标状态，成功返回
        }
        for (z = 0; z < DIGITS; z++)  if ((*s)[z] == space_number) {
            break;  // 找 0 的位置
        }
```

```
        x=z / MATRIX_EDGE, y=z % MATRIX_EDGE; // 获取行列编号
        for (d=0; d < 4; d++) {  // 向四个方向扩展
            const int newx = x + dx[d];
            const int newy = y + dy[d];
            const int newz = newx * MATRIX_EDGE + newy;

            if (newx >= 0 && newx < MATRIX_EDGE && newy >= 0 &&
                newy < MATRIX_EDGE) { // 没有越界
                state_t *t = &(q[rear]);
                memcpy(t, s, sizeof((*s)));
                assert((*s)[z] == space_number);
                (*t)[newz] = space_number;
                (*t)[z] = (*s)[newz];

                // 三个版本随意切换
                if (try_to_insert(rear)) { // 利用查找表判重
                // if (try_to_insert_hash(rear)) {
                // if (try_to_insert_stl(rear)) {
                    father[rear] = front;
                    move[rear] = dc[d];
                    distance[rear] = distance[front] + 1;
                    rear++;
                }
            }
        }

        front++;
    }

    return 0;//失败
}


/**
 * @brief 输入.
 * @return 无
 */
void input() {
    int ch, i;
    for (i = 0; i < DIGITS; ++i) {
        do {
            ch = getchar();
        } while ((ch != EOF) && ((ch < '1') || (ch > '8')) && (ch != 'x'));
        if (ch == EOF) return;
        if (ch == 'x') q[0][i] = 0; // x 映射成数字 0
        else           q[0][i] = ch - '0';
    }
    front = 0; rear = 1;
    father[0] = 0; // 初始状态无父状态
    distance[0] = 0;
    move[0] = -1;
    return;
}
```

```
int top = -1;
char stack[MAX];
/**
 * @brief 打印从初始状态到目标状态的移动序列.
 * @param[in] index 目标状态在队列 q 中的下标
 * @return 无
 */
void output(const int index) {
    int i;
    for (i = index; i > 0; i = father[i]) {
        stack[++top] = move[i];
    }
    for (i = top; i >= 0; --i) {
        printf("%c", stack[i]);
    }
    printf("\n");
}

int main() {
    int ans;

    input();

    ans = bfs();
    if ( ans > 0) {
        output(ans);
    } else {
        printf("no solution\n");
    }
    return 0;
}

/********************** 方案 1 把排列变成整数 ************************/
// 9 位变进制数（空格）能表示 0 到 (9!-1) 内的所有自然数，恰好有 9! 个，
// 与状态一一对应，因此可以把状态一一映射到一个 9 位变进制数

// 9 位变进制数，每个位数的单位，0!~8!
const int fac[] = {40320, 5040, 720, 120, 24, 6, 2, 1, 1};

// 采用本方案，由于是完美哈希，没有冲突，
// 可以用 MAX 代替 MAX_HASH_SIZE，减少内存占用量
int visited[MAX]; // 历史记录表

/** 初始化哈希表. */
void init_lookup_table() {
    memset(visited, 0, sizeof(visited));
}

/**
 * @brief 计算状态的 hash 值，这里用康托展开，是完美哈希.
 *
 * @param[in] s 状态
```

```
 * @return 序数，作为 hash 值
 */
int hash(const state_t *s) {
    int i, j;
    int key = 0; // 将 q[index] 映射到整数 key
    for (i = 0; i < 9; i++) {
        int cnt = 0;
        for (j = i + 1; j < 9; j++) if ((*s)[i] > (*s)[j]) cnt++;
        key += fac[i] * cnt;
    }
    return key;
}

/**
 * @brief 插入到 visited 表中.
 * @param[in] index 状态在队列中的位置
 * @return 成功返回 1，失败返回 0
 */
int try_to_insert(const int index) {
    const int key = hash(&q[index]); // 将 q[index] 映射到整数 code

    if (visited[key]) return 0;
    else visited[key] = 1;

    return 1;
}

/************************* 方案 2 哈希表 *********************************/
#define MAX_HASH_SIZE  1000000  // 状态的哈希表容量，比 9! 大即可

int head[MAX_HASH_SIZE];
int next[MAX];

void init_lookup_table_hash() {
    memset(head, 0, sizeof(head));
    memset(next, 0, sizeof(next));
}
int hash2(const state_t *s) {
    int i;
    int v = 0;
    for(i = 0; i < 9; i++) v = v * 10 + (*s)[i];
    return v % MAX_HASH_SIZE;
}

int try_to_insert_hash(const int index) {
    const int h = hash2(&(q[index]));
    int u = head[h]; // 从表头开始查找单链表
    while(u) {
        // 找到了，插入失败
        if(memcmp(q[u], q[index], sizeof(state_t)) == 0) return 0;
        u = next[u]; // 顺着链表继续找
    }
    next[index] = head[h]; // 插入到链表中
```

```
    head[h] = index; // head[h] 和 next[index] 组成了一个节点
    return 1;
}

/*************************** 方案 3 STL***********************************/
#include <set>
struct cmp {
    bool operator() (int a, int b) const {
        return memcmp(&q[a], &q[b], sizeof(state_t)) < 0;
    }
};
std::set<int, cmp> visited_set;

void init_lookup_table_stl() { visited_set.clear(); }

int try_to_insert_stl(const int index) {
    if (visited_set.count(index)) return 0;
    visited_set.insert(index);
    return 1;
}
```
eight_digits_bfs.c

相关的题目

与本题相同的题目:

- 《算法竞赛入门经典》[1] 第 131 页 7.5.3 节
- POJ 1077 Eight, http://poj.org/problem?id=1077

与本题相似的题目:

- POJ 2893 M × N Puzzle, http://poj.org/problem?id=2893

12.3 双向 BFS

12.3.1 八数码问题

题目见 §12.2.2。

代码

eight_digits_bibfs.c

eight_digits_bibfs.c

[1] 刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

12.4 最小生成树

“最小”指的是边的权值之和最小。

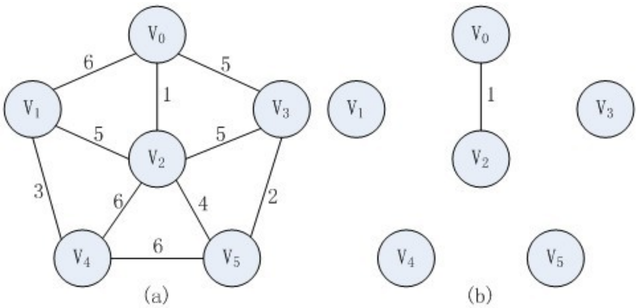
构造最小生成树 (Minimum Spanning Tree, MST) 有多种算法。其中多数算法利用了最小生成树的一个性质（简称为 MST 性质）：假设 $N=(V,E)$ 是一个连通网，U 是顶点集 V 的一个非空子集。若 (u,v) 是一条具有最小权值的边，其中 $u\in U,v\in V-U$，则必存在一颗包含边 (u,v) 的最小生成树。

Prim 算法和 Kruskal 算法是两个利用 MST 性质构造最小生成树的算法。它们都属于贪心法。

12.4.1 Prim 算法

假设 $N=(V,E)$ 是一个连通网，TE 是 N 上最小生成树中边的集合。算法从 $U=u_0(u_0\in V),TE=\{\}$ 开始，重复执行下述操作：在所有 $u\in U,v\in V-U$ 的边 $(u,v)\in E$ 中找一条代价最小的边 (u_0,v_0) 并入集合 TE，同时 v_0 并入 U，直至 $U=V$ 为止。此时 TE 中必有 $n-1$ 条边，则 $T=(V,TE)$ 为 N 的最小生成树。为实现这个算法需附设一个数组 `closedge`，以记录从 U 到 $V-U$ 具有最小代价的边。对每个顶点 $v_i\in V-U$，在辅助数组中存在一个相应分量 `closedge[i-1]`，它包括两个域，其中 `lowcost` 存储该边上的权。显然，$closedge[i].lowcost=\min\{cost(u,v_i),u\in U\}$。`adjvex` 域存储该边依附的在 U 中的顶点。

图 12-4所示为按 Prim 算法构造网的一棵最小生成树的过程，在构造过程中辅助数组中各分量值的变化如表12-1所示。



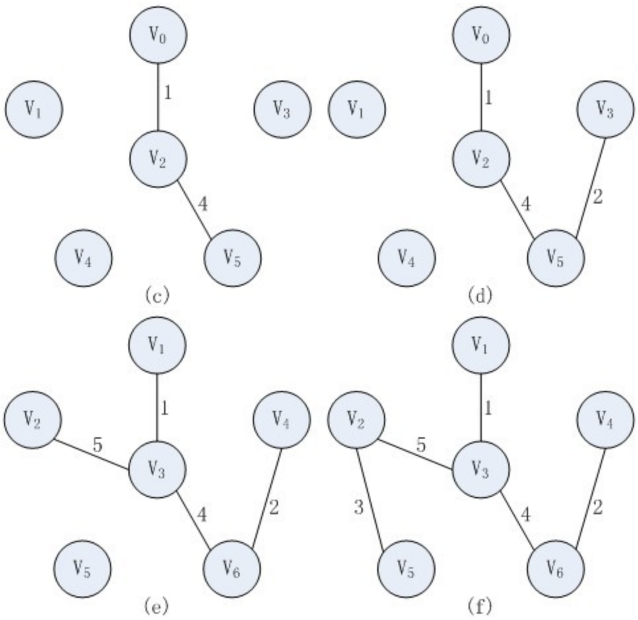


图 12-4　Prim 算法构造最小生成树的过程

表 12-1　构造最小生成树过程中辅助数组的变化

closedge \ i	1	2	3	4	5	U	U-V	k
adjvex	v_0	v_0	v_0			v_0	$\{v_1, v_2, v_3, v_4, v_5\}$	2
lowcost	6	1	5					
adjvex	v_2		v_1	v_2	v_2	$\{v_0, v_2\}$	$\{v_1, v_3, v_4, v_5\}$	5
lowcost	5	0	5	6	4			
adjvex	v_2		v_6	v_2		$\{v_0, v_2, v_5\}$	$\{v_1, v_3, v_4\}$	3
lowcost	5	0	2	6	0			
adjvex	v_2			v_2		$\{v_0, v_2, v_5, v_3\}$	$\{v_1, v_4\}$	1
lowcost	5	0	0	6	0			
adjvex				v_1		$\{v_0, v_2, v_5, v_3, v_1\}$	$\{v_4\}$	4
lowcost	0	0	0	3	0			
adjvex						$\{v_0, v_2, v_5, v_3, v_1, v_4\}$	$\{\}$	
lowcost	0	0	0	0	0			

代码

mgraph_prim1.c

```
#include <stdio.h>
#include <stdlib.h>  /* for malloc() */
#include <limits.h>  /* for INT_MAX */

/** 顶点数的最大值 */
#define MAX_VERTICES_NUM 100
/** 边的权值，对无权图，用 0 或 1 表示是否相邻；对有权图，则为权值. */
typedef int graph_weight_t;
/** 顶点信息，例如顶点名字. */
typedef char graph_vertex_t;

/**
 *@struct
 *@brief 邻接矩阵.
 */
typedef struct mgraph_t {
    int vertices_num; /* 顶点数 */
    int edges_num; /* 边数 */
    /* 顶点表，存放顶点的信息，如名字等 */
    graph_vertex_t vertices[MAX_VERTICES_NUM];
    /* 邻接矩阵，存放边的信息，如权重等 */
    graph_weight_t matrix[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
    int directed; /* 图（网）的种类，1 表示有向，0 表示无向 */
} mgraph_t;

mgraph_t g;

typedef struct closedge_t {
    int adjvex; /* 弧头，属于 U */
    graph_weight_t lowcost; /* 边 adjvex-> 本下标 的权值，INT_MIN 表示已经加入 U */
} closedge_t;

/*
 * @brief 在 V-E 集合中寻找最小的边
 * @param[in] closedge MST 中的边，起点为 adjvex，终点为本下标
 * @param[in] n closedge 数组的长度
 * @return 找到了则返回弧尾的下标，V-U 为空集则返回-1，表示终止
 */
static int min_element(const closedge_t closedge[], int n) {
    int i;
    int min_value = INT_MAX;
    int min_loc = -1;
    for (i = 0; i < n; i++)
        if (closedge[i].lowcost > INT_MIN) {
            if (min_value > closedge[i].lowcost) {
                min_value = closedge[i].lowcost;
                min_loc = i;
            }
        }
    return min_loc;
```

```
}

/**
 * @brief Prim 算法，求图的最小生成树.
 * @param[in] g 图对象的指针
 * @return MST 的边的权值之和
 */
graph_weight_t mgraph_prim(const mgraph_t *g) {
    graph_weight_t sum = 0; /* 权值之和 */
    int i, j;
    int u = 0; /* 从 0 号顶点出发 */
    const int n = g->vertices_num;
    /* closedge[n]，记录从顶点集 U 到 V-U 的边 */
    closedge_t* const closedge = (closedge_t*) malloc(n * sizeof(closedge_t));

    /* 辅助数组初始化 */
    for (i = 0; i < n; i++) if (i != u) {
        closedge[i].adjvex = u;
        closedge[i].lowcost = g->matrix[u][i];
    }
    closedge[u].lowcost = INT_MIN; /* 初始，U={u} */

    for (i = 0; i < n; i++) if (i != u) { /* 其余的 n-1 个顶点 */
        /* 求出 TE 的下一个顶点 k */
        const int k = min_element(closedge, n);
        /* 输出此边 closedge[k].adjvex --> k */
        printf("%c - %c\n", g->vertices[closedge[k].adjvex], g->vertices[k]);
        sum += g->matrix[closedge[k].adjvex][k];
        // sum += closedge[k].lowcost;  // 等价
        closedge[k].lowcost = INT_MIN;  /* 顶点 k 并入 U，表示此边加入 TE */
        /* 更新 k 的邻接点的值，不相邻为无穷大 */
        for (j = 0; j < n; j++) {
            const graph_weight_t w = g->matrix[k][j];
            if (w < closedge[j].lowcost) {
                closedge[j].adjvex = k;
                closedge[j].lowcost = w;
            }
        }
    }
    free(closedge);
    return sum;
}

/* test

输入数据:

7 11
A B 7
A D 5
B C 8
B D 9
B E 7
```

```
C E 5
D E 15
D F 6
E F 8
E G 9
F G 11

输出:

A - D : 5
D - F : 6
A - B : 7
B - E : 7
E - C : 5
E - G : 9
Total:39
*/
int main() {
    int i, j, k, m, n;
    char chx, chy;
    int cost;

    /* 读取节点和边的数目 */
    scanf("%d%d", &m, &n);
    getchar(); // 消耗回车键
    g.vertices_num = m;
    g.edges_num = n;
    for (i = 0; i < m; i++) g.vertices[i] = 'A' + i;

    /* 初始化图，所有节点间距离为无穷大 */
    for (i = 0; i < m; i++) {
        for (j = 0; j < m; j++) {
            g.matrix[i][j] = INT_MAX;
        }
    }

    /* 读取边信息 */
    for (k = 0; k < n; k++) {
        scanf("%c %c %d", &chx, &chy, &cost);
        getchar();
        i = chx - 'A';
        j = chy - 'A';
        g.matrix[i][j] = cost;
        g.matrix[j][i] = cost;
    }

    /* 求解最小生成树 */
    printf("Total:%d\n", mgraph_prim(&g));
    return 0;
}
```

mgraph_prim1.c

算法分析

假设网中有 n 个顶点，则第一个进行初始化的循环语句的频度为 n，第二个循环语句的频度为 $n-1$。其中有两个内循环：其一是在 `closedge[v].lowcost` 中求最小值，其频度为 $n-1$；其二是重新选择具有最小代价的边，其频度为 n。因此 Prim 算法的时间复杂度为 $O(n^2)$，与网中边数无关，因此适用于求边稠密的图的最小生成树。

Prim 算法的另一种实现是使用小根堆，其流程是：小根堆中存储一个端点在生成树中，另一个端点不在生成树的边，每次从小根堆的堆顶可选出权值最小的边 (u, v)，将其从堆中推出，加入生成树中。然后将新出现的所有一个端点在生成树中，一个端点不在生成树的边都插入小根堆中。下一轮迭代中，下一条满足要求的边又上升到堆顶。如此重复 $n-1$ 次，最后建立起该图的最小生成树。该算法的 C 代码实现如下。

代码

mgraph_prim2.c

```
#include <stdio.h>
#include <stdlib.h>  /* for malloc() */
#include <limits.h>  /* for INT_MAX */

/** 顶点数的最大值 */
#define MAX_VERTICES_NUM 100
/** 边的权值，对无权图，用 0 或 1 表示是否相邻；对有权图，则为权值. */
typedef int graph_weight_t;
/** 顶点信息，例如顶点名字. */
typedef char graph_vertex_t;

/**
 *@struct
 *@brief 邻接矩阵.
 */
typedef struct mgraph_t {
    int vertices_num; /* 顶点数 */
    int edges_num; /* 边数 */
    /* 顶点表，存放顶点的信息，如名字等 */
    graph_vertex_t vertices[MAX_VERTICES_NUM];
    /* 邻接矩阵，存放边的信息，如权重等 */
    graph_weight_t matrix[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
    int directed; /* 图（网）的种类，1 表示有向，0 表示无向 */
} mgraph_t;

mgraph_t g;


/**
 * @struct 边
 */
typedef struct edge_t{
    int tail;  /** 弧尾，from */
```

```
    int head;  /** 弧头, to */
    graph_weight_t cost;  /** 权值 */
}edge_t;

static int edge_cmp(const edge_t *e1, const edge_t *e2) {
    const edge_t* const e11 = (const edge_t *)e1;
    const edge_t* const e22 = (const edge_t *)e2;
    return e11->cost - e22->cost;
}

typedef edge_t heap_elem_t; // 元素的类型

/* 等价于复制粘贴，这里为了节约篇幅，使用 include，在 OJ 上提交时请用复制粘贴 */
#include "heap.c"


/**
  * @brief Prim 算法，求图的最小生成树.
  * @param[in] g 图对象的指针
  * @return MST 的边的权值之和
  */
int mgraph_prim(const mgraph_t *g){
    graph_weight_t sum = 0; /* 权值之和 */
    int u = 0; /* 从 0 号顶点出发 */
    int i, count = 1;
    edge_t ed;
    heap_t h;
    const int n = g->vertices_num;
    /* 判断顶点是否已经加入最小生成树 */
    int* U = (int *)malloc(n * sizeof(int));
    for(i = 0; i < n; i++) U[i] = 0;
    heap_init(&h, g->edges_num, edge_cmp);

    /* 开始顶点加入 U(所以 count 初始为 1) */
    U[u] = 1;
    while (count < n) {
        int v;
        for(v = 0; v < n; v++) if(!U[v]) { /* 若 v 不在生成树，(u,v) 加入堆 */
            ed.tail = u;
            ed.head = v;
            /* tail 在树内，head 不在树内 */
            ed.cost = g->matrix[u][v];
            heap_push(&h, ed);
        }
        while(!heap_empty(&h) && count < n) {
            /* 从堆中退出最小权值边，存入 ed */
            ed = heap_top(&h); heap_pop(&h);
            if(!U[ed.head]) {
                /* 输出生成树 TE 的边，即此边加入 TE */
                printf("%c - %c\n", g->vertices[ed.tail],
                        g->vertices[ed.head]);
                sum += g->matrix[ed.tail][ed.head];
                u = ed.head;
```

```
                /* u 并入到生成树的顶点集合 U */
                U[u] = 1;
                count++;
                break;
            }
        }
    }

    free(U);
    heap_uninit(&h);
    return sum;
}


// test, 输入数据和 main() 函数与 mgraph_prim1.c 相同
// int main() {
// ...
```
mgraph_prim2.c

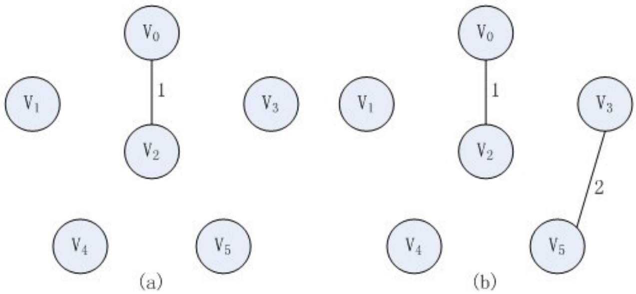
算法分析

该算法迭代次数为 $O(n)$，每次迭代将平均 e/n 条边插入最小堆中，e 条边从堆中删除，堆的插入和删除操作时间复杂度均为 $O(\log_2 e)$，则总的时间复杂度为 $O(e\log_2 e)$。

12.4.2 Kruskal 算法

假设连通网 $N = V, E$，则令最小生成树的初始状态为只有 n 个顶点而无边的非连通图 $T = (V,)$，图中每个顶点自成一个连通分量。在 E 中选择代价最小的边，若该边依附的顶点落在 T 中不同的连通分量上，则将此边加入到 T 中，否则舍去此边而选择下一条代价最小的边。依次类推，直至 T 中所有顶点都在同一连通分量上为止。

图12-5所示为 Kruskal 算法构造一棵最小生成树的过程。



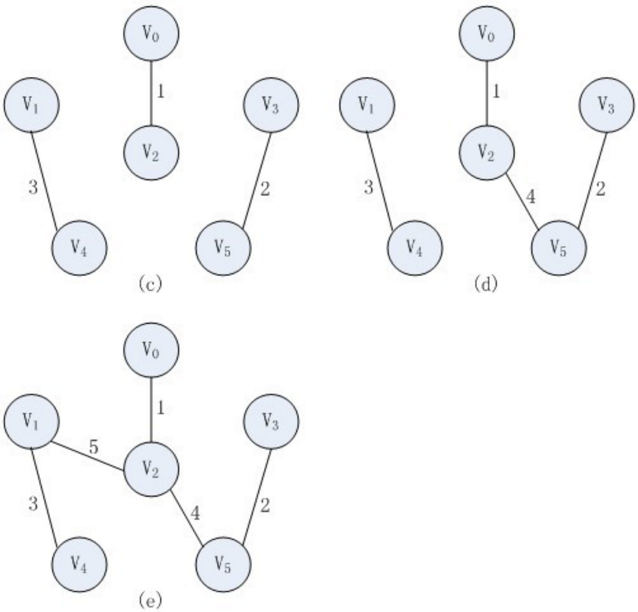


图 12-5 Kruskal 算法构造最小生成树的过程

下面是 Kruskal 算法的 C 语言实现。

代码

mgraph_kruskal.c

```
#include <stdio.h>
#include <stdlib.h>  /* for malloc() */
#include <limits.h>  /* for INT_MAX */

/************ 并查集 ************/
/**
  * @brief 初始化并查集.
  * @param[in] s 双亲表示法的数组
  * @param[in] n 数组 s 的元素个数
  * @return 无
  */
void ufs_init(int s[], int n) {
    int i;
    for(i = 0; i < n; i++) s[i] = -1;
}

/**
  * @brief Find 操作，查找包含元素 x 的树的根.
  * @param[in] s 双亲表示法的数组
  * @param[in] x 要查找的元素
  * @return 包含元素 x 的树的根
```

```
  */
int ufs_find(const int s[], int x) {
    while(s[x] >= 0) {
        x = s[x];
    }
    return x;
}

 /*
  * @brief Union 操作，求两个不相交集合的并集.
  * @param[in] s 双亲表示法的数组
  * @param[in] root1 一棵树的根
  * @param[in] root2 另一棵树的根
  * @return 无
  */
void ufs_union(int s[], int root1, int root2) {
    s[root1] += s[root2];
    s[root2] = root1;
}
/************ 并查集 ************/


/** 顶点数的最大值 */
#define MAX_VERTICES_NUM 100
/** 边的权值，对无权图，用 0 或 1 表示是否相邻；对有权图，则为权值. */
typedef int graph_weight_t;
#define INFINITY INT_MAX  /* 边权值无穷大 */
/** 顶点信息，例如顶点名字. */
typedef char graph_vertex_t;

/**
 *@struct
 *@brief 邻接矩阵.
 */
typedef struct mgraph_t {
    int vertices_num; /* 顶点数 */
    int edges_num; /* 边数 */
    /* 顶点表，存放顶点的信息，如名字等 */
    graph_vertex_t vertices[MAX_VERTICES_NUM];
    /* 邻接矩阵，存放边的信息，如权重等 */
    graph_weight_t matrix[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
    int directed; /* 图 (网) 的种类，1 表示有向，0 表示无向 */
} mgraph_t;

mgraph_t g;


/**
 * @struct 边
 */
typedef struct edge_t{
    int tail;  /** 弧尾，from */
    int head;  /** 弧头，to */
```

```
    graph_weight_t cost;  /** 权值 */
}edge_t;

static int edge_cmp(const edge_t *e1, const edge_t *e2) {
    const edge_t* const e11 = (const edge_t *)e1;
    const edge_t* const e22 = (const edge_t *)e2;
    return e11->cost - e22->cost;
}

typedef edge_t heap_elem_t; // 元素的类型

/* 等价于复制粘贴，这里为了节约篇幅，使用 include，在 OJ 上提交时请用复制粘贴 */
#include "heap.c"


/*
  * @brief Kruskal 算法，求图的最小生成树.
  * @param[in] g 图对象的指针
  * @return MST 的边的权值之和
  */
int mgraph_kruskal(const mgraph_t *g) {
    graph_weight_t sum = 0;
    edge_t ed;
    int u, v, count;
    const int n = g->vertices_num;
    heap_t h;
    int *ufs;

    ufs = (int *)malloc(n * sizeof(int));
    ufs_init(ufs, n);
    heap_init(&h, g->edges_num, edge_cmp);

    /* 把所有边插入堆中 */
    for(u = 0; u < n; u++) {
        for(v = u + 1; v < n; v++) {
            const int w = g->matrix[u][v];
            if(w < INFINITY) {
                ed.tail = u;
                ed.head = v;
                ed.cost = w;
                heap_push(&h, ed);
            }
        }
    }

    count = 0; /* 最小生成树的边数 */
    while (count < n - 1)
    {
        /* 从堆中退出最小权值边，存入 ed */
        ed = heap_top(&h);
        heap_pop(&h);
        /* 取两顶点所在集合的根 */
        u = ufs_find(ufs, ed.tail);
```

```
        v = ufs_find(ufs, ed.head);
        if(u != v) { /* 不是同一集合，说明不连通 */
            ufs_union(ufs, u, v); /* 合并，连通成一个分量 */
            /* 输出生成树 TE 的边，即此边加入 TE */
            printf("%d - %d\n", ed.tail, ed.head);
            sum += ed.cost;
            count++;
        }
    }

    free(ufs);
    heap_uninit(&h);
    return sum;
}


// test，输入数据和 main() 函数与 mgraph_prim1.c 相同
// int main() {
// ...
```
mgraph_kruskal.c

算法分析

如果采用邻接矩阵作为图的存储结构，则在建立小根堆时需要检测图的邻接矩阵，这需要 $O(n^2)$ 的时间。此外，需要将 e 条边组成初始的小根堆。如果直接从空堆开始，依次插入各边，需要 $O(e\log_2 e)$ 的时间。在构造最小生成树的过程中，需要进行 $O(e)$ 次出堆操作 `heap_remove()`、$2e$ 次并查集的 `ufs_find()` 操作以及 $n-1$ 次 `ufs_union()` 操作，计算时间分别为 $O(e\log_2 e)$、$O(\log_2 n)$ 和 $O(n)$，所以总时间为 $O(n^2+e\log_2 e)$。

如果采用邻接表作为图的存储结构，则在建立小根堆时需要检测图的邻接表，这需要 $O(n+e)$ 的时间。为建成初始的小根堆，需要 $O(e\log_2 e)$ 的时间。在构造最小生成树的过程中，需要进行 $O(e)$ 次出堆操作 `heap_remove()`、$2e$ 次并查集的 `ufs_find()` 操作以及 $n-1$ 次 `ufs_union()` 操作，计算时间分别为 $O(e\log_2 e)$、$O(e\log_2 n)$ 和 $O(n)$，所以总时间为 $O(n+e\log_2 e)$。

12.4.3 例题：POJ 1751 Highways

描述

一个名叫 Flatopia 的岛国地势非常平坦。不幸的是 Flatopia 的公共高速公路系统很差劲。Flatopia 的政府也意识到了这个问题，已经建造了许多高速公路用来连接比较重要的城镇。不过，仍然有一些城镇没有接入高速公路。因此，很有必要建造更多的高速公路，让任意两个城镇之间可以通过高速公路连接。

Flatopia 的城镇从 1 到 N 编号，城镇 i 的位置由笛卡尔坐标 (x_i, y_i) 表示。每条高速公路仅连接两个城镇。所有的高速公路都是直线，因此它们的长度就等于两个城镇之间的欧氏距离。所有的高速公路是双向的，高速公路之间可以相交，但是司机只能在公路的端点 (也即城镇) 换道。

Flatopia 政府希望能最小化建造高速公路的代价。由于 Flatopia 地势平坦，一条高速公路的代价正比于它的长度。因此，应该让高速公路的总长度最小。

输入

输入由两部分组成。第一部分描述所有的城镇，第二部分描述所有已经建造好的高速公路。

第一行包含一个整数 $N(1 \leq N \leq 750)$，表示城镇的数目。接下来的 N 行每行包含一对整数，x_i 和 y_i，由空格隔开，表示第 i 个城镇的坐标。坐标的绝对值不会超过 10000。每个城镇的坐标都不重叠。

接下来一行包含一个整数 $M(0 \leq M \leq 1000)$，表示已经存在的高速公路的数目。接下来的 M 行每行包含一对整数，给出了一对城镇编号，表示这两个城镇被一条高速公路连接起来。每两个城镇之间最多被一条高速公路连接。

输出

输出所有需要新建的高速公路。每行一个高速公路，用一对城镇编号表示。

如果不需要新建高速公路，输出为空。

样例输入

```
9
1 5
0 0
3 2
4 5
5 1
0 4
5 2
1 2
5 3
3
1 3
9 7
1 2
```

样例输出

```
1 6
3 7
4 9
5 7
8 3
```

分析

很明显，最小生成树。

题中的网络是一个完全图，任意两个城镇之间都有边，权值是两点间的距离。因此 Prim 算法比 Kruskal 算法效率更高。

对于已经存在的高速公路，令它们权值为 0，可以保证它们一定会被选中。

因为题目只需要输出新建的高速公路的两个端点，不需要输出最小生成树的长度，所以计算距离的时候不用 sqrt，也就不用 double 了。

代码

poj_1751_highways_prim.c

```c
#include <stdio.h>
#include <stdlib.h>  /* for malloc() */
#include <limits.h>  /* for INT_MAX */

/** 顶点数的最大值 */
#define MAX_VERTICES_NUM 750
/** 边的权值，对无权图，用 0 或 1 表示是否相邻；对有权图，则为权值. */
typedef int graph_weight_t;
/** 顶点信息，例如顶点名字. */
typedef int graph_vertex_t;

/**
 *@struct
 *@brief 邻接矩阵.
 */
typedef struct mgraph_t {
    int vertices_num; /* 顶点数 */
    int edges_num; /* 边数 */
    /* 顶点表，存放顶点的信息，如名字等 */
    graph_vertex_t vertices[MAX_VERTICES_NUM];
    /* 邻接矩阵，存放边的信息，如权重等 */
    graph_weight_t matrix[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
    int directed; /* 图（网）的种类，1 表示有向，0 表示无向 */
} mgraph_t;

mgraph_t g;

typedef struct closedge_t {
    int adjvex; /* 弧头，属于 U */
```

```
    graph_weight_t lowcost; /* 边 adjvex-> 本下标 的权值，INT_MIN 表示已经加入 U */
} closedge_t;

/*
 * @brief 在 V-E 集合中寻找最小的边
 * @param[in] closedge MST 中的边，起点为 adjvex，终点为本下标
 * @param[in] n closedge 数组的长度
 * @return 找到了则返回弧尾的下标，V-U 为空集则返回-1，表示终止
 */
static int min_element(const closedge_t closedge[], int n) {
    int i;
    int min_value = INT_MAX;
    int min_loc = -1;
    for (i = 0; i < n; i++)
        if (closedge[i].lowcost > INT_MIN) {
            if (min_value > closedge[i].lowcost) {
                min_value = closedge[i].lowcost;
                min_loc = i;
            }
        }
    return min_loc;
}

/**
 * @brief Prim 算法，求图的最小生成树.
 * @param[in] g 图对象的指针
 * @return MST 的边的权值之和
 */
graph_weight_t mgraph_prim(const mgraph_t *g) {
    graph_weight_t sum = 0; /* 权值之和 */
    int i, j;
    int u = 0; /* 从 0 号顶点出发 */
    const int n = g->vertices_num;
    /* closedge[n]，记录从顶点集 U 到 V-U 的边 */
    closedge_t* const closedge = (closedge_t*) malloc(n * sizeof(closedge_t));

    /* 辅助数组初始化 */
    for (i = 0; i < n; i++) if (i != u) {
        closedge[i].adjvex = u;
        closedge[i].lowcost = g->matrix[u][i];
    }
    closedge[u].lowcost = INT_MIN; /* 初始，U={u} */

    for (i = 0; i < n; i++) if (i != u) { /* 其余的 n-1 个顶点 */
        /* 求出 TE 的下一个顶点 k */
        const int k = min_element(closedge, n);
        /* 输出此边 closedge[k].adjvex --> k */
        if (g->matrix[closedge[k].adjvex][k] > 0)
            printf("%d %d\n", g->vertices[closedge[k].adjvex], g->vertices[k]);
        sum += g->matrix[closedge[k].adjvex][k];
        // sum += closedge[k].lowcost;  // 等价
        closedge[k].lowcost = INT_MIN;  /* 顶点 k 并入 U，表示此边加入 TE */
        /* 更新 k 的邻接点的值，不相邻为无穷大 */
```

```
        for (j = 0; j < n; j++) {
            const graph_weight_t w = g->matrix[k][j];
            if (w < closedge[j].lowcost) {
                closedge[j].adjvex = k;
                closedge[j].lowcost = w;
            }
        }
    }
    free(closedge);
    return sum;
}

/* 输入数据 */
int n, m, x[MAX_VERTICES_NUM], y[MAX_VERTICES_NUM];

/*
 * @brief 两点之间的距离.
 *
 * 因为题目只需要输出新建的高速公路的两个端点，不需要输出最小生成
 * 树的长度，所以计算距离的时候不用 sqrt，也就不用 double 了。
 *
 * @param[in] i 编号为 i+1 的城镇
 * @param[in] j 编号为 j+1 的城镇
 *
 * @return 欧氏距离的平方
 */
static int distance(int i,int j) {
    return (x[i]-x[j]) * (x[i]-x[j]) + (y[i]-y[j]) * (y[i]-y[j]);
}

int main() {
    int i, j;
    scanf("%d",&n);
    g.vertices_num = n;
    g.edges_num = n * (n-1) / 2;
    g.directed = 0;
    for(i = 0; i < n; i++) g.vertices[i] = i+1;

    for(i = 0; i < n; i++) scanf("%d %d", &x[i], &y[i]);
    for(i = 0; i < n; i++)
        for(j = i; j < n; j++)
            g.matrix[i][j] = g.matrix[j][i] = distance(i,j);

    scanf("%d",&m);
    for(i = 0; i < m; i++) {
        int a, b;
        scanf("%d %d",&a, &b);
        g.matrix[a-1][b-1] = g.matrix[b-1][a-1] = 0;
    }

    mgraph_prim(&g);

    return 0;
```

```
}
```

poj_1751_highways_prim.c

相关的题目

与本题相同的题目:

- TODO

与本题相似的题目:

- POJ 2485 Highways, http://poj.org/problem?id=2485
- POJ 1861 Network, http://poj.org/problem?id=1861
- POJ 2395 Out of Hay, http://poj.org/problem?id=2395
- POJ 2377 Bad Cowtractors, http://poj.org/problem?id=2377
- POJ 2421 Constructing Roads, http://poj.org/problem?id=2421
- POJ 1679 The Unique MST, http://poj.org/problem?id=1679
- POJ 1258 Agri-Net, http://poj.org/problem?id=1258
- POJ 1251 Jungle Roads, http://poj.org/problem?id=1251
- POJ 3625 Building Roads, http://poj.org/problem?id=3625
- POJ 1789 Truck History, http://poj.org/problem?id=1789

12.5 最短路径

12.5.1 单源最短路径——Dijkstra 算法

假设 S 为已求得最短路径的点的集合，则可证明：下一条最短路径（设其终点为 x）或者是弧 (v,x)，或者是中间只经过 S 中的顶点而最后到达顶点 x 的路径。

Dijkstra 算法流程如下:

1. S 为已找到从 v 出发的最短路径的终点的集合，它的初始状态为空集。`dist[i]` 存放的是 v 到 v_i 的最短路径长度，根据前面所述性质，`dist[i]=min{dist[i],weight(v,`v_i`)}`。`path[i]` 存放的是最短路径上指向 v_i 的弧尾顶点。那么从 v 出发到图上其余 v_i 的最短路径长度的初值为:

$$\text{dist}[i] = \text{weight}(v, v_i), v_i \in V$$

2. 选择 v_j，使得

$$\mathrm{dist}[j] = \min\{\mathrm{dist}[j], \mathrm{weight}(v, v_j) | v_j \in V - S\}$$

将 v_j 加入到 S，

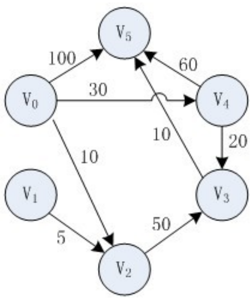
$$S = S \cup v_j$$

3. 修改从 v 出发到集合 $V - S$ 上任一顶点 v_k 可达的最短路径长度，并记录下这条边。

```
if(dist[j] + weight(j, k) < dist[k]) {
    dist[k] = dist[j] + weight(j, k);
    path[k] = j; /* 修改到 k 的最短路径 */
}
```

4. 重复 2，3 共 $n - 1$ 次。

例如，对图12-6所示的有向图及其邻接矩阵运行运行 Dijkstra 算法，



$$\begin{bmatrix} \infty & \infty & 10 & \infty & 30 & 100 \\ \infty & \infty & 5 & \infty & \infty & \infty \\ \infty & \infty & \infty & 50 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 10 \\ \infty & \infty & \infty & 20 & \infty & 60 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

图 12-6　有向图及其邻接矩阵

运算过程中 v_0 到其余个顶点的最短路近，`dist[]` 向量的变化情况如表12-2所示（从一列到下一列只需要更新新加入点的邻接点)。

表 12-2　Dijkstra 算法过程中 dist[] 向量的变化情况

终点	i=1	i=2	i=3	i=4	i=5
v_1	∞	∞	∞	∞	∞
v_2	**10** $(\mathbf{v_0}, \mathbf{v_2})$				
v_3	∞	60 (v_0, v_2, v_3)	**50** $(\mathbf{v_0}, \mathbf{v_4}, \mathbf{v_5})$		
v_4	30 (v_0, v_4)	**30** $(\mathbf{v_0}, \mathbf{v_4})$			
v_5	100 (v_0, v_5)	100 (v_0, v_5)	90 (v_0, v_4, v_5)	**60** $(\mathbf{v_0}, \mathbf{v_4}, \mathbf{v_3}, \mathbf{v_5})$	
v_j	v_2	v_4	v_3	v_5	
S	(v_0, v_2)	(v_0, v_2, v_4)	(v_0, v_2, v_3, v_4)	$(v_0, v_2, v_3, v_4, v_5)$	

Dijkstra 算法的 C 语言实现如下。

代码

mgraph_dijkstra.c

```
#include <stdio.h>
#include <stdlib.h>  /* for malloc() */
#include <limits.h>  /* for INT_MAX */

/** 顶点数的最大值 */
#define MAX_VERTICES_NUM 100
/** 边的权值，对无权图，用 0 或 1 表示是否相邻；对有权图，则为权值. */
typedef int graph_weight_t;
#define GRAPH_INFINITY INT_MAX
/** 顶点信息，例如顶点名字. */
typedef char graph_vertex_t;

/**
 *@struct
 *@brief 邻接矩阵.
 */
typedef struct mgraph_t {
    int vertices_num; /* 顶点数 */
    int edges_num; /* 边数 */
    /* 顶点表，存放顶点的信息，如名字等 */
    graph_vertex_t vertices[MAX_VERTICES_NUM];
    /* 邻接矩阵，存放边的信息，如权重等 */
    graph_weight_t matrix[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
    int directed; /* 图（网）的种类，1 表示有向，0 表示无向 */
} mgraph_t;

mgraph_t g;
```

```
/** path[i] 存放的是最短路径上指向 vi 的弧尾顶点 */
int path[MAX_VERTICES_NUM];
/** dist[i] 存放的是 v 到 vi 的最短路径长度 */
int dist[MAX_VERTICES_NUM];


/*
  * @brief Dijkstra 算法求单源最短路径.
  * @param[in] g 图对象的指针
  * @param[in] v 起点
  * @param[out] dist dist[i] 存放的是 v 到 vi 的最短路径长度
  * @param[out] path path[i] 存放的是最短路径上指向 vi 的弧尾顶点
  * @return 无
  */
void mgraph_dijkstra(const mgraph_t *g, int v, int dist[], int path[]) {
    int i, j;
    const int n = g->vertices_num;

    // 初始化 S 集合
    int *S = (int*)malloc( n * sizeof(int));
    for(i = 0; i < n; i++) S[i] = 0;
    S[v] = 1; /* 初始化, 顶点 u 加入 S */

    // 初始化 dist 和 path
    for(i = 0; i < n; i++) if (i !=v) {
        dist[i] = g->matrix[v][i];
        if(dist[i] < GRAPH_INFINITY) {
            path[i] = v;
        }  else {
            path[i] = -1; /* 没有顶点指向 i */
        }
    }
    dist[v] = 0;
    path[v] = -1;

    for(i = 0; i < n; i++) if (i !=v) {
        /* 选不在 S 中的最短路径顶点 u */
        int u;
        graph_weight_t min = GRAPH_INFINITY;
        for(j = 0; j < n; j++) {
            if(!S[j] && dist[j] < min) {
                u = j;
                min = dist[j];
            }
        }
        S[u] = 1;
        for(j = 0; j < n; j++) {
            const graph_weight_t w = g->matrix[u][j];
            /* 顶点 j 未就加入 S, 且经过 u 到 j 可缩短路径 */
            if(!S[j] && w < GRAPH_INFINITY &&
                dist[u] + w < dist[j]) {
                dist[j] = dist[u] + w;
```

```
                path[j] = u; /* 修改到 j 的最短路径 */
            }
        }
    }
    free(S);
}

/*
 * @brief 打印从起点到 v 的最短路径
 * @param[in] v 终点
 * @param[in] path Dijkstra 计算好的 path
 * @return 无
 */
static void print_path_r(int v, const int path[]) {
    if (path[v] == -1) {
        printf("%c", g.vertices[v]);
    } else {
        print_path_r(path[v], path);
        printf("->%c", g.vertices[v]);
    }
}

/**
 * @brief 打印 u 到其他所有点的最短路径
 * @param[in] path Dijkstra 计算好的 path
 * @param[in] n path 的长度
 * @return 无
 */
void print_path(const int path[], int n) {
    int i;
    for (i = 0; i < n; i++) if (path[i] != -1) {
        print_path_r(i, path);
        printf("\n");
    }
}


/* test

输入数据:

6 8
A C 10
A E 30
A F 100
B C 5
C D 50
D 5 10
E D 20
E F 60

输出:
```

```
A->C
A->E->D
A->E
A->E->F
*/
int main() {
    int i, j, k;
    char chx, chy;
    graph_weight_t cost;

    /* 读取节点和边的数目 */
    scanf("%d%d", &(g.vertices_num), &(g.edges_num));
    g.directed = 1;
    for (i = 0; i < g.vertices_num; i++) g.vertices[i] = 'A' + i;

    /* 初始化图，所有节点间距离为无穷大 */
    for (i = 0; i < g.vertices_num; i++) {
        for (j = 0; j < g.vertices_num; j++) {
            g.matrix[i][j] = GRAPH_INFINITY;
        }
    }

    /* 读取边信息 */
    getchar(); // 消耗回车键
    for (k = 0; k < g.edges_num; k++) {
        scanf("%c %c %d", &chx, &chy, &cost);
        getchar();
        i = chx - 'A';
        j = chy - 'A';
        g.matrix[i][j] = cost;
    }

    /* 求 V0 到其他所有顶点的最短路径 */
    mgraph_dijkstra(&g, 0, dist, path);
    print_path(path, g.vertices_num);
    return 0;
}
```

mgraph_dijkstra.c

算法分析

该算法包含了两个并列的 for 循环，第一个 for 循环做辅助数组的初始化工作，计算时间为 $O(n)$，第二个 for 循环是二重嵌套循环，进行最短路径的求解工作，由于对图中几乎每个顶点都要做计算，每个顶点的又要对集合 S 内的顶点进行检测，对集合 $V-S$ 内中的顶点进行修改，所以运算时间复杂度为 $O(n^2)$。算法总的时间复杂度为 $O(n^2)$。

12.5.2　每点最短路径——Floyd 算法

Floyd 算法的基本思想是：假设求从定点 v_i 到 v_j 的最短路径。初始时，若 v_i 与 v_j 之间存在边，则最短路径长度为此边的权值；若不存在边，则最短路径长度为无穷大。以后逐步在路径中加入顶点 $k(k=0,1,...,n-1)$ 作为中间顶点，如果加入中间顶点后，得到的路径比原来的路径长度减少了，则以新路径代替原路径。

首先比较 (v_i,v_j) 和 (v_i,v_0,v_j) 的路径长度，取较短者为从 v_i 到 v_j 的中间顶点的序号不大于 0 的最短路径。如果 (v_i,v_0,v_j) 较短，则取 (v_i,v_0,v_j) 作为最短路径。假如在路径上再增加一个顶点 v_1，也就是说，如果 $(v_i,...,v_1)$ 和 $(v_1,...,v_j)$ 分别是当前找到的中间定点的序号不大于 0 的最短路径，那么 $(vi,...,v1,...,vj)$ 就有可能是从 v_i 到 v_j 的中间顶点的序号不大于 1 的最短路径，将它和已经得到的从 v_i 到 v_j 的中间顶点的序号不大于 0 的最短路径相比较，选出较短者作为从 v_i 到 v_j 的中间顶点的序号不大于 1 的最短路径。再增加一个顶点 v_2，继续进行试探，依此类推。一般的，若 $(v_i,...,v_k)$ 和 $(v_k,...,v_j)$ 分别是从 v_i 到 v_k 和从 v_k 到 v_j 的中间定点的序号不大于 $k-1$ 的最短路径，则将 $(v_i,...,v_k,...,v_j)$ 和已经得到的从 v_i 到 v_j 的中间顶点的序号不大于 $k-1$ 的最短路径相比，较短者便是从 v_i 到 v_j 的中间顶点的序号不大于 k 的最短路径。这样，在经过 n 次比较后，最后求得的必是从 v_i 到 v_j 的最短路径。

现定义一个 n 阶方阵序列，

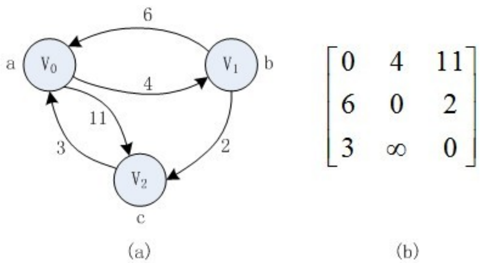
$$D^{(-1)},D^{(0)},D^{(1)},...,,D^{(k)},...,,D^{(n-1)}$$

其中，

$$\begin{aligned}
D^{(-1)}[i][j] &= \text{g->matrix}[i][j],\\
D^{(k)}[i][j] &= \min\left\{D^{(k-1)}[i][j],D^{(k-1)}[i][k]+D^{(k-1)}[k][j]\right\},0\le k\le n-1
\end{aligned}$$

上述公式中，$D^{(k)}[i][j]$ 是从 v_i 到 v_j 的中间顶点的序号不大于 k 的最短路径的长度；$D^{(n-1)}[i][j]$ 是从 v_i 到 v_j 的最短路径的长度。

例如，对图12-7所示的有向图及其邻接矩阵运行 Floyd 算法，



$$\begin{bmatrix}0 & 4 & 11\\ 6 & 0 & 2\\ 3 & \infty & 0\end{bmatrix}$$

图 12-7　有向图及其邻接矩阵

运算过程中矩阵 D 的变化如表12-3所示。

表 12-3　Floyd 算法过程中方阵和最短路径的变化

D	$\mathbf{D}^{(0)}$			$\mathbf{D}^{(1)}$			$\mathbf{D}^{(2)}$			$\mathbf{D}^{(3)}$		
	0	1	2	0	1	2	0	1	2	0	1	2
0	0	4	11	0	4	11	0	4	6	0	4	6
1	6	0	2	6	0	2	6	0	2	5	0	2
2	3	∞	0	3	7	0	3	7	0	3	7	0
P	$\mathbf{P}^{(0)}$			$\mathbf{P}^{(1)}$			$\mathbf{P}^{(2)}$			$\mathbf{P}^{(3)}$		
	0	1	2	0	1	2	0	1	2	0	1	2
0		A B	A C		AB	A C		AB	AB C		AB	AB C
1	B A		B C	B A		B C	B A		BC	BC A		BC
2	C A			C A	CA B		C A	CA B		CA	CA B	

Floyd 算法的 C 语言实现如下。

代码

mgraph_floyd.c

```
#include <stdio.h>
#include <stdlib.h>  /* for malloc() */
#include <limits.h>  /* for INT_MAX */

/** 顶点数的最大值 */
#define MAX_VERTICES_NUM 100
/** 边的权值，对无权图，用 0 或 1 表示是否相邻；对有权图，则为权值. */
typedef int graph_weight_t;
#define GRAPH_INFINITY (INT_MAX / 2)   /* 确保加法不溢出 */
/** 顶点信息，例如顶点名字. */
typedef char graph_vertex_t;

/**
 *@struct
 *@brief 邻接矩阵.
 */
typedef struct mgraph_t {
    int vertices_num; /* 顶点数 */
    int edges_num; /* 边数 */
    /* 顶点表，存放顶点的信息，如名字等 */
    graph_vertex_t vertices[MAX_VERTICES_NUM];
    /* 邻接矩阵，存放边的信息，如权重等 */
```

```
    graph_weight_t matrix[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
    int directed; /* 图 (网) 的种类, 1 表示有向, 0 表示无向 */
} mgraph_t;


mgraph_t g;
/** dist[i][j] 是顶点 i 和 j 之间最短路径长度 */
graph_weight_t dist[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
/** path[i][j] 是最短路径上 i 和 j 之间的顶点 */
int path[MAX_VERTICES_NUM][MAX_VERTICES_NUM];


/*
  * @brief Floyd 算法求每点之间最短路径.
  * @param[in] g 图对象的指针
  * @param[out] dist dist[i][j] 是顶点 i 和 j 之间最短路径长度
  * @param[out] path path[i][j] 是最短路径上 i 和 j 之间的顶点
  * @return 无
  */
void mgraph_floyd(const mgraph_t *g,
       int dist[][MAX_VERTICES_NUM],
       int path[][MAX_VERTICES_NUM]) {
    int i, j, k;
    const int n = g->vertices_num;

    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            if(i != j) {
                dist[i][j] = g->matrix[i][j];
                path[i][j] = i;
            } else {
                dist[i][j] = 0;
                path[i][j] = -1;
            }
        }
    }
    for(k = 0; k < n; k++) {
        for(i = 0; i < n; i++) {
            for(j = 0; j < n; j++) {
                /* i 到 j 的路径上加入顶点 k 可以缩短路径长度 */
                if(dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    path[i][j] = k;
                }
            }
        }
    }
}

/*
 * @brief 打印从 u 到 v 的最短路径
 * @param[in] u 起点
 * @param[in] v 终点
```

```
 * @param[in] path Floyd 计算好的 path
 * @return 无
 */
static void print_path_r(int u, int v, const int path[][MAX_VERTICES_NUM]) {
    if (path[u][v] == -1) {
        printf("%c", g.vertices[u]);
    } else {
        print_path_r(u, path[u][v], path);
        printf("->%c", g.vertices[v]);
    }
}

/**
 * @brief 打印 u 到其他所有点的最短路径
 * @param[in] path Dijkstra 计算好的 path
 * @param[in] n path 的长度
 * @return 无
 */
void print_path(const mgraph_t *g, const int path[][MAX_VERTICES_NUM]) {
    int i, j;
    for (i = 0; i < g->vertices_num; i++) {
        for (j = 0; j < g->vertices_num; j++) {
            if (i != j) {
                print_path_r(i, j, path);
                printf("\n");
            }
        }
        printf("\n");
    }
}


/* test

输入数据:

3 5
A B 4
A C 11
B A 6
B C 2
C A 3

输出:

A->B
A->B->C

B->C->A
B->C

C->A
C->A->B
```

```
*/
int main() {
    int i, j, k;
    char chx, chy;
    graph_weight_t cost;

    /* 读取节点和边的数目 */
    scanf("%d%d", &(g.vertices_num), &(g.edges_num ));
    g.directed = 1;
    for (i = 0; i < g.vertices_num; i++) g.vertices[i] = 'A' + i;

    /* 初始化图，所有节点间距离为无穷大 */
    for (i = 0; i < g.vertices_num; i++) {
        for (j = 0; j < g.vertices_num; j++) {
            g.matrix[i][j] = GRAPH_INFINITY;
        }
    }

    /* 读取边信息 */
    getchar(); // 消耗回车键
    for (k = 0; k < g.edges_num ; k++) {
        scanf("%c %c %d", &chx, &chy, &cost);
        getchar();
        i = chx - 'A';
        j = chy - 'A';
        g.matrix[i][j] = cost;
    }

    /* 求两两之间的最短路径 */
    mgraph_floyd(&g, dist, path);
    print_path(&g, path);
    return 0;
}
```

mgraph_floyd.c

算法分析

该算法中有两个并列的 for 循环，第一个循环是个二重循环，用于初始化方阵 D；第二个循环是个三重循环，逐步生成 $D^{(0)}, D^{(1)}, ..., D^{(n-1)}$。所以算法总的时间复杂度为 $O(n^3)$。

Dijkstra 算法权值不能为负，Floyd 权值可以为负，但环路之和不能为负。

12.5.3 例题：HDU 2544 最短路

描述

在每年的校赛里，所有进入决赛的同学都会获得一件很漂亮的 t-shirt。但是每当我们的工作人员把上百件的衣服从商店运回到赛场的时候，却是非常累的！所以现在他们想要

寻找最短的从商店到赛场的路线，你可以帮助他们吗？

输入

输入包括多组数据。每组数据第一行是两个整数 $N,M(N \leq 100, M \leq 10000)$，$N$ 表示成都的大街上有几个路口，标号为 1 的路口是商店所在地，标号为 N 的路口是赛场所在地，M 则表示在成都有几条路。$N=M=0$ 表示输入结束。接下来 M 行，每行包括 3 个整数 $A,B,C(1 \leq A,B \leq N, 1 \leq C \leq 1000)$, 表示在路口 A 与路口 B 之间有一条路，我们的工作人员需要 C 分钟的时间走过这条路。输入保证至少存在 1 条商店到赛场的路线。

输出

对于每组输入，输出一行，表示工作人员从商店走到赛场的最短时间

样例输入

```
2 1
1 2 3
3 3
1 2 5
2 3 5
3 1 2
0 0
```

样例输出

```
3
2
```

分析

单源最短路径，用 Dijkstra 算法，将第 §12.5.1节中的代码稍加修改即可。

注意，街道是双向的，所以给边赋值时要对称赋值。

代码

hdu_2544.c

```
/* http://acm.hdu.edu.cn/showproblem.php?pid=2544 */
#include <stdio.h>
#include <stdlib.h>  /* for malloc() */
#include <limits.h>  /* for INT_MAX */

/** 顶点数的最大值 */
#define MAX_VERTICES_NUM 100
/** 边的权值, 对无权图, 用 0 或 1 表示是否相邻; 对有权图, 则为权值. */
```

```c
typedef int graph_weight_t;
#define GRAPH_INFINITY INT_MAX
/** 顶点信息，例如顶点名字. */
typedef char graph_vertex_t;

/**
 *@struct
 *@brief 邻接矩阵.
 */
typedef struct mgraph_t {
    int vertices_num; /* 顶点数 */
    int edges_num; /* 边数 */
    /* 顶点表，存放顶点的信息，如名字等 */
    graph_vertex_t vertices[MAX_VERTICES_NUM];
    /* 邻接矩阵，存放边的信息，如权重等 */
    graph_weight_t matrix[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
    int directed; /* 图（网）的种类，1 表示有向，0 表示无向 */
} mgraph_t;

mgraph_t g;

/** path[i] 存放的是最短路径上指向 vi 的弧尾顶点 */
int path[MAX_VERTICES_NUM];
/** dist[i] 存放的是 v 到 vi 的最短路径长度 */
int dist[MAX_VERTICES_NUM];


/*
  * @brief Dijkstra 算法求单源最短路径.
  * @param[in] g 图对象的指针
  * @param[in] v 起点
  * @param[out] dist dist[i] 存放的是 v 到 vi 的最短路径长度
  * @param[out] path path[i] 存放的是最短路径上指向 vi 的弧尾顶点
  * @return 无
  */
void mgraph_dijkstra(const mgraph_t *g, int v, int dist[], int path[]) {
    int i, j;
    const int n = g->vertices_num;

    // 初始化 S 集合
    int *S = (int*)malloc( n * sizeof(int));
    for(i = 0; i < n; i++) S[i] = 0;
    S[v] = 1; /* 初始化，顶点 u 加入 S */

    // 初始化 dist 和 path
    for(i = 0; i < n; i++) if (i !=v) {
        dist[i] = g->matrix[v][i];
        if(dist[i] < GRAPH_INFINITY) {
            path[i] = v;
        }  else {
            path[i] = -1; /* 没有顶点指向 i */
        }
    }
```

```
    dist[v] = 0;
    path[v] = -1;

    for(i = 0; i < n; i++) if (i !=v) {
        /* 选不在 S 中的最短路径顶点 u */
        int u;
        graph_weight_t min = GRAPH_INFINITY;
        for(j = 0; j < n; j++) {
            if(!S[j] && dist[j] < min) {
                u = j;
                min = dist[j];
            }
        }
        S[u] = 1;
        for(j = 0; j < n; j++) {
            const graph_weight_t w = g->matrix[u][j];
            /* 顶点 j 未就加入 S, 且经过 u 到 j 可缩短路径 */
            if(!S[j] && w < GRAPH_INFINITY &&
                dist[u] + w < dist[j]) {
                dist[j] = dist[u] + w;
                path[j] = u; /* 修改到 j 的最短路径 */
            }
        }
    }
    free(S);
}

int main() {
    int i, j, k;
    int x, y;
    graph_weight_t cost;

    /* 读取节点和边的数目 */
    while(scanf("%d%d", &(g.vertices_num), &(g.edges_num)) > 0)
        if (g.vertices_num > 0 && g.edges_num > 0){
        g.directed = 1;
        for (i = 0; i < g.vertices_num; i++) g.vertices[i] = '1' + i;

        /* 初始化图, 所有节点间距离为无穷大 */
        for (i = 0; i < g.vertices_num; i++) {
            for (j = 0; j < g.vertices_num; j++) {
                g.matrix[i][j] = GRAPH_INFINITY;
            }
        }

        /* 读取边信息 */
        for (k = 0; k < g.edges_num; k++) {
            scanf("%d %d %d", &x, &y, &cost);
            i = x - 1;
            j = y - 1;
            g.matrix[i][j] = cost;
            g.matrix[j][i] = cost;  // 注意, 街道是双向的
        }
```

```
        /* 求 商店（即路口 1） 到其他所有顶点的最短路径 */
        mgraph_dijkstra(&g, 0, dist, path);
        /* 打印商店（即路口 1）到赛场（即路口 N）的最短路径长度 */
        printf("%d\n", dist[g.vertices_num-1]);
    }
    return 0;
}
```

hdu_2544.c

相关的题目

与本题相同的题目:

- HDU 2544 最短路, http://acm.hdu.edu.cn/showproblem.php?pid=2544

与本题相似的题目:

- POJ 2253 Frogger, http://poj.org/problem?id=2253
- POJ 3268 Silver Cow Party, http://poj.org/problem?id=3268
- POJ 1797 Heavy Transportation, http://poj.org/problem?id=1797
- POJ 1847 Tram, http://poj.org/problem?id=1847

12.5.4 例题：POJ 1125 Stockbroker Grapevine

描述

Stockbrokers are known to overreact to rumours. You have been contracted to develop a method of spreading disinformation amongst the stockbrokers to give your employer the tactical edge in the stock market. For maximum effect, you have to spread the rumours in the fastest possible way.

Unfortunately for you, stockbrokers only trust information coming from their ”Trusted sources” This means you have to take into account the structure of their contacts when starting a rumour. It takes a certain amount of time for a specific stockbroker to pass the rumour on to each of his colleagues. Your task will be to write a program that tells you which stockbroker to choose as your starting point for the rumour, as well as the time it will take for the rumour to spread throughout the stockbroker community. This duration is measured as the time needed for the last person to receive the information.

输入

Your program will input data for different sets of stockbrokers. Each set starts with a line with the number of stockbrokers. Following this is a line for each stockbroker which contains the number of people who they have contact with, who these people are, and the time taken for them to pass the message to each person. The format of each stockbroker line is as follows: The line starts with the number of contacts (n), followed by n pairs of integers, one pair for each contact. Each pair lists first a number referring to the contact (e.g. a '1' means person number one in the set), followed by the time in minutes taken to pass a message to that person. There are no special punctuation symbols or spacing rules.

Each person is numbered 1 through to the number of stockbrokers. The time taken to pass the message on will be between 1 and 10 minutes (inclusive), and the number of contacts will range between 0 and one less than the number of stockbrokers. The number of stockbrokers will range from 1 to 100. The input is terminated by a set of stockbrokers containing 0 (zero) people.

输出

For each set of data, your program must output a single line containing the person who results in the fastest message transmission, and how long before the last person will receive any given message after you give it to this person, measured in integer minutes.

It is possible that your program will receive a network of connections that excludes some persons, i.e. some people may be unreachable. If your program detects such a broken network, simply output the message "disjoint". Note that the time taken to pass the message from person A to person B is not necessarily the same as the time taken to pass it from B to A, if such transmission is possible at all.

样例输入

```
3
2 2 4 3 5
2 1 2 3 6
2 1 2 2 2
5
3 4 4 2 8 5 3
1 5 8
4 1 6 4 10 2 7 5 2
0
2 2 5 1 5
7
2 2 6 7 1
0
```

```
3 1 5 2 8 4 7
0
2 2 9 4 10
2 3 8 4 7
2 5 8 6 3
7
2 2 6 7 8
0
3 1 5 2 8 4 7
0
2 2 9 4 10
2 3 8 4 7
2 5 8 6 3
0
```

样例输出

```
3 2
3 10
1 12
7 17
```

分析

代码

poj_1125.c

```c
/* http://poj.org/problem?id=1125 */
#include <stdio.h>
#include <stdlib.h>  /* for malloc() */
#include <limits.h>  /* for INT_MAX */

/** 顶点数的最大值 */
#define MAX_VERTICES_NUM 100
/** 边的权值，对无权图，用 0 或 1 表示是否相邻；对有权图，则为权值. */
typedef int graph_weight_t;
#define GRAPH_INFINITY (INT_MAX / 2)   /* 确保加法不溢出 */
/** 顶点信息，例如顶点名字. */
typedef char graph_vertex_t;

/**
 *@struct
 *@brief 邻接矩阵.
 */
typedef struct mgraph_t {
    int vertices_num; /* 顶点数 */
    int edges_num; /* 边数 */
    /* 顶点表，存放顶点的信息，如名字等 */
    graph_vertex_t vertices[MAX_VERTICES_NUM];
    /* 邻接矩阵，存放边的信息，如权重等 */
    graph_weight_t matrix[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
    int directed; /* 图（网）的种类，1 表示有向，0 表示无向 */
```

```
} mgraph_t;


mgraph_t g;
/** dist[i][j] 是顶点 i 和 j 之间最短路径长度 */
graph_weight_t dist[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
/** path[i][j] 是最短路径上 i 和 j 之间的顶点 */
int path[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
/** 存放每个点的最短路径的最长者 */
graph_weight_t max_len[MAX_VERTICES_NUM];

/**
  * @brief Floyd 算法求每点之间最短路径.
  * @param[in] g 图对象的指针
  * @param[out] dist dist[i][j] 是顶点 i 和 j 之间最短路径长度
  * @param[out] path path[i][j] 是最短路径上 i 和 j 之间的顶点
  * @return 无
  */
void mgraph_floyd(const mgraph_t *g,
       int dist[][MAX_VERTICES_NUM],
       int path[][MAX_VERTICES_NUM]) {
    int i, j, k;
    const int n = g->vertices_num;

    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            if(i != j) {
                dist[i][j] = g->matrix[i][j];
                path[i][j] = i;
            } else {
                dist[i][j] = 0;
                path[i][j] = -1;
            }
        }
    }
    for(k = 0; k < n; k++) {
        for(i = 0; i < n; i++) {
            for(j = 0; j < n; j++) {
                /* i 到 j 的路径上加入顶点 k 可以缩短路径长度 */
                if(dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    path[i][j] = k;
                }
            }
        }
    }
}

static int max_element(const graph_weight_t w[], int begin, int end) {
    int i;
    int max_value = INT_MIN;
    int max_pos = -1;
    for (i = begin; i < end; i++) {
```

```
        if (max_value < w[i]) {
            max_value = w[i];
            max_pos = i;
        }
    }
    return max_pos;
}

static int min_element(const graph_weight_t w[], int begin, int end) {
    int i;
    int min_value = INT_MAX;
    int min_pos = -1;
    for (i = begin; i < end; i++) {
        if (min_value > w[i]) {
            min_value = w[i];
            min_pos = i;
        }
    }
    return min_pos;
}

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

    /* 读取顶点数 */
    while (scanf("%d", &(g.vertices_num)) > 0 && g.vertices_num > 0) {
        g.edges_num = 0;
        g.directed = 1;
        for (i = 0; i < g.vertices_num; i++) g.vertices[i] = '1' + i;

        /* 初始化图，所有节点间距离为无穷大 */
        for (i = 0; i < g.vertices_num; i++) {
            for (j = 0; j < g.vertices_num; j++) {
                g.matrix[i][j] = GRAPH_INFINITY;
            }
        }

        /* 读取边信息 */
        for (i = 0; i < g.vertices_num; i++) {
            int m;
            graph_weight_t cost;
            scanf("%d", &m);
            g.edges_num += m;
            while (m--) {
                scanf("%d %d", &j, &cost);
                --j;
                g.matrix[i][j] = cost;
            }
        }

        /* 求两两之间的最短路径 */
        mgraph_floyd(&g, dist, path);
```

```
        /* 找最短路径的最长着 */
        for (i = 0; i < g.vertices_num; i++) {
            max_len[i] = dist[i][max_element(dist[i], 0, g.vertices_num)];
        }
        /* 找 max_len 的最小者   */
        min_pos = min_element(max_len, 0, g.vertices_num);

        if (max_len[min_pos] == GRAPH_INFINITY) {
            printf("disjoint\n");
        } else {
            printf("%d %d\n", min_pos+1, max_len[min_pos]);
        }
    }
    return 0;
}
```

poj_1125.c

相关的题目

与本题相同的题目:

- POJ 1125 Stockbroker Grapevine, http://poj.org/problem?id=1125

与本题相似的题目:

- POJ 3615 Cow Hurdles, http://poj.org/problem?id=3615
- POJ 3660 Cow Contest, http://poj.org/problem?id=3660
- POJ 2502 Subway, http://poj.org/problem?id=2502
- HDU 3631 Shortest Path, http://acm.hdu.edu.cn/showproblem.php?pid=3631

12.6 拓扑排序

由某个集合上的一个偏序得到该集合上的一个全序，这个操作称为**拓扑排序**。

拓扑序列的特点是：若有向边 $< V_i, V_j >$ 是途中的弧，则在序列中顶点 V_i 必须排在顶点 V_j 之前。

如果用有向图表示一个工程，顶点表示活动，用有向边 $< v_i, v_j >$ 表示活动必须先于活动进行。这种有向图叫做顶点表示活动的网络 (Activity On Vertext Network)，简称 **AOV 网络**。

检测 AOV 网络是否存在环的方法是对 AOV 网络构造其顶点的拓扑有序序列。拓扑排序的基本步骤是:

1. 在有向图中选一个没有前驱的顶点且输出之;

2. 从图中删除该顶点和所有以它为尾的弧线。

重复以上两步，直至全部顶点输出，或当前图中不存在无前驱的顶点为止（这种情况说明图中存在环)。

拓扑排序的 C 语言实现如下。

代码

mgraph_topo_sort.c

```
#include <stdio.h>
#include <stdlib.h>  /* for malloc() */
#include <limits.h>  /* for INT_MAX */

/** 顶点数的最大值 */
#define MAX_VERTICES_NUM 100
/** 边的权值，对无权图，用 0 或 1 表示是否相邻；对有权图，则为权值. */
typedef int graph_weight_t;
#define GRAPH_INFINITY INT_MAX
/** 顶点信息，例如顶点名字. */
typedef char graph_vertex_t;

/**
 *@struct
 *@brief 邻接矩阵.
 */
typedef struct mgraph_t {
    int vertices_num; /* 顶点数 */
    int edges_num; /* 边数 */
    /* 顶点表，存放顶点的信息，如名字等 */
    graph_vertex_t vertices[MAX_VERTICES_NUM];
    /* 邻接矩阵，存放边的信息，如权重等 */
    graph_weight_t matrix[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
    int directed; /* 图 (网) 的种类，1 表示有向，0 表示无向 */
} mgraph_t;

mgraph_t g;
/** 拓扑排序的结果. */
int topological[MAX_VERTICES_NUM];

// 等价于复制粘贴，这里为了节约篇幅，使用 include，在 OJ 上提交时请用复制粘贴
#include "stack.c"

/*
  * @brief 拓扑排序.
  * @param[in] g 图对象的指针
  * @param[out] topological 保存拓扑排序的结果
  * @return 无环返回 1，有环返回 0
  */
int mgraph_topo_sort(const mgraph_t *g, int topological[]) {
    int i, j, u;
    int count = 0; /* 拓扑序列的元素个数 */
```

```
    const int n = g->vertices_num;
    /* in_degree[i] 是顶点 i 的入度 */
    int *in_degree = (int*)malloc(n * sizeof(int));
    stack_t s;

    memset(in_degree, 0, n * sizeof(int));
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            if (g->matrix[i][j] < GRAPH_INFINITY)
                in_degree[j]++;
        }
    }

    stack_init(&s, n * sizeof(int));
    for(i = 0; i < n; i ++) {
        if(in_degree[i] == 0)
            stack_push(&s, i);
    }

    while(!stack_empty(&s)) {
        u = stack_top(&s); stack_pop(&s);
        topological[count++] = u;
        for (i = 0; i < n; i++) if (g->matrix[u][i] < GRAPH_INFINITY) {
            if(--in_degree[i] == 0) stack_push(&s, i);
        }
    }

    free(in_degree);
    if(count != n) { /* 有环 */
        return 0;
    } else { /* 无环 */
        return 1;
    }
}


/* test

输入数据:

6 8
A C 10
A E 30
A F 100
B C 5
C D 50
D 5 10
E D 20
E F 60

输出:

B A E F C D
```

```c
*/
int main() {
    int i, j, k;
    char chx, chy;
    graph_weight_t cost;

    /* 读取节点和边的数目 */
    scanf("%d%d", &(g.vertices_num), &(g.edges_num));
    getchar(); // 消耗回车键
    g.directed = 1;
    for (i = 0; i < g.vertices_num; i++) g.vertices[i] = 'A' + i;

    /* 初始化图，所有节点间距离为无穷大 */
    for (i = 0; i < g.vertices_num; i++) {
        for (j = 0; j < g.vertices_num; j++) {
            g.matrix[i][j] = GRAPH_INFINITY;
        }
    }

    /* 读取边信息 */
    for (k = 0; k < g.edges_num; k++) {
        scanf("%c %c %d", &chx, &chy, &cost);
        getchar();
        i = chx - 'A';
        j = chy - 'A';
        g.matrix[i][j] = cost;
    }

    /* 拓扑排序 */
    mgraph_topo_sort(&g, topological);
    for (i = 0; i < g.vertices_num; i++) {
        printf("%c ", g.vertices[topological[i]]);
    }
    return 0;
}
```

mgraph_topo_sort.c

算法分析

对有 n 个顶点和 e 条边的 AOV 网络而言，求各顶点的入度所需时间为 $O(e)$，建立零入度顶点栈所需时间为 $O(n)$；在拓扑排序过程中，若有向图无环，每个顶进一次栈出一次栈，顶点入度减 1 的操作共执行了 e 次。所以总的时间复杂度为 $O(n+e)$。

当有向图中无环时，也可以利用深度优先搜索进行拓扑排序。因为图中无环，深度优先遍历不会死循环。进行深度优先遍历时，最先退出 DFS 函数的顶点即为出度为零的顶点，是拓扑有序序列的最后一个顶点。由此，按退出 DFS 函数的先后次序记录下来的顶点序列即为逆向的拓扑有序序列。

12.6.1 例题：POJ 1094 Sorting It All Out

描述

An ascending sorted sequence of distinct values is one in which some form of a less-than operator is used to order the elements from smallest to largest. For example, the sorted sequence A, B, C, D implies that $A < B, B < C$ and $C < D$. in this problem, we will give you a set of relations of the form $A < B$ and ask you to determine whether a sorted order has been specified or not.

输入

Input consists of multiple problem instances. Each instance starts with a line containing two positive integers n and m. the first value indicated the number of objects to sort, where $2 \leq n \leq 26$. The objects to be sorted will be the first n characters of the uppercase alphabet. The second value m indicates the number of relations of the form $A < B$ which will be given in this problem instance. Next will be m lines, each containing one such relation consisting of three characters: an uppercase letter, the character "<" and a second uppercase letter. No letter will be outside the range of the first n letters of the alphabet. Values of $n = m = 0$ indicate end of input.

输出

For each problem instance, output consists of one line. This line should be one of the following three:

Sorted sequence determined after xxx relations: $yyy...y$.

Sorted sequence cannot be determined.

Inconsistency found after xxx relations.

where xxx is the number of relations processed at the time either a sorted sequence is determined or an inconsistency is found, whichever comes first, and $yyy...y$ is the sorted, ascending sequence.

样例输入

```
4 6
A<B
A<C
B<C
C<D
```

```
B<D
A<B
3 2
A<B
B<A
26 1
A<Z
6 6
A<F
B<D
C<E
F<D
D<E
E<F
0 0
```

样例输出

```
Sorted sequence determined after 4 relations: ABCD.
Inconsistency found after 2 relations.
Sorted sequence cannot be determined.
Inconsistency found after 6 relations.
```

分析

根据题目的要求，我们要每输入一次就要进行一次拓扑排序 `topological_sort()`，这样才能做到不成功（即发现有环）时，能知道是哪步不成功，并且给出输出。

还有要注意的就是如果我们可以提前判断结果了，但后面还有输入没完成，那么我们必须继续完成输入，不然剩下的输入会影响下一次 case 的输入。

代码

poj_1094.c

```c
#include <stdio.h>
#include <stdlib.h>  /* for malloc() */
#include <limits.h>  /* for INT_MAX */

/** 顶点数的最大值 */
#define MAX_VERTICES_NUM 26
/** 边的权值，对无权图，用 0 或 1 表示是否相邻；对有权图，则为权值. */
typedef int graph_weight_t;
#define GRAPH_INFINITY INT_MAX
/** 顶点信息，例如顶点名字. */
typedef char graph_vertex_t;

/**
 *@struct
 *@brief 邻接矩阵.
 */
```

```c
typedef struct mgraph_t {
    int vertices_num; /* 顶点数 */
    int edges_num; /* 边数 */
    /* 顶点表，存放顶点的信息，如名字等 */
    graph_vertex_t vertices[MAX_VERTICES_NUM];
    /* 邻接矩阵，存放边的信息，如权重等 */
    graph_weight_t matrix[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
    int directed; /* 图（网）的种类，1 表示有向，0 表示无向 */
} mgraph_t;

mgraph_t g;
/** 拓扑排序的结果. */
int topological[MAX_VERTICES_NUM];

// 等价于复制粘贴，这里为了节约篇幅，使用 include，在 OJ 上提交时请用复制粘贴
#include "stack.c"

/**
  * @brief 拓扑排序.
  * @param[in] g 图对象的指针
  * @param[out] topological 保存拓扑排序的结果
  * @return 无环返回 1，有环返回 0
  */
int mgraph_topological_sort(const mgraph_t *g, int topological[]) {
    int i, j, u;
    int count = 0; /* 拓扑序列的元素个数 */
    int insufficient = 0;  /* 条件不足 */
    const int n = g->vertices_num;
    /* in_degree[i] 是顶点 i 的入度 */
    int *in_degree = (int*)malloc(n * sizeof(int));
    stack_t s;

    memset(in_degree, 0, n * sizeof(int));
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            if (g->matrix[i][j] < GRAPH_INFINITY)
                in_degree[j]++;
        }
    }

    stack_init(&s, n * sizeof(int));
    for(i = 0; i < n; i ++) {
        if(in_degree[i] == 0) {
            stack_push(&s, i);
        }
    }

    while(!stack_empty(&s)) {
        // 栈内应该始终只有一个元素
        if (stack_size(&s) > 1) insufficient = 1;
        // 删除顶点 u
        u = stack_top(&s); stack_pop(&s);
        topological[count++] = u;
```

```
        --in_degree[u];  // 变成 -1，表示已经输出
        // 更新入度
        for (i = 0; i < n; i++) if (g->matrix[u][i] < GRAPH_INFINITY) {
            --in_degree[i];
        }
        // 选择入度为 0 的顶点
        for (i = 0; i < n; i++) if (g->matrix[u][i] < GRAPH_INFINITY) {
            if (in_degree[i] == 0) stack_push(&s, i);
        }
    }

    free(in_degree);
    if(count < n) { /* 有环 */
        return 0;
    } else { /* 无环 */
        if (insufficient) {  /* 有孤立点，说明条件不足 */
            return -1;
        } else {
            return 1;
        }
    }
}


int main() {
    int i, j, k, m;  // m 不一定是边的数目，因为输入边可能有重复

    /* 读取节点和边的数目 */
    while (scanf("%d %d", &(g.vertices_num), &m)
            && g.vertices_num > 0 && m > 0) {
        char s[4];
        int ok;  // 是否有环，0 表示有环
        int finished = 0;  // 排序完成，结束，发现有环，可以提前结束

        g.directed = 1;
        for (i = 0; i < g.vertices_num; i++) g.vertices[i] = 'A' + i;

        /* 初始化图，所有节点间距离为无穷大 */
        for (i = 0; i < g.vertices_num; i++) {
            for (j = 0; j < g.vertices_num; j++) {
                g.matrix[i][j] = GRAPH_INFINITY;
            }
        }

        /* 读取边信息 */
        for (k = 0; k < m; k++) {
            scanf("%s", s);
            i = s[0] - 'A';
            j = s[2] - 'A';
            g.matrix[i][j] = 1;

            if (finished) continue;    // 完成，则 continue，消耗输入
```

```
            /* 拓扑排序 */
            ok = mgraph_topological_sort(&g, topological);

            if (ok == 0) {  // 有环存在
                printf("Inconsistency found after %d relations.\n", k + 1);
                finished = 1;  // 提前结束，记住要继续消耗输入
            }
            if (ok == 1 && k) {
                printf("Sorted sequence determined after %d relations: ", k+1);
                for (i = 0; i < g.vertices_num; i++) {
                    printf("%c", g.vertices[topological[i]]);
                }
                printf(".\n");
                finished = 1;
            }
            // ok==-1, continue
        }
        if (finished == 0) {
            printf("Sorted sequence cannot be determined.\n");
        }
    }
    return 0;
}
```

poj_1094.c

相关的题目

与本题相同的题目：

- POJ 1094 Sorting It All Out, http://poj.org/problem?id=1094

与本题相似的题目：

- POJ 3267 The Cow Lexicon, http://poj.org/problem?id=3267
- POJ 3687 Labeling Balls, http://poj.org/problem?id=3687

12.7 关键路径

用有向边上的权值表示活动的持续时间，用顶点表示时间，这样的有向图叫做边表示的活动网络 (Activity On Edge Network)，简称 **AOE 网络**。

路径最长的路径叫做**关键路径** (Critical Path)。假设开始点为 v_1，从 v_1 到 v_i 的最长路径长度叫做事件 v_i 的最早发生时间。这个事件决定了所有以 v_i 为尾的弧所表示的活动的最早开始时间。我们用 $e(i)$ 表示活动 a_i 的最早开始时间。还可以定义一个活动的最迟开始时间 $l(i)$，这是在不推迟整个工程完成的前提下，活动 a_i 最迟必须开始进行的时间。两者之差 $l(i) - e(i)$ 意味着完成活动 a_i 的时间余量。我们把 $l(i) = e(i)$ 的活动叫做关键活动。

设活动 a_i 由弧 $<j,k>$ 表示，为了求得活动的 $e(i)$ 和 $l(i)$，首先应求得事件的最早发生时间 $ve(j)$ 和最迟发生时间 $vl(j)$，其持续时间记为 $dut(<j,k>)$，则有如下关系

$$\begin{aligned} e(i) &= ve(j) \\ l(i) &= vl(k) - dut(<j,k>) \end{aligned}$$

求 $ve(j)$ 和 $vl(k)$ 需分两步进行:

1. 从 $ve(0)=0$ 开始向前递推

$$ve(j) = \max\{ve(i) + dut(<i,j>)\}, <i,j> \in T$$

其中 T 是所有以顶点 j 为弧头的边的集合。

2. 从 $vl(n-1) = ve(n-1)$ 起向后递推

$$vl(j) = \min\{vl(k) - dut(<j,k>)\}, <j,k> \in S$$

其中 S 是所有以顶点 j 为弧尾的边的集合。

例如，对图12-8(a) 所示 AOE 网络的计算过程如表12-4所示，可见 a_2、a_5 和 a_7 为关键活动，组成一条从起点到终点的关键路径，如图12-8(b) 所示。

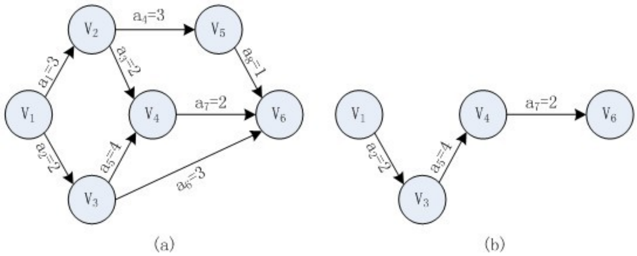


图 12-8　有向图及其邻接矩阵

表 12-4　图12-8(a) 所示 AOE 网络的关键路径的计算过程

顶点	ve		vl		活动	e	l	l-e
v_1	0		0		a_1	0	1	1
v_2	3		4		a_2	0	0	0
v_3	2		2		a_3	3	4	1
v_4	6	↓	6	↑	a_4	3	4	1
v_5	6		7		a_5	2	2	0
v_6	8		8		a_6	2	5	3
					a_7	6	6	0
					a_8	6	7	1

邻接矩阵上的关键路径的 C 语言实现如下。

代码

mgraph_critical_path.c

```
#include <stdio.h>
#include <stdlib.h>  /* for malloc() */
#include <limits.h>  /* for INT_MAX */

/** 顶点数的最大值 */
#define MAX_VERTICES_NUM 100
/** 边的权值，对无权图，用 0 或 1 表示是否相邻；对有权图，则为权值. */
typedef int graph_weight_t;
#define GRAPH_INFINITY INT_MAX
/** 顶点信息，例如顶点名字. */
typedef char graph_vertex_t;

/**
 *@struct
 *@brief 邻接矩阵.
 */
typedef struct mgraph_t {
    int vertices_num; /* 顶点数 */
    int edges_num; /* 边数 */
    /* 顶点表，存放顶点的信息，如名字等 */
    graph_vertex_t vertices[MAX_VERTICES_NUM];
    /* 邻接矩阵，存放边的信息，如权重等 */
    graph_weight_t matrix[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
    int directed; /* 图（网）的种类，1 表示有向，0 表示无向 */
} mgraph_t;

mgraph_t g;
/** 拓扑排序的结果. */
int topological[MAX_VERTICES_NUM];
/** 关键路径，其余顶点为-1. */
int path[MAX_VERTICES_NUM];


/* 等价于复制粘贴，这里为了节约篇幅，使用 include，在 OJ 上提交时请用复制粘贴 */
#include "stack.c"


/*
  * @brief 按照拓扑排序的顺序，计算所有顶点的最早发生时间 ve.
  * @param[in] g 图对象的指针
  * @param[out] topological 保存拓扑排序的结果
  * @param[out] ve 所有事件的最早发生时间
  * @return 无环返回 1，有环返回 0
  */
static int mgraph_toposort_ve(const mgraph_t *g, int topological[],
        graph_weight_t ve[]) {
    int i, j, u;
```

```
    int count = 0; /* 拓扑序列的元素个数 */
    const int n = g->vertices_num;
    /* in_degree[i] 是顶点 i 的入度 */
    int *in_degree = (int*)malloc(n * sizeof(int));
    stack_t s;

    memset(in_degree, 0, n * sizeof(int));
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            if (g->matrix[i][j] < GRAPH_INFINITY)
                in_degree[j]++;
        }
    }

    stack_init(&s, n * sizeof(int));
    for(i = 0; i < n; i ++) {
        if(in_degree[i] == 0) {
            stack_push(&s, i);
        }
    }

    memset(ve, 0, n * sizeof(graph_weight_t));

    while(!stack_empty(&s)) {
        // 删除顶点 u
        u = stack_top(&s); stack_pop(&s);
        topological[count++] = u;
        --in_degree[u];  // 变成 -1，表示已经输出
        // 更新入度
        for (i = 0; i < n; i++) if (g->matrix[u][i] < GRAPH_INFINITY) {
            --in_degree[i];
        }
        // 更新邻接点的 ve
        for (i = 0; i < n; i++) if (g->matrix[u][i] < GRAPH_INFINITY) {
            if (ve[i] < ve[u] + g->matrix[u][i])
                ve[i] = ve[u] + g->matrix[u][i];
        }
        // 选择入度为 0 的顶点
        for (i = 0; i < n; i++) if (g->matrix[u][i] < GRAPH_INFINITY) {
            if (in_degree[i] == 0) stack_push(&s, i);
        }
    }

    free(in_degree);
    if(count < n) { /* 有环 */
        return 0;
    } else { /* 无环 */
        return 1;
    }
}

/**
  * @brief 求关键路径，第一个顶点为起点，最后一个顶点为终点.
```

```
 * @param[in] g 图对象的指针
 * @param[out] ve 所有事件的最早发生时间
 * @param[inout] path 关键路径
 * @return 无环返回关键路径的顶点个数，有环返回 0
 */
int mgraph_critical_path(const mgraph_t *g, int path[MAX_VERTICES_NUM]) {
    int i, j;
    int count = 0;    // 关键路径的顶点个数
    graph_weight_t *ve = (graph_weight_t*) malloc(
            g->vertices_num * sizeof(graph_weight_t));
    graph_weight_t *vl = (graph_weight_t*) malloc(
            g->vertices_num * sizeof(graph_weight_t));

    if (!mgraph_toposort_ve(g, topological, ve)) return 0;  // 有环

    for (i = 0; i < MAX_VERTICES_NUM; i++) path[i] = -1;
    // 初始化 vl 为最大
    for (i = 0; i < g->vertices_num; i++) vl[i] = ve[g->vertices_num-1];

    // 逆序计算 vl
    for (i = g->vertices_num-1; i >=0; i--) {
        int k = topological[i];
        for (j = 0; j < g->vertices_num; j++) {
            if (g->matrix[j][k] < GRAPH_INFINITY) {
                if (vl[j] > vl[k] - g->matrix[j][k])
                    vl[j] = vl[k] - g->matrix[j][k];
            }
        }
    }
    for (i = 0; i < g->vertices_num; i++) {
        for (j = 0; j < g->vertices_num; j++) {
            int e = ve[i];
            int l = vl[j] - g->matrix[i][j];
            if (e == l) {
                if (i == 0) {
                    path[count++] = i;
                    path[count++] = j;
                } else {
                    path[count++] = j;
                }
            }
        }
    }

    free(ve);
    free(vl);
    return count;
}


/* test

输入数据:
```

```
6 8
A B 3
A C 2
C D 4
B D 2
C F 3
B E 3
E F 1
D F 2

输出: A C D F

*/
int main() {
    int i, j, k;
    char chx, chy;
    graph_weight_t cost;
    int count;

    /* 读取节点和边的数目 */
    scanf("%d%d", &(g.vertices_num), &(g.edges_num));
    g.directed = 1;
    for (i = 0; i < g.vertices_num; i++) g.vertices[i] = 'A' + i;

    /* 初始化图，所有节点间距离为无穷大 */
    for (i = 0; i < g.vertices_num; i++) {
        for (j = 0; j < g.vertices_num; j++) {
            g.matrix[i][j] = GRAPH_INFINITY;
        }
    }

    /* 读取边信息 */
    getchar(); // 消耗回车键
    for (k = 0; k < g.edges_num; k++) {
        scanf("%c %c %d", &chx, &chy, &cost);
        getchar();
        i = chx - 'A';
        j = chy - 'A';
        g.matrix[i][j] = cost;
    }

    /* 拓扑排序 */
    count = mgraph_critical_path(&g, path);
    for (i = 0; i < count; i++) {
        printf("%c ", g.vertices[path[i]]);
    }
    return 0;
}
```

mgraph_critical_path.c

算法分析

一次正向，复杂度为 $O(n^2)$，一次逆向，复杂度为 $O(n^2)$，因此，该算法的复杂度为 $O(n^2)$。

12.8 A* 算法

12.8.1 八数码问题

题目见 §12.2.2。

代码

eight_digits_astar.c

```
/**
 简单解释几个要点，便于理解代码.
 1. 怎么判断是否有解? 只要计算出的逆序个数总和为奇数，该数据必然无解
 2. 如何判断某一状态是否到过? 本题存在一种完美哈希方案，即用康托展开。
    详见 http://128kj.iteye.com/blog/1699795
    2.1. 将一个状态视为数字 0-8 的一个排列，将此排列转化为序数，作为此状
 态的 HASH 值。0 表示空格. 转化算法此处不再赘述。

    2.2. 排列转化为序数，用序数作为 hash 值
    例，1 2 3 这三个数字的全排列，按字典序，依次为
 123 -- 0
 132 -- 1
 213 -- 2
 231 -- 3
 312 -- 4
 321 -- 5
 其中，左侧为排列，右侧为其序数。

 3. 使用数据结构 堆 加速挑选最优值。

 4. 函数 g 的计算，此状态在搜索树中已经走过的路径的节点数.

 5. 估价函数 h ，采用曼哈顿距离，见代码 calcH 函数。曼哈顿距离的定义是,
 假设有两个点 (x1,y1),(x2,y2)，则曼哈顿距离 L1=|x1-x2| + |y1-y2|
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// 3x3 的棋盘，状态最多有 9! 种
// 8 位变进制数（空格）能表示 0 到 (9!-1) 内的所有自然数，恰好有 9! 个,
// 与状态一一对应，因此可以把状态一一映射到一个 8 位变进制数
#define     MAX         362880

#define DIGITS 9 // 棋盘中数字的个数，也是变进制数需要的位数
```

```
#define     MATRIX_EDGE 3      // 棋盘边长

#define     MOD         10     // 按十取模

typedef struct {
    int state; // 状态
    int parent;     // 父状态
    int flag;   // -1 表示已经展开过了 closed，0 表示死节点，1 表示还未展开，open
    int g, h, f; // 三个评估函数
    char choice;  // 左右上下四个方向移动，见全局常量 DI DJ DC
} state_t;

state_t states[MAX];  // 全局的一条状态变化路径

int  startIndex, goalIndex; // 开始状态，目标状态对应的 hash 值

// 目标状态
const int goal = 123456780;
// 每个数字在棋盘中的位置，例如 0，在 (1,1)=4 这个位置上
const int goal_pos[DIGITS] = {8,0,1,2,3,4,5,6,7};
const int space_number = 0; // 空格对应着数字 0

// 上下左右四个方向
const int DI[] = {-1, 1, 0, 0};
const int DJ[] = {0, 0, -1, 1};
const char DC[] = { 'u', 'd', 'l', 'r' };

// 9 位变进制数，每个位数的单位，0!~8!
const int fac[] = {40320, 5040, 720, 120, 24, 6, 2, 1, 1};

/**
 * @brief 计算状态的 hash 值，这里用康托展开，是完美哈希.
 *
 * @param[in] s 当前状态
 * @return 序数，作为 hash 值
 */
int hash(int s) {
    int i, j;
    int d[DIGITS];
    int key = 0;

    for(i = DIGITS - 1; i >=0; i--) {
        d[i] = s % MOD;
        s /= MOD;
    }

    for (i = 0; i < DIGITS; i++) {
        int c = 0; // 逆序数
        for (j = i + 1; j < DIGITS; j++) {
            if(d[j] < d[i]) {
                c++;
            }
        }
```

```
        key += c * fac[i];
    }

    return key;
}


/**
 * 估价函数 h。
 * @param s 状态
 * @return 预估代价
 */
int calcH(int s) {
    int i;
    int h = 0;

    for (i = DIGITS - 1; i >= 0; --i) {
        const int p = s % 10;
        s /= 10;
        // 曼哈顿距离
        h += abs(i / MATRIX_EDGE - goal_pos[p] / MATRIX_EDGE) +
            abs(i % MATRIX_EDGE - goal_pos[p] % MATRIX_EDGE);
    }
    return h;
}

/**
 * @brief 输入.
 * @return  成功返回数字，失败返回 0
 * @remark 《算法竞赛入门经典》第 131 页 7.5.3 节，是用 0 表示空格，
 * POJ 1077 是用'x' 表示空格，前者简化了一点，POJ 1077 还需要把'x' 映射成 0
 */
int input() {
    int ch, i;
    int start = 0;
    for (i = 0; i < DIGITS; ++i) {
        do {
            ch = getchar();
        } while ((ch != EOF) && ((ch < '1') || (ch > '8')) && (ch != 'x'));
        if (ch == EOF) return 0;
        if (ch == 'x') start = start * MOD + space_number; // x 映射成数字 0
        else             start = start * MOD + ch - '0';
    }
    return start;
}

/**
 * 计算一个排列的逆序数，0 除外.
 */
int inversion_count(int permutation) {
    int i, j;
    int d[DIGITS];
    int c = 0; // 逆序数
```

```
    for(i = DIGITS - 1; i >=0; i--) {
        d[i] = permutation % MOD;
        permutation /= MOD;
    }

    for (i = 1; i < DIGITS; i++)  if (d[i] != space_number) {
        for (j = 0; j < i; j++) {
            if(d[j] != space_number) {
                if (d[j] > d[i]) {
                    c++;
                }
            }
        }
    }
    return c;
}

/**
 * 判断是否无解.
 *
 * 求出除 0 之外所有数字的逆序数之和，也就是每个数字后面比它小的数字的个数的和，
 * 称为这个状态的逆序。若两个状态的逆序奇偶性相同，则可相互到达，否则不可相互到达。
 * 由于原始状态的逆序数为 0（偶数），因此逆序数为偶数的目标状态有解。
 *
 * @param s 目标状态
 * @return 1 表示无解，0 表示有解
 */
int not_solvable(const int s) {
    return inversion_count(s) % 2;
}

// 存放 next() 的输出结果
char choice[4]; // 四个防线
int nextIndex[4]; // 接下来的四个状态

/*
 * @brief 向四个方向扩展
 * @param[in] s 状态
 * @return 无
 */
void next(int s) {
    int i, j, k;
    int p[MATRIX_EDGE][MATRIX_EDGE]; // 一个状态对应的矩阵
    int i0, j0;  // 空格位置

    for (i = MATRIX_EDGE - 1; i >= 0; i--) {
        for (j = MATRIX_EDGE - 1; j >= 0; j--) {
            p[i][j] = s % MOD;
            s /= MOD;
            if (p[i][j] == space_number) {
                i0 = i;
                j0 = j;
```

```
            }
        }
    }
    // 向四个方向探索
    for (k = 0; k < 4; ++k) {
        const int sx = i0 + DI[k]; // 空格的新位置 (sx, sy)
        const int sy = j0 + DJ[k];
        if ((sx >= 0) && (sx < 3) && (sy >= 0) && (sy < 3)) {
            int key;
            p[i0][j0] = p[sx][sy];
            p[sx][sy] = space_number;
            // 移动空格后，计算新的状态
            s = 0;
            for (i = 0; i < MATRIX_EDGE; i++)
                for (j = 0; j < MATRIX_EDGE; j++)
                    s = s * MOD + p[i][j];
            p[sx][sy] = p[i0][j0]; // 将矩阵还原，(i0, j0) 可以不管

            key = nextIndex[k] = hash(s);
            choice[k] = DC[k];
            if (states[key].state == 0) { // 该状态还没有出现过
                states[key].state = s;
                states[key].h = calcH(s);
            }
        } else {// 越界了
            nextIndex[k] = -1;
        }
    }
}

/* 等价于复制粘贴，这里为了节约篇幅，使用 include，在 OJ 上提交时请用复制粘贴 */
#include "heap.c"
heap_t heap;
int heapIndex[MAX + 4]; // 状态 x 在 heap 中的下标

/**
 * @brief A* 搜索
 * @param[in] start 初始状态
 * @return 如果无解，返回 0，如果有解返回 1
 */
int astar(const int start) {
    int i, j, k, ng, nf;
    if (not_solvable(start)) return 0;

    startIndex = hash(start);
    goalIndex  = hash(goal);
    if (start == goal) return 1;

    memset(states, 0, sizeof(states));
    states[startIndex].state = start;
    states[startIndex].flag    = 1;
    states[startIndex].g       = 0;
    states[startIndex].h       = states[startIndex].f = calcH(start);
```

```
    heap_push(&heap, startIndex);
    while(!heap_empty(&heap)) {
        i = heap_top(&heap); heap_pop(&heap);
        if (i == goalIndex) return 1; // 找到目标，返回

        states[i].flag = -1;
        ng = states[i].g + 1;
        next(states[i].state);
        for (k = 0; k < 4; ++k) {
            j = nextIndex[k];
            if (j < 0) continue;
            nf = ng + states[j].h;
            if ((states[j].flag == 0) || ((states[j].flag == 1) &&
                (nf < states[j].f))) {
                states[j].parent = i;
                states[j].choice = choice[k];
                states[j].g    = ng;
                states[j].f    = nf;
                if (states[j].flag > 0) {
                    heap_sift_up(&heap, heapIndex[j]);
                    heap_sift_down(&heap, heapIndex[j]);
                } else {
                    heap_push(&heap, j);
                    states[j].flag = 1;
                }
            }
        }
    }
    return 0;
}

/* 等价于复制粘贴，这里为了节约篇幅，使用 include，在 OJ 上提交时请用复制粘贴 */
#include "stack.c"
/**
 * @brief 打印移动序列.
 * @return 无
 */
void output() {
    int i;
    stack_t stack;
    stack_init(&stack, MAX);

    for (i = goalIndex; i != startIndex; i = states[i].parent) {
        stack_push(&stack, states[i].choice);
    }
    while(!stack_empty(&stack)) {
        printf("%c", stack_top(&stack));
        stack_pop(&stack);
    }

    printf("\n");
    stack_uninit(&stack);
```

```
}

/**
 * @brief 打印棋盘的每次变化.
 * @return 无
 */
void output1() {
    int i;
    int d[DIGITS];
    stack_t stack;
    stack_init(&stack, MAX);

    for (i = goalIndex; i != startIndex; i = states[i].parent) {
        stack_push(&stack, states[i].state);
    }
    stack_push(&stack, states[startIndex].state);

    while(!stack_empty(&stack)) {
        stack_elem_t tmp = stack_top(&stack);
        stack_pop(&stack);
        for(i = DIGITS - 1; i >=0; i--) {
            d[i] = tmp % MOD;
            tmp /= MOD;
        }
        for(i = 0; i < DIGITS; i++) {
            if((i + 1) % MATRIX_EDGE == 0) {
                printf("%d\n", d[i]);
            } else {
                printf("%d ", d[i]);
            }
        }
        printf("\n");
    }
    stack_uninit(&stack);
}

int main() {
    const int start = input();
    heap_init(&heap, MAX + 4, cmp_int);
    if (start > 0) {
        if (astar(start)) {
            output();
        } else {
            printf("no solution\n");
        }
    }
    heap_uninit(&heap);
    return 0;
}
```

eight_digits_astar.c

第 13 章
数学方法与常见模型

数学对于算法很重要。没有好的数学基础，很难在算法上达到一定高度。本章介绍算法竞赛中常用的数学方法和模型。

13.1 数论

13.1.1 欧几里德算法

求最大公约数 (greates common divisor) 有很多方法，最经典的方法是欧几里德算法 (Euclidean algorithm[①])，又称辗转相除法。

gcd.c

```
/**
 * @brief 求最大公约数，欧几里德算法，也即辗转相除法
 *
 * @param[in] a a, a > b > 0
 * @param[in] b b
 * @return a 和 b 的最大公约数
 */
int gcd(int a, int b) {
    if (b == 0) return a;
    return gcd(b, a % b);
}

/**
 * @brief 求最大公约数，欧几里德算法，迭代版本
 *
 * @param[in] a a, a > b > 0
 * @param[in] b b
 * @return a 和 b 的最大公约数
 */
int gcd1(int a, int b) {
    while (b != 0) {
        int tmp = b;
        b = a % b;
        a = tmp;
    }
```

[①] http://en.wikipedia.org/wiki/Euclid_algorithm

```
    return a;
}

/**
 * @brief 求最大公约数，欧几里德算法，迭代版本，基于减法
 *
 * @param[in] a a, a > b > 0
 * @param[in] b b
 * @return a 和 b 的最大公约数
 */
int gcd2(int a, int b) {
    while (a != b) {
        if (a > b) {
            a -= b;
        } else {
            b -= a;
        }
    }
    return a;
}
```

gcd.c

求出了最大公约数，可以利用它来求最小公倍数 (least common multiple), $\mathrm{lcm}(a,b) = a \times b/\mathrm{gcd}(a,b)$。

例题

- wikioi 1212 最大公约数，http://www.wikioi.com/problem/1212/

13.1.2 扩展欧几里德算法

定理: 对于不完全为 0 的非负整数 a,b，必然存在整数对 x,y ，使得 $gcd(a,b) = ax+by$。

这里 x 和 y 不一定是正数。扩展欧几里德算法 (Extended Euclidean algorithm[①]) 就是用来求 x 和 y 的。

ex_gcd.c

```
/**
 * @brief 扩展欧几里德算法
 * @param[in] a a, a > b > 0
 * @param[in] b b
 * @param[out] d d=gcd(a,b)
 * @param[out] x
 * @param[out] y
 * @return 无
 */
void ex_gcd(int a, int b, int *d, int *x, int *y) {
    if(b == 0) {
```

① http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm

```
        *d = a; *x = 1; *y = 0;
    } else {
        ex_gcd(b, a % b, d, y, x); *y -= (*x)*(a/b);
    }
}
```

ex_gcd.c

扩展欧几里德算法的应用主要有以下三个:

- 求解不定方程;
- 求解模线性方程(线性同余方程);
- 求解模的逆元;

13.1.3 素数判定

素数判定，又称素数测试 (Primality test[①])，即给定一个正整数 n，判断它是否是素数。

暴力枚举法

从 2 到 n，依次作为除数，让 n 除以它们，只要有一个能整除 n，则 n 不是素数。

```
/**
 * @brief 判断正整数 n 是否是素数
 * @param[in] n 正整数
 * @return 是, 返回 1, 否, 返回 0
 */
int is_prime(int n) {
    int i;
    if (n < 2) return 0;
    for (i = 2; i < n; i++) {
        if (n % i == 0) return 0;
    }
    return 1;
}
```

可以稍微做一点改进，从 1 到 $\sqrt{n}$，依次作为除数。

```
/**
 * @brief 判断正整数 n 是否是素数, 上界改为 sqrt(n)
 * @param[in] n 正整数
 * @return 是, 返回 1, 否, 返回 0
 */
int is_prime(int n) {
    int i;
    if (n < 2) return 0;
    const int upper = sqrt(n);

    for (i = 2; i <= upper; i++) {
```

[①] http://en.wikipedia.org/wiki/Primality_test

```
        if (n % i == 0) return 0;
    }
    return 1;
}

/**
 * @brief 判断正整数 n 是否是素数，上界改为 sqrt(n)，但不使用 sqrt() 函数
 * @param[in] n 正整数
 * @return 是，返回 1，否，返回 0
 */
int is_prime1(int n) {
    int i;
    if (n < 2) return 0;
    for (i = 2; i*i <= n; i++) {
        if (n % i == 0) return 0;
    }
    return 1;
}
```

Eratosthenes 筛法

更高效的素数判定方法应该是预先计算出一张素数表，当判断一个数是否是素数时，直接查表即可。

怎样计算？用 Eratosthenes 筛法 (Sieve of Eratosthenes[①])

给出要筛数值的范围 n，找出 $\sqrt{n}$ 以内的素数 $p_1, p_2, \ldots, p_k$。先用 2 去筛，即把 2 留下，把 2 的倍数剔除掉；再用下一个质数，也就是 3 筛，把 3 留下，把 3 的倍数剔除掉；接下去用下一个质数 5 筛，把 5 留下，把 5 的倍数剔除掉；不断重复下去……。

```
/******** Eratosthenes 筛法 ********/
#define MAXN 30000
/** prime_table[i]==1 表示 i 是素数，等于 0 则不是素数 */
int prime_table[MAXN+1];

void compute_prime_table() {
    int i, j;
    const int upper = sqrt(MAXN);

    for (i = 2; i <= MAXN; i++) prime_table[i] = 1;
    prime_table[0] = 0;
    prime_table[1] = 0;

    for (i = 2; i < upper; i++) if(prime_table[i]) {
        for (j = 2; j * i <= MAXN; j++) prime_table[j*i] = 0;
    }
}

int is_prime(int n) {
```

[①] http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

```
    return prime_table[n];
}
```

暴力枚举法优化版

来源未知。

```
int is_prime(unsigned long int n) {
    unsigned long int q, r, d;

    if (n < 32)
        return (0xa08a28acUL >> n) & 1;
    if ((n & 1) == 0)
        return 0;

    if (n % 3 == 0)
        return 0;
    if (n % 5 == 0)
        return 0;
    if (n % 7 == 0)
        return 0;

    for (d = 11;;) {
        q = n / d;
        r = n - q * d;
        if (q < d)
            return 1;
        if (r == 0)
            break;
        d += 2;
        q = n / d;
        r = n - q * d;
        if (q < d)
            return 1;
        if (r == 0)
            break;
        d += 4;
    }
    return 0;
}
```

例题

- wikioi 1430 素数判定，http://www.wikioi.com/problem/1430/

13.2 组合数学

第 14 章 大整数运算

在 32 位 CPU 下，C/C++ 中的 int 能表示的范围是 $[-2^{32}, 2^{32}-1]$，unsigned int 能表示的范围是 $[0, 2^{32}]$。所以，int 和 unsigned int 都不能保存超过 10 位的整数（解方程 $10^x \leq 2^{32}$，可得 $x \leq 9.63$）。有时我们需要参与运算的整数，可能会远远不止 10 位，我们称这种基本数据类型无法表示的整数为大整数。如何表示和存放大整数呢？基本的思想是：用数组模拟大整数。一个数组元素，存放大整数中的一位。

例如，一个 200 位的十进制整数，可以用 `int x[200]` 来表示，一个数组元素对应一个位。这样做有点浪费空间，因为一个 int 可以表示的范围远远大于 10。因此，我们可以用一个数组元素，表示 4 个数位（一个 int 可以表示的范围也远远大于 10000，为什么一个数组元素只表示 4 个数位，可不可以表示 9 个数位？留给读者思考），这时，数组不再是 10 进制，而是 10000 进制。使用万进制，数组长度可以缩减到原来的 1/4。

14.1 大整数加法

描述

求两个非负的大整数相加的和。

输入

有两行，每行是一个不超过 200 位的非负整数，可能有多余的前导 0。

输出

一行，即相加后的结果。结果里不能有多余的前导 0，即如果结果是 342，那么就不能输出为 0342。

样例输入

```
22222222222222222222
33333333333333333333
```

样例输出

```
5555555555555555555
```

代码

bigint_add.c

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

/* 一个数组元素表示 4 个十进制位，即数组是万进制的 */
#define BIGINT_MOD 10000
#define MOD_LEN 4
#define MAX_LEN (200/MOD_LEN+1)  /* 整数的最大位数 */

char    a[MAX_LEN * MOD_LEN], b[MAX_LEN * MOD_LEN];
int     x[MAX_LEN], y[MAX_LEN], z[MAX_LEN + 1];

/**
 * @brief 打印大整数.
 * @param[in] x 大整数，用数组表示，低位在低地址
 * @param[in] n 数组 x 的长度
 * @return 无
 */
void bigint_print(const int x[], const int n) {
    int i;
    int start_output = 0;  /* 用于跳过前导 0 */
    for (i = n - 1; i >= 0; --i) {
        if (start_output) {  /* 如果多余的 0 已经都跳过，则输出 */
            printf("%04d", x[i]);
        } else if (x[i] > 0) {
            printf("%d", x[i]);
            start_output = 1; /* 碰到第一个非 0 的值，就说明多余的 0 已经都跳过 */
        }
    }

    if(!start_output) printf("0");  /* 当 x 全为 0 时 */
}

/**
 * @brief 将输入的字符串转化为大整数.
 * @param[in] s 输入的字符串
 * @param[out] x 大整数
 * @return 无
 */
void bigint_input(const char s[], int x[]) {
    int i, j = 0;
    const int len = strlen(s);
    for (i = 0; i < MAX_LEN; i++) x[i] = 0;

    // for (i = len - 1; i >= 0; i--) a[j++] = s[i] - '0';
    for (i = len; i > 0; i -= MOD_LEN) {  /* [i-MOD_LEN, i) */
        int temp = 0;
```

```
        int k;
        const int low = i-MOD_LEN > 0 ? i-MOD_LEN : 0;
        for (k = low; k < i; k++) {
            temp = temp * 10 + s[k] - '0';
        }

        x[j++] = temp;
    }
}

/**
 * @brief 大整数加法
 * @param[in] x x
 * @param[in] y y
 * @param[out] z z=x+y
 * @return 无
 */
void bigint_add(const int x[], const int y[], int z[]) {
    int i;
    for (i = 0; i < MAX_LEN + 1; i++) z[i] = 0;

    for (i = 0; i < MAX_LEN; i++) {  /* 逐位相加 */
        z[i] += x[i] + y[i];
        if (z[i] >= BIGINT_MOD) {  /* 看是否要进位 */
            z[i] -= BIGINT_MOD;
            z[i+1] ++;  /* 进位 */
        }
    }
}


int main() {
    scanf("%s%s", a, b);

    bigint_input(a, x);
    bigint_input(b, y);

    bigint_add(x, y, z);
    bigint_print(z, MAX_LEN + 1);
    printf("\n");
    return 0;
}
```
bigint_add.c

相关的题目

与本题相同的题目：

- 《程序设计导引及在线实践》[1]第 144 页 7.1 节
- 百练 2981 大整数加法, http://poj.grids.cn/practice/2981/

[1]李文新, 程序设计导引及在线实践, 清华大学出版社, 2007

与本题相似的题目：

- TODO

14.2 大整数减法

描述

求两个非负的大整数相减的差。

输入

第 1 行是测试数据的组数 T，每组测试数据占 2 行，第 1 行是被减数 a，第 2 行是减数 b(a > b)，每行数据不超过 100 个字符，没有多余的前导 0。每组测试数据之间有一个空行。

输出

每组测试数据输出一行，即相应的差

样例输入

```
2
9999999999999999999999999999999999999
9999999999999

5409656775097850895687056798068970934546546575676768678435435345
1
```

分析

模拟小学生列竖式做加法，从个位开始逐位相加，超过或达到 10（这里用万进制，则是 10000）则进位。

两个 200 位的大整数相加，结果可能会有 201 位。

样例输出

```
9999999999999999999999990000000000000
5409656775097850895687056798068970934546546575676768678435435344
```

代码

bigint_sub.c

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

/* 一个数组元素表示 4 个十进制位，即数组是万进制的 */
#define BIGINT_MOD 10000
#define MOD_LEN 4
#define MAX_LEN (100/MOD_LEN+1)  /* 整数的最大位数 */

char    a[MAX_LEN * MOD_LEN], b[MAX_LEN * MOD_LEN];
int     x[MAX_LEN], y[MAX_LEN], z[MAX_LEN];

/**
 * @brief 打印大整数.
 * @param[in] x 大整数，用数组表示，低位在低地址
 * @param[in] n 数组 x 的长度
 * @return 无
 */
void bigint_print(const int x[], const int n) {
    int i;
    int start_output = 0;  /* 用于跳过前导 0 */
    for (i = n - 1; i >= 0; --i) {
        if (start_output) {  /* 如果多余的 0 已经都跳过，则输出 */
            printf("%04d", x[i]);
        } else if (x[i] > 0) {
            printf("%d", x[i]);
            start_output = 1; /* 碰到第一个非 0 的值，就说明多余的 0 已经都跳过 */
        }
    }

    if(!start_output) printf("0");  /* 当 x 全为 0 时 */
}

/**
 * @brief 将输入的字符串转化为大整数.
 * @param[in] s 输入的字符串
 * @param[out] x 大整数
 * @return 无
 */
void bigint_input(const char s[], int x[]) {
    int i, j = 0;
    const int len = strlen(s);
    for (i = 0; i < MAX_LEN; i++) x[i] = 0;

    // for (i = len - 1; i >= 0; i--) a[j++] = s[i] - '0';
    for (i = len; i > 0; i -= MOD_LEN) {  /* [i-MOD_LEN, i) */
        int temp = 0;
        int k;
        const int low = i-MOD_LEN > 0 ? i-MOD_LEN : 0;
        for (k = low; k < i; k++) {
            temp = temp * 10 + s[k] - '0';
```

```c
        }

        x[j++] = temp;
    }
}

/**
 * @brief 大整数减法.
 *
 * @param[in] x x
 * @param[in] y y, x>y
 * @param[out] z z=x-y
 * @return 无
 */
void bigint_sub(const int x[], const int y[], int z[]) {
    int i;
    for (i = 0; i < MAX_LEN; i++) z[i] = 0;

    for (i = 0; i < MAX_LEN; i++) {  /* 逐位相减 */
        z[i] += x[i] - y[i];
        if (z[i] < 0) {  /* 看是否要借位 */
            z[i] += BIGINT_MOD;
            z[i+1] --;  /* 借位 */
        }
    }
}

int main() {
    int T;
    scanf("%d", &T);

    while (T-- > 0) {
        scanf("%s%s",a,b);

        bigint_input(a, x);
        bigint_input(b, y);

        bigint_sub(x, y, z);
        bigint_print(z, MAX_LEN);
        printf("\n");
    }
    return 0;
}
```

bigint_sub.c

相关的题目

与本题相同的题目:

- 百练 2736 大整数减法, http://poj.grids.cn/practice/2736/

与本题相似的题目:

- TODO

14.3 大整数乘法

描述

求两个非负的大整数相乘的积。

输入

有两行，每行是一个不超过 200 位的非负整数，没有多余的前导 0。

输出

一行，即相乘后的结果。结果里不能有多余的前导 0。

样例输入

```
12345678900
98765432100
```

样例输出

```
1219326311126352690000
```

分析

两个 200 位的数相乘，积最多会有 400 位。

计算的过程基本上和小学生列竖式做乘法相同。为编程方便，并不急于处理进位，而将进位问题留待最后统一处理。

现以 835×49 为例来说明程序的计算过程。

先算 835×9。5×9 得到 45 个 1，3×9 得到 27 个 10，8×9 得到 72 个 100。由于不急于处理进位，所以 835×9 算完后，aResult 如下：

下标		5	4	3	2	1	0
aResult	····	0	0	0	72	27	45

接下来算 4×5。此处 4×5 的结果代表 20 个 10，因此要 aResult[1]+=20，变为：

下标		5	4	3	2	1	0
aResult	····		0	0	72	**47**	45

再下来算 4×3。此处 4×3 的结果代表 12 个 100，因此要 aResult[2]+= 12，变为：

下标		5	4	3	2	1	0
aResult	....	0	0	0	**84**	47	45

最后算 4×8。此处 4×8 的结果代表 32 个 1000 ，因此要 aResult[3]+= 32，变为：

下标		5	4	3	2	1	0
aResult	....	0	0	**32**	84	47	45

乘法过程完毕。接下来从 aResult[0] 开始向高位逐位处理进位问题。aResult[0] 留下 5，把 4 加到 aResult[1] 上，aResult[1] 变为 51 后，应留下 1 ，把 5 加到 aResult[2] 上……最终使得 aResult 里的每个元素都是 1 位数，结果就算出来了：

下标		5	4	3	2	1	0
aResult	....	0	4	0	9	1	5

总结一个规律，即一个数的第 i 位和另一个数的第 j 位相乘所得的数，一定是要累加到结果的第 i+j 位上。这里 i, j 都是从右往左，从 0 开始数。

代码

bigint_mul.c

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

/* 一个数组元素表示 4 个十进制位，即数组是万进制的 */
#define BIGINT_MOD 10000
#define MOD_LEN 4
#define MAX_LEN (200/MOD_LEN+1)  /* 整数的最大位数 */

char    a[MAX_LEN * MOD_LEN], b[MAX_LEN * MOD_LEN];
int     x[MAX_LEN], y[MAX_LEN], z[MAX_LEN * 2];

/**
 * @brief 打印大整数.
 * @param[in] x 大整数，用数组表示，低位在低地址
 * @param[in] n 数组 x 的长度
 * @return 无
 */
void bigint_print(const int x[], const int n) {
    int i;
    int start_output = 0;  /* 用于跳过前导 0 */
    for (i = n - 1; i >= 0; --i) {
        if (start_output) {  /* 如果多余的 0 已经都跳过，则输出 */
            printf("%04d", x[i]);
        } else if (x[i] > 0) {
```

```
            printf("%d", x[i]);
            start_output = 1; /* 碰到第一个非 0 的值，就说明多余的 0 已经都跳过 */
        }
    }

    if(!start_output) printf("0");  /* 当 x 全为 0 时 */
}

/**
 * @brief 将输入的字符串转化为大整数.
 * @param[in] s 输入的字符串
 * @param[out] x 大整数
 * @return 无
 */
void bigint_input(const char s[], int x[]) {
    int i, j = 0;
    const int len = strlen(s);
    for (i = 0; i < MAX_LEN; i++) x[i] = 0;

    // for (i = len - 1; i >= 0; i--) a[j++] = s[i] - '0';
    for (i = len; i > 0; i -= MOD_LEN) {  /* [i-MOD_LEN, i) */
        int temp = 0;
        int k;
        const int low = i-MOD_LEN > 0 ? i-MOD_LEN : 0;
        for (k = low; k < i; k++) {
            temp = temp * 10 + s[k] - '0';
        }

        x[j++] = temp;
    }
}

/**
 * @brief 大整数乘法.
 * @param[in] x x
 * @param[in] y y
 * @param[out] z z=x*y
 * @return 无
 */
void bigint_mul(const int x[], const int y[], int z[]) {
    int i, j;
    for (i = 0; i < MAX_LEN * 2; i++) z[i] = 0;

    for (i = 0; i < MAX_LEN; i++) {
        for (j = 0; j < MAX_LEN; j++) { /* 用 y[i]，去乘以 x 的各位 */
            z[i + j] += y[i] * x[j]; /* 两数第 i，j 位相乘，累加到结果的第 i+j 位 */

            if (z[i + j] >= BIGINT_MOD) { /* 看是否要进位 */
                z[i + j + 1] += z[i + j] / BIGINT_MOD; /* 进位 */
                z[i + j] %= BIGINT_MOD;
            }
        }
    }
```

```
}

int main() {
    scanf("%s%s", a, b);

    bigint_input(a, x);
    bigint_input(b, y);

    bigint_mul(x, y, z);
    bigint_print(z, MAX_LEN * 2);
    printf("\n");
    return 0;
}
```

bigint_mul.c

相关的题目

与本题相同的题目:

- 《程序设计导引及在线实践》[①]第 146 页 7.2 节
- 百练 2980 大整数乘法, http://poj.grids.cn/practice/2980/

与本题相似的题目:

- TODO

14.4 大整数除法

描述

求两个非负的大整数相除的商。

输入

第 1 行是测试数据的组数 T，每组测试数据占 2 行，第 1 行是被除数，第 2 行是除数，每行数据不超过 100 个字符。每组测试数据之间有一个空行。

输出

每组测试数据输出一行，即相应的整数商

[①]李文新, 程序设计导引及在线实践, 清华大学出版社, 2007

样例输入

```
3
2405337312963373359009260457742057439230496493930355595797660791082739646
2987192585318701752584429931160870372907079248971095012509790550883793197894

10000000000000000000000000000000000000000
10000000000

5409656775097850895687056798068970934546546575676768678435435345
1
```

样例输出

```
0
1000000000000000000000000000000
5409656775097850895687056798068970934546546575676768678435435345
```

分析

基本的思想是反复做减法，看看从被除数里最多能减去多少个除数，商就是多少。一个一个减显然太慢，如何减得更快一些呢？以 7546 除以 23 为例来看一下：开始商为 0。先减去 23 的 100 倍，就是 2300，发现够减 3 次，余下 646。于是商的值就增加 300。然后用 646 减去 230，发现够减 2 次，余下 186，于是商的值增加 20。最后用 186 减去 23，够减 8 次，因此最终商就是 328。

所以本题的核心是要写一个大整数的减法函数，然后反复调用该函数进行减法操作。

计算除数的 10 倍、100 倍的时候，不用做乘法，直接在除数后面补 0 即可。

代码

bigint_div.c

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

/* 一个数组元素表示 4 个十进制位，即数组是万进制的 */
#define BIGINT_MOD 10000
#define MOD_LEN 4
#define MAX_LEN (100/MOD_LEN+1)  /* 整数的最大位数 */

char    a[MAX_LEN * MOD_LEN], b[MAX_LEN * MOD_LEN];
int     x[MAX_LEN], y[MAX_LEN], z[MAX_LEN];

/**
 * @brief 打印大整数.
 * @param[in] x 大整数，用数组表示，低位在低地址
 * @param[in] n 数组 x 的长度
```

```
 * @return 无
 */
void bigint_print(const int x[], const int n) {
    int i;
    int start_output = 0;  /* 用于跳过前导 0 */
    for (i = n - 1; i >= 0; --i) {
        if (start_output) {  /* 如果多余的 0 已经都跳过，则输出 */
            printf("%04d", x[i]);
        } else if (x[i] > 0) {
            printf("%d", x[i]);
            start_output = 1; /* 碰到第一个非 0 的值，就说明多余的 0 已经都跳过 */
        }
    }

    if(!start_output) printf("0");  /* 当 x 全为 0 时 */
}

/**
 * @brief 将输入的字符串转化为大整数.
 * @param[in] s 输入的字符串
 * @param[out] x 大整数
 * @return 无
 */
void bigint_input(const char s[], int x[]) {
    int i, j = 0;
    const int len = strlen(s);
    for (i = 0; i < MAX_LEN; i++) x[i] = 0;

    // for (i = len - 1; i >= 0; i--) a[j++] = s[i] - '0';
    for (i = len; i > 0; i -= MOD_LEN) {  /* [i-MOD_LEN, i) */
        int temp = 0;
        int k;
        const int low = i-MOD_LEN > 0 ? i-MOD_LEN : 0;
        for (k = low; k < i; k++) {
            temp = temp * 10 + s[k] - '0';
        }

        x[j++] = temp;
    }
}

/**
 * @brief 计算大整数的位数.
 *
 * @param[inout] x 大整数
 * @return 位数
 */
static int length(const int x[]) {
    int i;
    int result = 0;
    for (i = MAX_LEN - 1; i >= 0; i--) if (x[i] > 0) {
        result = i + 1;
        break;
```

```
    }
    return result;
}

/**
 * @brief 大整数减法.
 *
 * @param[inout] x x
 * @param[in] y y
 * @return 如果 x < y，返回-1，如果 x=y，返回 0，如果 x>y，返回 1
 */
static int bigint_sub(int x[], const int y[]) {
    int i;
    const int lenx = length(x);
    const int leny = length(y);

    /* 判断 x 是否比 y 大 */
    if (lenx < leny) return -1;
    else if (lenx == leny) {
        int larger = 0;
        for (i = lenx - 1; i >= 0; i--) {
            if (x[i] > y[i]) {
                larger = 1;
            } else if (x[i] < y[i]) {
                if (!larger) return -1;
            }
        }
    }

    for (i = 0; i < MAX_LEN; i++) {  /* 逐位相减 */
        x[i] -= y[i];
        if (x[i] < 0) {  /* 看是否要借位 */
            x[i] += BIGINT_MOD;
            x[i+1] --;  /* 借位 */
        }
    }

    return 1;
}

/**
 * @brief 大整数除法.
 *
 * @param[inout] x x
 * @param[in] y y
 * @param[out] z z=x/y
 * @return 无
 */
void bigint_div(int x[], const int y[], int z[]) {
    int i;
    int *yy; /* y 的副本 */
    const int xlen = length(x);
    int ylen = length(y);
```

```
    const int times = xlen - ylen;

    for (i = 0; i < MAX_LEN; i++) z[i] = 0;
    if (times < 0) return;

    yy = (int*)malloc(sizeof(int) * MAX_LEN);
    memcpy(yy, y, sizeof(int) * MAX_LEN);


    /* 将 yy 右移 times 位，使其长度和 x 相同，即 yy 乘以 10000 的 times 次幂 */
    for (i = xlen - 1; i >= 0; i--) {
        if (i >= times) yy[i] = yy[i - times];
        else yy[i] = 0;
    }

    /* 先减去若干个 y×(10000 的 times 次方)，
      不够减了，再减去若干个 y×(10000 的 times-1 次方)
      一直减到不够减为止 */
    ylen = xlen;
    for (i = 0; i <= times; i++) {
        int j;
        while (bigint_sub(x, yy) >= 0) {
            z[times - i]++;
        }

        /* yy 除以 BIGINT_MOD，即左移一位 */
        for (j = 1; j < ylen; j++) {
            yy[j - 1] = yy[j];
        }
        yy[--ylen] = 0;
    }

    /* 下面的循环统一处理进位 */
    for (i = 0; i < MAX_LEN - 1; i++) {
        if (z[i] >= BIGINT_MOD) {  /* 看是否要进位 */
            z[i+1] += z[i] / BIGINT_MOD;  /* 进位 */
            z[i] %= BIGINT_MOD;
        }
    }
    free(yy);
}

int main() {
    int T;
    scanf("%d", &T);

    while (T-- > 0) {
        scanf("%s%s",a,b);

        bigint_input(a, x);
        bigint_input(b, y);

        bigint_div(x, y, z);
```

```
        bigint_print(z, MAX_LEN);
        printf("\n");
    }
    return 0;
}
```

bigint_div.c

相关的题目

与本题相同的题目：

- 《程序设计导引及在线实践》[①]第 149 页 7.3 节
- 百练 2737 大整数除法, http://poj.grids.cn/practice/2737/

与本题相似的题目：

- TODO

[①]李文新, 程序设计导引及在线实践, 清华大学出版社, 2007

第 15 章 字符串

15.1 字符串排序

15.2 单词查找树

15.3 子串查找

字符串的一种基本操作就是**子串查找** (substring search)：给定一个长度为 N 的文本和一个长度为 M 的模式串 (pattern string)，在文本中找到一个与该模式相符的子字符串。

最简单的算法是暴力查找，时间复杂度是 $O(MN)$。下面介绍两个更高效的算法。

15.3.1 KMP 算法

KMP 算法是 Knuth、Morris 和 Pratt 在 1976 年发表的。它的基本思想是，当出现不匹配时，就能知晓一部分文本的内容（因为在匹配失败之前它们已经和模式相匹配)。我们可以利用这些信息避免将指针回退到所有这些已知的字符之前。这样，当出现不匹配时，可以提前判断如何重新开始查找，而这种判断只取决于模式本身。

详细解释请参考《算法》[①]第 5.3.3 节。这本书讲的是确定有限状态自动机 (DFA) 的方法。

推荐网上的几篇比较好的博客，讲的是部分匹配表 (partial match table) 的方法（即 next 数组)，“字符串匹配的 KMP 算法”http://t.cn/zTOPfdh，图文并茂，非常通俗易懂，作者是阮一峰；“KMP 算法详解”http://www.matrix67.com/blog/archives/115，作者是顾森 Matrix67；”Knuth-Morris-Pratt string matching” http://www.ics.uci.edu/ eppstein/161/960227.html。

使用 next 数组的 KMP 算法的 C 语言实现如下。

kmp.c

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

[①]《算法》，Robert Sedgewick，人民邮电出版社，http://book.douban.com/subject/10432347/

```c
#include <string.h>

/*
 * @brief 计算部分匹配表，即 next 数组.
 *
 * @param[in] pattern 模式串
 * @param[in] m 模式串的长度
 * @param[out] next next 数组
 * @return 无
 */
void compute_prefix(const char pattern[], const int m, int next[]) {
    int i;
    int j = -1;

    next[0] = j;
    for (i = 1; i < m; i++) {
        while (j > -1 && pattern[j + 1] != pattern[i]) j = next[j];

        if (pattern[i] == pattern[j + 1]) j++;
        next[i] = j;
    }
}

/*
 * @brief KMP 算法.
 *
 * @param[in] text 文本
 * @param[in] n 文本的长度
 * @param[in] pattern 模式串
 * @param[in] m 模式串的长度
 * @return 成功则返回第一次匹配的位置，失败则返回-1
 */
int kmp(const char text[], const int n, const char pattern[], const int m) {
    int i;
    int j = -1;
    int *next = (int*)malloc(sizeof(int) * m);

    compute_prefix(pattern, m, next);

    for (i = 0; i < n; i++) {
        while (j > -1 && pattern[j + 1] != text[i]) j = next[j];

        if (text[i] == pattern[j + 1]) j++;
        if (j == m - 1) {
            free(next);
            return i-j;
        }
    }

    free(next);
    return -1;
}
```

```c
int main(int argc, char *argv[]) {
    char text[] = "ABC ABCDAB ABCDABCDABDE";
    char pattern[] = "ABCDABD";
    char *ch = text;
    int i = kmp(text, strlen(text), pattern, strlen(pattern));

    if (i >= 0) printf("matched @: %s\n", ch + i);
    return 0;
}
```

kmp.c

15.3.2 Boyer-Moore 算法

详细解释请参考《算法》[①]第 5.3.4 节。

推荐网上的几篇比较好的博客，“字符串匹配的 Boyer-Moore 算法” http://www.ruanyifeng.com/blog/2013/05/boyer-moore_string_search_algorithm.html，图文并茂，非常通俗易懂，作者是阮一峰；Boyer-Moore algorithm, http://www-igm.univ-mlv.fr/ lecroq/string/node14.html。

有兴趣的读者还可以看原始论文[②]。

Boyer-Moore 算法的 C 语言实现如下。

boyer_moore.c

```c
/**
 * 本代码参考了 http://www-igm.univ-mlv.fr/~lecroq/string/node14.html
 * 精力有限的话，可以只计算坏字符的后移，好后缀的位移是可选的，因此可以删除
 * suffixes(), pre_gs() 函数
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ASIZE 256  /* ASCII 字母的种类 */

/*
 * @brief 预处理，计算每个字母最靠右的位置.
 *
 * @param[in] pattern 模式串
 * @param[in] m 模式串的长度
 * @param[out] right 每个字母最靠右的位置
 * @return 无
 */
static void pre_right(const char pattern[], const int m, int right[]) {
    int i;

    for (i = 0; i < ASIZE; ++i) right[i] = -1;
    for (i = 0; i < m; ++i) right[pattern[i]] = i;
}
```

[①]《算法》，Robert Sedgewick，人民邮电出版社，http://book.douban.com/subject/10432347/

[②]BOYER R.S., MOORE J.S., 1977, A fast string searching algorithm. Communications of the ACM. 20:762-772.

```
static void suffixes(const char pattern[], const int m, int suff[]) {
    int f, g, i;

    suff[m - 1] = m;
    g = m - 1;
    for (i = m - 2; i >= 0; --i) {
        if (i > g && suff[i + m - 1 - f] < i - g)
            suff[i] = suff[i + m - 1 - f];
        else {
            if (i < g)
                g = i;
            f = i;
            while (g >= 0 && pattern[g] == pattern[g + m - 1 - f])
                --g;
            suff[i] = f - g;
        }
    }
}

/*
 * @brief 预处理，计算好后缀的后移位置.
 *
 * @param[in] pattern 模式串
 * @param[in] m 模式串的长度
 * @param[out] gs 好后缀的后移位置
 * @return 无
 */
static void pre_gs(const char pattern[], const int m, int gs[]) {
    int i, j;
    int *suff = (int*)malloc(sizeof(int) * (m + 1));

    suffixes(pattern, m, suff);

    for (i = 0; i < m; ++i) gs[i] = m;

    j = 0;
    for (i = m - 1; i >= 0; --i) if (suff[i] == i + 1)
        for (; j < m - 1 - i; ++j) if (gs[j] == m)
            gs[j] = m - 1 - i;
    for (i = 0; i <= m - 2; ++i)
        gs[m - 1 - suff[i]] = m - 1 - i;
    free(suff);
}

/**
 * @brief Boyer-Moore 算法.
 *
 * @param[in] text 文本
 * @param[in] n 文本的长度
 * @param[in] pattern 模式串
 * @param[in] m 模式串的长度
```

```
 * @return 成功则返回第一次匹配的位置，失败则返回-1
 */
int boyer_moore(const char text[], const int n,
                const char pattern[], const int m) {
    int i, j;
    int right[ASIZE];  /* bad-character shift */
    int *gs = (int*)malloc(sizeof(int) * (m + 1));  /* good-suffix shift */

    /* Preprocessing */
    pre_right(pattern, m, right);
    pre_gs(pattern, m, gs);

    /* Searching */
    j = 0;
    while (j <= n - m) {
        for (i = m - 1; i >= 0 && pattern[i] == text[i + j]; --i);

        if (i < 0) { /* 找到一个匹配 */
            /* printf("%d ", j);
            j += bmGs[0]; */
            free(gs);
            return j;
        } else {
            const int max = gs[i] > right[text[i + j]] - m + 1 + i ?
                gs[i] : i - right[text[i + j]];
            j += max;
        }
    }
    free(gs);
    return -1;
}


int main() {
    const char text[]="HERE IS A SIMPLE EXAMPLE";
    const char pattern[] = "EXAMPLE";
    const int pos = boyer_moore(text, strlen(text), pattern, strlen(pattern));
    printf("%d\n", pos); /* 17 */
    return 0;
}
```

boyer_moore.c

15.3.3 Rabin-Karp 算法

详细解释请参考《算法》[①]第 5.3.5 节。

Rabin-Karp 算法的 C 语言实现如下。

rabin_karp.c

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

[①]《算法》，Robert Sedgewick，人民邮电出版社，http://book.douban.com/subject/10432347/

```
const int R = 256;  /** ASCII 字母表的大小，R 进制 */
/** 哈希表的大小，选用一个大素数，这里用 16 位整数范围内最大的素数 */
const long Q = 0xfff1;

/*
 * @brief 哈希函数.
 *
 * @param[in] key 待计算的字符串
 * @param[int] M 字符串的长度
 * @return 长度为 M 的子字符串的哈希值
 */
static long hash(const char key[], const int M) {
    int j;
    long h = 0;
    for (j = 0; j < M; ++j) h = (h * R + key[j]) % Q;
    return h;
}

/*
 * @brief 计算新的 hash.
 *
 * @param[int] h 该段子字符串所对应的哈希值
 * @param[in] first 长度为 M 的子串的第一个字符
 * @param[in] next 长度为 M 的子串的下一个字符
 * @param[int] RM R^(M-1) % Q
 * @return 起始于位置 i+1 的 M 个字符的子字符串所对应的哈希值
 */
static long rehash(const long h, const char first, const char next,
                   const long RM) {
    long newh = (h + Q - RM * first % Q) % Q;
    newh = (newh * R + next) % Q;
    return newh;
}

/*
 * @brief 用蒙特卡洛算法，判断两个字符串是否相等.
 *
 * @param[in] pattern 模式串
 * @param[in] substring 原始文本长度为 M 的子串
 * @param[in] M 模式串的长度，也是 substring 的长度
 * @return 两个字符串相同，返回 1，否则返回 0
 */
static int check(const char pattern[], const char substring[], const int M) {
    return 1;
}

/**
 * @brief Rabin-Karp 算法.
 *
 * @param[in] text 文本
 * @param[in] n 文本的长度
 * @param[in] pattern 模式串
```

```
 * @param[in] m 模式串的长度
 * @return 成功则返回第一次匹配的位置，失败则返回-1
 */
int rabin_karp(const char text[], const int n,
               const char pattern[], const int m) {
    int i;
    const long pattern_hash = hash(pattern, m);
    long text_hash = hash(text, m);
    int RM = 1;
    for (i = 0; i < m - 1; ++i) RM = (RM * R) % Q;

    for (i = 0; i <= n - m; ++i) {
        if (text_hash == pattern_hash) {
            if (check(pattern, &text[i], m)) return i;
        }
        text_hash = rehash(text_hash, text[i], text[i + m], RM);
    }
    return -1;
}


int main() {
    const char text[]="HERE IS A SIMPLE EXAMPLE";
    const char pattern[] = "EXAMPLE";
    const int pos = rabin_karp(text, strlen(text), pattern, strlen(pattern));
    printf("%d\n", pos); /* 17 */
    return 0;
}
```

rabin_karp.c

15.3.4 总结

算法	版本	复杂度		在文本中回退	正确性	辅助空间
		最坏情况	平均情况			
KMP 算法	完整的 DFA	2N	1.1N	否	是	MR
	部分匹配表	3N	1.1N	否	是	M
	完整版本	3N	N/M	是	是	R
Boyer-Moore 算法	坏字符向后位移	MN	N/M	是	是	R
Rabin-Karp 算法 *	蒙特卡洛算法	7N	7N	否	是 *	1
	拉斯维加斯算法	$7N^*$	7N	是	是	1

* 概率保证，需要使用均匀和独立的散列函数

15.4 正则表达式

第 16 章
基础功能

在面试和笔试中，经常会出现这类题目，把 C++, Java 标准库中的一些函数单独拿出来，让你重新实现。这类题目短小精悍，很能考验一个人的基本功是否扎实。

16.1 下一个排列

描述

实现 C++ STL 中的 `next_permutation()`，函数原型如下：

```
/**
 * @brief 返回下一个排列，例如当前排列是 12345，下一个是 12354
 * @param[inout] num 当前排列，例如 12345
 * @param[in] len num 的长度
 * @return 无
 */
void next_permutation(int num[], int len);
```

分析

算法过程如图 16-1所示（来自 http://fisherlei.blogspot.com/2012/12/leetcode-next-permutation.html）。

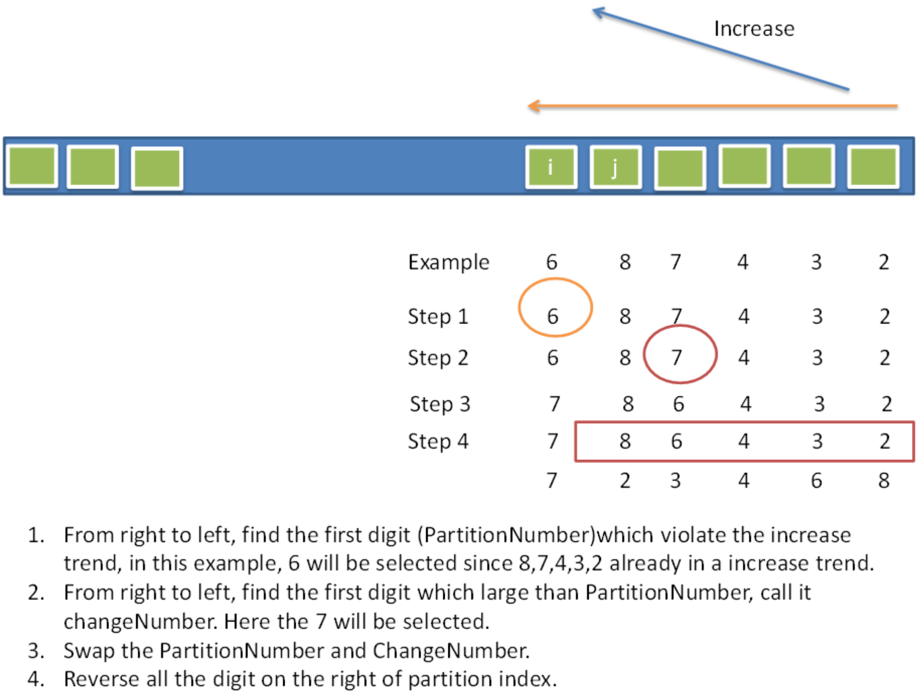


图 16-1 下一个排列算法流程

代码

next_permutation.c

```
static void swap(int array[], int i, int j) {
    const int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}

static void reverse(int array[], int begin, int end) { //左闭右开区间
    end--;
    while (begin < end)
        swap(array, begin++, end--);
}

/**
 * @brief 返回下一个排列，例如当前排列是 12345，下一个是 12354
 * @param[inout] num 当前排列，例如 12345
 * @param[in] len num 的长度
 * @return 成功返回 1，失败返回 0
 */
int next_permutation(int num[], int len) {
```

```
    int i = len - 2;  // partition number's index
    while (i >= 0 && num[i] >= num[i + 1])
        i--;
    if(i == -1) return 0;
    if (i >= 0) {
        int j = len - 1;  // change number's index
        while (num[j] <= num[i])
            --j;
        swap(num, i, j);
        reverse(num, i + 1, len);
    } else {
        reverse(num, i + 1, len);
    }
    return 1;
}
```

next_permutation.c

相关的题目

与本题相同的题目:

- LeetCode - Next Permutation, http://leetcode.com/onlinejudge#question_31

16.2 数组循环右移

描述

将一个长度为 n 的数组 A 的元素循环右移(ROR, Rotate Right)k 位，比如数组 1, 2, 3, 4, 5 循环右移 3 位之后就变成 3, 4, 5, 1, 2。

方法一

最直接的做法是另开一个大小一样的数组 B，遍历一下，令 $B[(i+k)\%n] = A[i]$，再将 B 的内容写回到 A 即可。这个方法的时间复杂度为 O(n)，空间复杂度也为 O(n)。代码如下:

ror.c

```
void ror1(int array[], int n, int k) {
    int i;
    int *B = (int*) malloc(n * sizeof(int));

    k %= n;
    if (k == 0)
        return;

    for (i = 0; i < n; i++) {
        B[(i + k) % n] = array[i];
    }
    for (i = 0; i < n; i++) {
        array[i] = B[i];
```

```
    }
}
```

ror.c

方法二

另一种简单的做法，每次将数组中的所有元素右移一位，循环 k 次。这个方法的时间复杂度为 O(n*k)，空间复杂度为 O(1)。代码如下：

ror.c

```
void ror2(int array[], int n, int k) {
    int i, tmp;
    k %= n;
    if (k == 0)
        return;

    while (k--) {
        tmp = array[n - 1];
        for (i = n - 1; i > 0; i--) {
            array[i] = array[i - 1];
        }
        array[0] = tmp;
    }
}
```

ror.c

方法三

先将 A 的元素倒置，即 1, 2, 3, 4, 5 变成 5, 4, 3, 2, 1，然后将前 k 位倒置，即 3, 4, 5, 2, 1，再将后 n-k 位倒置，即 3, 4, 5, 1, 2，完成。

证明：记 A 的前 n-k 位为 X，后 k 位为 Y，则 A=XY，将 A 循环右移 k 位后，应该得到 YX。根据该算法，先将 A 整体倒置，得到 $(XY)^T = Y^T X^T$，然后将前 k 位倒置，得到 $Y X^T$，最后将后 n-k 位倒置，得到 YX，正好是所求的结果，证毕。

这个方法的时间复杂度为 O(2n)，空间复杂度为 O(1)。代码如下：

ror.c

```
static void swap(int array[], int i, int j) {
    const int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}

static void reverse(int array[], int begin, int end) { //左闭右开区间
    end--;
    while (begin < end)
        swap(array, begin++, end--);
}

void ror3(int *array, int n, int k) {
    k %= n;
```

```
    if (k == 0)
        return;

    reverse(array, 0, n);
    reverse(array, 0, k);
    reverse(array, k, n - k);
}
```
ror.c

这种方法需要对每个位置写入 2 次，看上去也不怎么好，那有没有更好的呢？

方法四

我们要做的只是把每个元素放到它应该在的位置，比如开头的例子，1 应该放在 4 的位置，把 1 放好之后，4 就没地方了，那 4 应该在哪呢，在 2 的位置，依此类推，就可以把所有的元素都放好，而且只放了一次。看上去这样做很完美，但仔细想想就能想出反例子，比如 1, 2, 3, 4, 5, 6, 7, 8, 9 右移 3 位，就是 1 放在 4 个位置，4 放在 7 的位置，然后 7 放回 1，这时候一圈兜完了，但只排好了 3 个元素，剩下的 6 个元素没有动过，怎么办呢？继续下一个，就是 2，然后 2、5、8 也排好了，继续 3、6、9，这时候下一个元素是 1 了，应该停止了（因为 1、4、7 已经排好了），那程序怎么会知道停在这里了，于是就想到了最大公约数，9 和 3 的最大公约数是 3，于是做前 3 个数的循环就可以了，为什么上一个例子只需做一次，因为元素个数（5）和移动位数（3）互质。具体的数学证明略。

代码如下：

ror.c
```
static int gcd(int a, int b) {
    assert(a >= b);
    if (b == 0) {
        return a;
    }

    while (b > 0) {
        int tmp = a % b;
        a = b;
        b = tmp;
    }

    return a;
}

void ror4(int *array, int n, int k) {
    int i;
    const int g = gcd(n, k);

    k %= n;
    if (k == 0)
        return;

    for (i = 0; i < g; ++i) {
        int j = i;
```

```
        int cur = array[j], tmp;

        do {
            tmp = array[(j + k) % n];
            array[(j + k) % n] = cur;
            cur = tmp;
            j = (j + k) % n;
        } while (j != i);
    }
}

// test
int main(void) {
    int i;
    int a[] = { 1, 2, 3, 4, 5 };
    ror4(a, 5, 3);
    for (i = 0; i < 5; i++) {
        printf("%d ", a[i]);
    }

    return EXIT_SUCCESS;
}
```

ror.c