

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 2013-9-13

版权声明

本作品采用“Creative Commons 署名 -非商业性使用 -相同方式共享 3.0 Unported 许可协议 (cc by-nc-sa)”进行许可。<http://creativecommons.org/licenses/by-nc-sa/3.0/>

内容简介

本书的目标读者是准备去北美找工作的码农，也适用于在国内找工作的码农，以及刚接触 ACM 算法竞赛的新手。

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，这就是所谓的“经典”。同时，本书的每一个题目，都至少在两本纸质书中出现过。

这本书的定位，与 ACM 算法竞赛类书籍不同。全书的题目比 ACM 竞赛简单，没有高难度的题目，但每道题目，都有详细生动的解释，还给出了可以直接在 OJ 上 AC 的代码。同时，题目的范围不限于算法竞赛，还包括了一些面试中常碰到的工程类题目。

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。

Github 地址

本书是开源的，项目地址：<https://github.com/soulmachine/acm-cheatsheet>

^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	1	4.6.4 食物链	40
第 2 章 线性表	2	4.7 线段树	42
第 3 章 栈和队列	3	4.7.1 原理和实现	42
3.1 栈	3	4.7.2 Balanced Lineup	43
3.1.1 栈的 C 语言实现	3	4.7.3 线段树练习 1	45
3.1.2 Hanoi 塔问题	5	4.7.4 A Simple Problem with Integers	49
3.1.3 进制转换	6	4.7.5 约瑟夫问题	52
3.2 队列	8	第 5 章 查找	56
3.2.1 队列的 C 语言实现	8	5.1 折半查找	56
3.2.2 打印杨辉三角	10	第 6 章 排序	57
第 4 章 树	12	6.1 插入排序	57
4.1 二叉树的遍历	12	6.1.1 直接插入排序	57
4.2 线索二叉树	15	6.1.2 折半插入排序	58
4.3 Morris Traversal	18	6.1.3 希尔 (Shell) 插入排序	58
4.3.1 Morris 中序遍历	18	6.2 交换排序	60
4.3.2 Morris 先序遍历	19	6.2.1 冒泡排序	60
4.3.3 Morris 后序遍历	20	6.2.2 快速排序	61
4.3.4 C 语言实现	21	6.3 选择排序	63
4.4 重建二叉树	25	6.3.1 简单选择排序	63
4.5 堆	27	6.3.2 堆排序	64
4.5.1 原理和实现	27	6.4 归并排序	65
4.5.2 最小的 N 个和	31	6.5 基数排序	66
4.6 并查集	33	6.6 总结和比较	70
4.6.1 原理和实现	33	第 7 章 暴力枚举法	71
4.6.2 病毒感染者	36	7.1 枚举排列	71
4.6.3 两个黑帮	38		

7.1.1	生成 1 到 n 的全排列 . . .	71	10.8	棋盘型动态规划	137
7.1.2	生成可重集的排列 . . .	73	10.8.1	数字三角形	137
7.1.3	下一个排列	76	10.8.2	过河卒	140
7.2	子集生成	77	10.8.3	传纸条	143
7.2.1	增量构造法	77	10.8.4	骑士游历	146
7.2.2	位向量法	78	10.9	划分型动态规划	148
7.2.3	二进制法	78	10.9.1	乘积最大	148
第 8 章	分治法	80	10.9.2	数的划分	151
8.1	棋盘覆盖	80	10.10	树型动态规划	153
8.2	循环赛日程表	83	10.10.1	访问艺术馆	153
第 9 章	贪心法	87	10.10.2	没有上司的舞会	156
9.1	最优装载	87	10.11	最大子矩形	159
9.2	哈弗曼编码	87	10.11.1	奶牛浴场	159
9.3	部分背包问题	89	10.11.2	最大全 1 子矩阵	164
第 10 章	动态规划	90	第 11 章	广度优先搜索	167
10.1	动规和记忆化搜索的区别 . . .	91	11.1	算法框架	167
10.2	最长公共子序列	91	11.2	走迷宫	173
10.3	最大连续子序列和	94	11.3	八数码问题	178
10.4	最大 M 子段和	97	11.4	四子连棋	190
10.5	背包问题	100	11.5	双向 BFS	196
10.5.1	0-1 背包问题	100	11.5.1	八数码问题	196
10.5.2	完全背包问题	104	11.6	A* 算法	196
10.5.3	多重背包问题	109	11.6.1	八数码问题	197
10.6	序列型动态规划	112	第 12 章	深度优先搜索	205
10.6.1	最长上升子序列	113	12.1	深搜与回溯法的区别	205
10.6.2	嵌套矩形	114	12.2	深搜与递归的区别	205
10.6.3	线段覆盖 2	117	12.3	四色问题	206
10.6.4	硬币问题	120	12.4	全排列	208
10.7	区间型动态规划	126	12.5	八皇后问题	210
10.7.1	最优矩阵链乘	126	12.6	还原 IP 地址	214
10.7.2	石子合并	129	12.7	Combination Sum	215
10.7.3	矩阵取数游戏	131	12.8	Combination Sum II	216

第 13 章 图	218	14.1.2 扩展欧几里德算法 . . .	273
13.1 深度优先搜索	218	14.1.3 素数判定	275
13.1.1 黑白图像	219	14.1.4 大整数取模	278
13.1.2 欧拉回路	221	14.2 组合数学	279
13.2 广度优先搜索	225	第 15 章 大整数运算	280
13.3 最小生成树	225	15.1 大整数加法	280
13.3.1 Prim 算法	225	15.2 大整数减法	283
13.3.2 Kruskal 算法	232	15.3 大整数乘法	286
13.3.3 Highways	237	15.4 大整数除法	289
13.3.4 最优布线问题	240	15.5 大数阶乘	294
13.4 最短路径	241	15.5.1 大数阶乘的位数	294
13.4.1 单源最短路径——Di- jkstra 算法	241	15.5.2 大数阶乘	296
13.4.2 每点最短路径—— Floyd 算法	247	第 16 章 字符串	299
13.4.3 例题: HDU 2544 最短路	251	16.1 字符串排序	299
13.4.4 例题: POJ 1125 Stock- broker Grapevine	254	16.2 单词查找树	299
13.5 拓扑排序	258	16.3 子串查找	299
13.5.1 例题: POJ 1094 Sorting It All Out	261	16.3.1 KMP 算法	299
13.6 关键路径	266	16.3.2 Boyer-Moore 算法	301
第 14 章 数学方法与常见模型	272	16.3.3 Rabin-Karp 算法	303
14.1 数论	272	16.3.4 总结	305
14.1.1 欧几里德算法	272	16.4 正则表达式	305
		第 17 章 基础功能	306
		17.1 下一个排列	306
		17.2 数组循环右移	308

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
/**
 * @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：

```

/*
 * @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

```

hanoi.c

```

* @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 进制转换

convert_base.cpp

```

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

/**
 * @brief 进制转换, 将一个 10 进制整数转化为 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 进制转换，将一个 10 进制整数转化为 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;
}

/**
 * @brief 进制转换，将一个 d 进制整数转化为 10 进制，d<=16.
 * @param[in] s d 进制整数
 * @param[in] d d 进制
 * @return 10 进制整数
 */
int restore(const char s[MAX], const int d) {
    int i;
    int result = 0;
    int one;

    for (i = 0; s[i] != '\0'; i++) {
        if (s[i] >= '0' && s[i] <= '9') one = s[i] - '0';
        else if (s[i] >= 'A' && s[i] <= 'F') one = s[i] - 'A' + 10;
        else one = s[i] - 'a' + 10; /* (s[i] >= 'a' && s[i] <= 'f') */

        result = result * d + one;
    }
    return result;
}

```

}

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;
}

/**
 * @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

```

#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

```
#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);
    }
}
```

binary_tree.cpp

```

    }
}

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

```

```

        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。

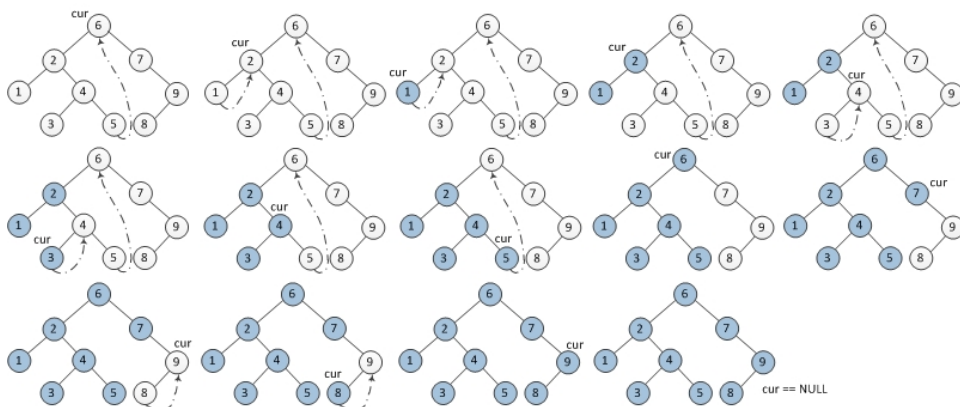


图 4-2 Morris 中序遍历

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。

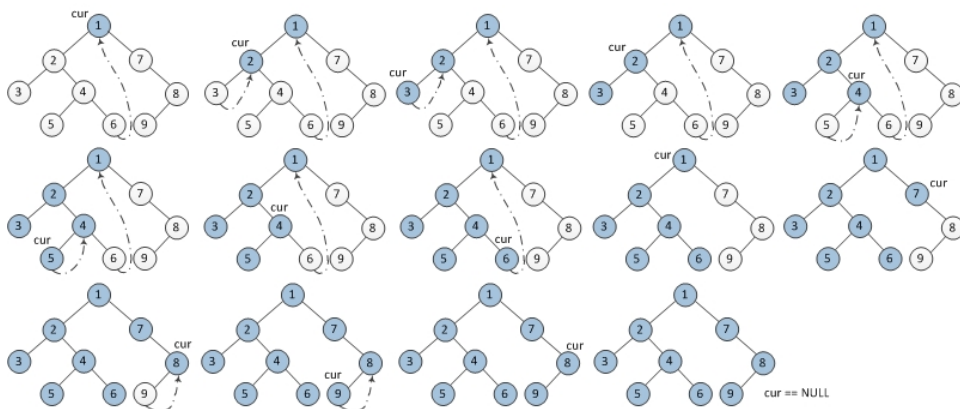


图 4-3 Morris 先序遍历

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。

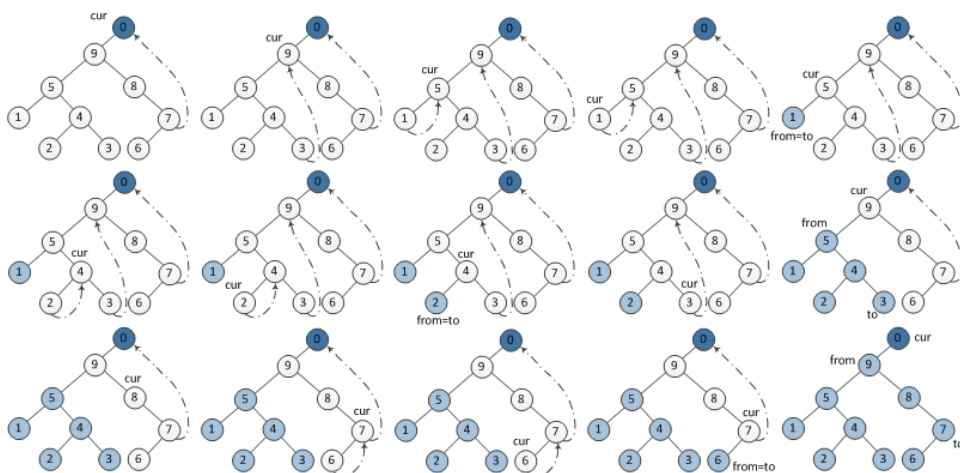


图 4-4 Morris 后序遍历

C 语言实现见第 §4.3.4 节。

4.3.4 C 语言实现

```

/** @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;

/**

```

morris_traversal.cpp

```

    *@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, *prev;

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

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

```

```

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

            if (node->rchild == NULL ) { /* 还没线索化, 则建立线索 */
                visit(cur->data); /* 仅这一行与中序不同 */
                node->rchild = cur;
                prev = cur;
                cur = cur->lchild;
            } else { /* 已经线索化, 则访问节点, 并删除线索 */
                node->rchild = NULL;
                prev = cur;
                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, *prev = NULL;
    while (cur != NULL ) {
        if (cur->lchild == NULL ) {
            prev = cur;
            cur = cur->rchild;
        } else {
            bt_node_t *node = cur->lchild;
            while (node->rchild != NULL && node->rchild != cur)
                node = node->rchild;

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

```

```

        visit_reverse(cur->lchild, prev, visit); // call print
        prev->rchild = NULL;
        prev = cur;
        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

```

#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

```

binary_tree_rebuild.c

```

    */
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 原理和实现

C++ 可以直接使用 `std::priority_queue`。

```

/** @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
 */

```

heap.c

```

#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->elems[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

4.5.2 最小的 N 个和

描述

有两个长度为 N 的序列 A 和 B ，在 A 和 B 中各任取一个数可以得到 N^2 个和，求这 N^2 个和中最小的 N 个。

输入

第一行输入一个正整数 N ；第二行 N 个整数 A_i 且 $A_i \leq 10^9$ ；第三行 N 个整数 B_i ，且 $B_i \leq 10^9$ 。

输出

输出仅一行，包含 N 个整数，从小到大输出这 N 个最小的和，相邻数字之间用空格隔开。

样例输入

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

样例输出

```
2 3 4 4 5
```

分析

由于数据太大，有 N^2 个和，不能通过先求和再排序的方式来求解，这个时候就要用到堆了。

首先将 A, B 两数组排序，我们可以建立这样一个有序表：

$$\begin{aligned} A_1 + B_1 &< A_1 + B_2 < A_1 + B_3 < \dots < A_1 + B_N \\ A_2 + B_1 &< A_2 + B_2 < A_2 + B_3 < \dots < A_2 + B_N \\ &\dots \\ A_N + B_1 &< A_N + B_2 < A_N + B_3 < \dots < A_N + B_N \end{aligned}$$

首先将 $A[i] + B[0]$ 压入堆中，设每次出堆的元素为 $\text{sum} = A[a] + B[b]$ ，则将 $A[a] + B[b+1]$ 入堆，这样可以保证前 N 个出堆的元素为最小的前 N 项。在实现的时候，可以不用保存 B 数组的下标，通过 $\text{sum} - B[b] + B[b+1]$ 来替换 $A[a] + B[b+1]$ 来节省空间。

代码

```
sequence.c
/* wikioi 1245 最小的 N 个和, http://www.wikioi.com/problem/1245/ */
#include <stdio.h>
#include <stdlib.h> /* for malloc() */
#include <string.h> /* for memcpy() */

#define MAXN 100000

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

typedef struct node_t {
    int sum;
    int b; /* sum=a[i]+b[b] */
} node_t;

typedef node_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"

void k_merge() {
    heap_t h;
    int i;
    node_t tmp;

    qsort(a, N, sizeof(int), cmp_int);
    qsort(b, N, sizeof(int), cmp_int);
    heap_init(&h, N, cmp_node);
```

```
    for (i = 0; i < N; i++) {
        tmp.sum = a[i]+b[0];
        tmp.b = 0;
        heap_push(&h, tmp);
    }

    for (i = 0; i < N; i++) {
        tmp = heap_top(&h); heap_pop(&h);
        printf("%d ", tmp.sum);
        tmp.sum = tmp.sum - b[tmp.b] + b[tmp.b + 1];
        tmp.b++;
        heap_push(&h, tmp);
    }

    heap_uninit(&h);
    return;
}

int main() {
    int i;

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

    k_merge();
    return 0;
}
```

sequence.c

相关的题目

与本题相同的题目：

- wikioi 1245 最小的 N 个和, <http://www.wikioi.com/problem/1245/>

与本题相似的题目：

- POJ 2442 Sequence, <http://poj.org/problem?id=2442>

4.6 并查集

4.6.1 原理和实现

通常用树双亲表示作为并查集的存储结构。每个集合以一棵树表示，数组元素的下标代表元素名，根结点的双亲指针为一个负数，表示集合的元素个数。如图 4-5、图 4-6和

图 4-7所示。

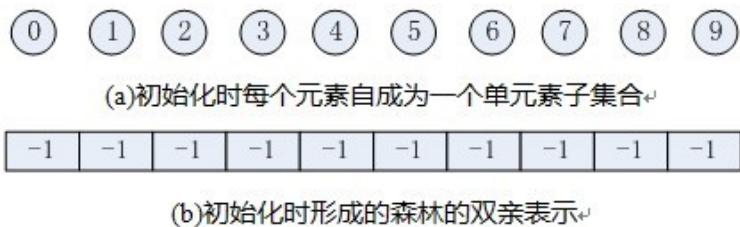


图 4-5 并查集的初始化

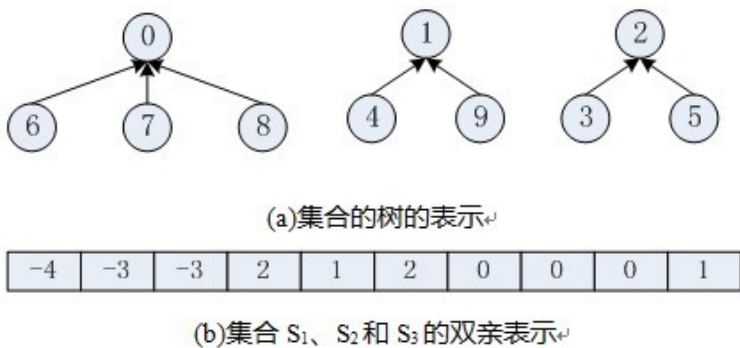


图 4-6 用树表示并查集

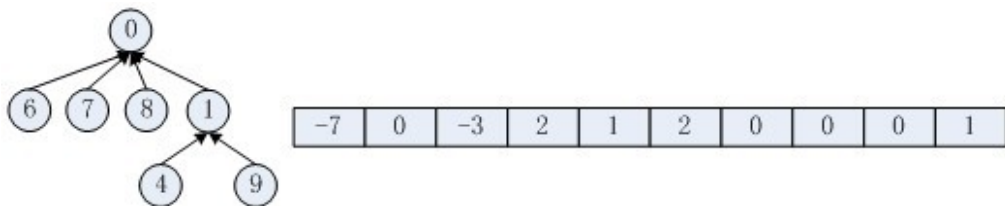


图 4-7 两个集合的并

并查集的 C 语言实现如下。

```

/**
 * @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;
}

```

ufs.c


```
/** Find 操作, 朴素版. */
int ufs_find_naive(const int s[], int x) {
    while (s[x] >= 0) {
        x = s[x];
    }
    return x;
}

/**
 * @brief Find 操作, 带路径压缩.
 * @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(int s[], int x) {
    int oldx = x; /* 记录原始 x */
    while (s[x] >= 0) {
        x = s[x];
    }
    while (oldx != x) {
        int temp = s[oldx];
        s[oldx] = x;
        oldx = temp;
    }
    return x;
}

/* 带路径压缩的 find, 递归版. */
int ufs_find1(int s[], int x) {
    if (s[x] < 0) return x;
    s[x] = ufs_find1(s, s[x]); /* 回溯时的压缩路径 */
    return s[x];
}

/**
 * @brief Union 操作, 将 root2 并入到 root1.
 * @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 如果二者已经在同一集合, 并失败, 返回 0, 否则返回 1
 */
int ufs_union(int s[], int root1, int root2) {
    if (root1 == root2) return 0;
    s[root1] += s[root2];
    s[root2] = root1;
    return 1;
}
```

4.6.2 病毒感染

描述

一个学校有 n 个社团，一个学生能同时加入不同的社团。由于社团内的同学们交往频繁，如果一个学生感染了病毒，该社团的所有学生都会感染病毒。现在 0 号学生感染了病毒，问一共有多少个人感染了病毒。

输入

输入包含多组测试用例。每个测试用例，第一行包含两个整数 n, m ， n 表示学生个数， m 表示社团个数。假设 $0 < n \leq 30000, 0 \leq m \leq 500$ 。每个学生从 0 到 $n - 1$ 编号。接下来是 m 行，每行开头是一个整数 k ，表示该社团的学生个数，接着是 k 个整数表示该社团的学生编号。最后一个测试用例， $n = 0, m = 0$ ，表示输入结束。

输出

对每个测试用例，输出感染了病毒的学生数目。

样例输入

```
100 4
2 1 2
5 10 13 11 12 14
2 0 1
2 99 2
200 2
1 5
5 1 2 3 4 5
1 0
0 0
```

样例输出

```
4
1
1
```

分析

非常简单的并查集题目。

代码

```
/* POJ 1611 The Suspects, http://poj.org/problem?id=1611 */
#include <stdio.h>

#define MAXN 30000
int s[MAXN];

/* 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 "ufs.c" /* 见“树-> 并查集”这节 */

int main() {
    int n, m, k;
    while (scanf("%d%d", &n, &m) && n > 0) {
        ufs_init(s, MAXN);
        while (m--) {
            int x, y; /* 两个学生 */
            int rx, ry; /* x, y 所属的集合的根 */
            scanf("%d", &k);

            k--;
            scanf("%d", &x);
            rx = ufs_find(s, x);
            while (k--) {
                scanf("%d", &y);
                ry = ufs_find(s, y);
                ufs_union(s, rx, ry); /* 只要是跟 x 同一个集合的都并进去 */
            }
            /* 最后搜索 0 属于哪个集合, 这个集合有多少人 */
            printf("%d\n", -s[ufs_find(s, 0)]);
        }
        return 0;
    }
}
```

suspects.c

相关的题目

与本题相同的题目:

- POJ 1611 The Suspects, <http://poj.org/problem?id=1611>

与本题相似的题目:

- None

4.6.3 两个黑帮

描述

Tadu 城市有两个黑帮帮派，已知有 N 黑帮分子，从 1 到 N 编号，每个人至少属于一个帮派。每个帮派至少有一人。给你 M 条信息，有两类信息：

- $D\ a\ b$ ，明确告诉你， a 和 b 属于不同的帮派
- $A\ a\ b$ ，问你， a 和 b 是否属于不同的帮派

输入

第一行是一个整数 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 N 和 M ，接下来是 M 行，每行包含一条消息。

输出

对每条消息“ $A\ a\ b$ ”，基于当前获得的信息，输出判断。答案是“`In the same gang.`”，“`In different gangs.`”和“`Not sure yet.`”中的一个。

样例输入

```
1
5 5
A 1 2
D 1 2
A 1 2
D 2 4
A 1 4
```

样例输出

```
Not sure yet.
In different gangs.
In the same gang.
```

分析

把不在一个集合的节点直接用并查集合在一起。这样的话，如果询问的 2 个节点在同一个并查集里面，那么它们之间的关系是确定的，否则无法确定它们的关系。

现在还有一个问题是，在同一个集合里面的 2 个节点是敌对关系还是朋友关系？可以给每个节点另外附加个信息，记录其距离集合根节点的距离。如果，询问的 2 个节点距离其根节点的距离都是奇数或者都是偶数，那么这 2 个节点是朋友关系，否则是敌对关系。

代码

```
/* POJ 2524 Find them, Catch them, http://poj.org/problem?id=1703 */
#include <stdio.h>

#define MAXN 1000001
int s[MAXN]; /* 0 位置未用 */
char dist[MAXN]; /* 到根节点的距离的奇偶性 */

/**
 * @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;
        dist[i] = 0;
    }
}

/**
 * @brief Find 操作.
 * @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) {
        dist[x] = (dist[x] + dist[s[x]]) % 2;
        x = s[x];
    }
    return x;
}

/**
 * @brief Union 操作, 将 root2 并入到 root1.
 * @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 如果二者已经在同一集合, 并失败, 返回 0, 否则返回 1
 */
int ufs_union(int s[], int root1, int root2) {
    if (root1 == root2) return 0;
    s[root1] += s[root2];
    s[root2] = root1;
    return 1;
}

int main() {
```

two_gangs.c

```

int T;

scanf("%d", &T);
while (T--) {
    int n, m;
    char c;
    int x, y, rx, ry;
    scanf("%d%d%c", &n, &m);
    ufs_init(s, MAXN);

    while (m--) {
        scanf("%c%d%d%c", &c, &x, &y); //注意输入
        rx = ufs_find(s, x);
        ry = ufs_find(s, y);

        if (c == 'A') {
            if (rx == ry) { //如果根节点相同, 则表示能判断关系
                if (dist[x] != dist[y])
                    printf("In different gangs.\n");
                else
                    printf("In the same gang.\n");
            } else
                printf("Not sure yet.\n");
        } else if (c == 'D') {
            ufs_union(s, rx, ry);
            /* ry 与 y 关系 + y 与 x 的关系 + x 与 rx 的关系 = ry 与 rx 的关系 */
            dist[ry] = (dist[y] + 1 + dist[x]) % 2;
        }
    }
}
return 0;
}

```

two_gangs.c

相关的题目

与本题相同的题目:

- POJ 1703 Find them, Catch them, <http://poj.org/problem?id=1703>

与本题相似的题目:

- None

4.6.4 食物链

描述

动物王国中有三类动物 A,B,C, 这三类动物的食物链构成了有趣的环形。A 吃 B, B 吃 C, C 吃 A。现有 N 个动物, 从 1 到 N 编号。每个动物都是 A,B,C 中的一种, 但是我们并不知道它到底是哪一种。

有人用两种说法对这 N 个动物所构成的食物链关系进行描述：

- 第一种说法是“1 X Y”，表示 X 和 Y 是同类。
- 第二种说法是“2 X Y”，表示 X 吃 Y。

此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。

- 当前的话与前面的某些真的话冲突，就是假话；
- 当前的话中 X 或 Y 比 N 大，就是假话；
- 当前的话表示 X 吃 X，就是假话。

你的任务是根据给定的 N ($1 \leq N \leq 50,000$) 和 K 句话 ($0 \leq K \leq 100,000$)，输出假话的总数。

输入

第一行是两个整数 N 和 K ，以一个空格分隔。

以下 K 行每行是三个正整数 D ， X ， Y ，两数之间用一个空格隔开，其中 D 表示说法的种类。

- 若 $D=1$ ，则表示 X 和 Y 是同类。
- 若 $D=2$ ，则表示 X 吃 Y。

输出

只有一个整数，表示假话的数目。

样例输入

```
100 7
1 101 1
2 1 2
2 2 3
2 3 3
1 1 3
2 3 1
1 5 5
```

样例输出

```
3
```

分析

代码

food_chain.c

food_chain.c

相关的题目

与本题相同的题目：

- POJ 1182 食物链, <http://poj.org/problem?id=1182>
- wikioi 1074 食物链, <http://www.wikioi.com/problem/1074/>

与本题相似的题目：

- None

4.7 线段树

4.7.1 原理和实现

线段树，也叫区间树 (interval tree)，它在各个节点保存一条线段（即子数组）。设数列 A 包含 N 个元素，则线段树的根节点表示整个区间 $A[1, N]$ ，左孩子表示区间 $A[1, (1 + N)/2]$ ，右孩子表示区间 $A[(1 + N)/2 + 1, N]$ ，不断递归，直到叶子节点，叶子节点只包含一个元素。

线段树有如下特征：

- 线段树是一棵完全二叉树
- 线段树的深度不超过 $\log L$, L 是区间的长度
- 线段树把一个长度为 L 的区间分成不超过 $2 \log L$ 条线段

线段树的基本操作有构造线段树、区间查询和区间修改。

线段树通常用于解决和区间统计有关的问题。比如某些数据可以按区间进行划分，按区间动态进行修改，而且还需要按区间多次进行查询，那么使用线段树可以达到较快的查询速度。

用线段树解题，关键是要想清楚每个节点要存哪些信息（当然区间起点和终点，以及左右孩子指针是必须的），以及这些信息如何高效查询，更新。不要一更新就更新到叶子节点，那样更新操作的效率最坏有可能 $O(N)$ 的。

4.7.2 Balanced Lineup

描述

给定 $N(1 \leq N \leq 50,000)$ 个数, A_1, A_2, \dots, A_N , 求任意区间中最大数和最小数的差。

输入

第一行包含两个整数, N 和 Q 。 Q 表示查询次数。

第 2 到 $N+1$ 行, 每行包含一个整数 A_i 。

第 $N+2$ 到 $N+Q+1$ 行, 每行包含两个整数 a 和 $b(1 \leq a \leq b \leq N)$, 表示区间 $A[a, b]$ 。

输出

对每个查询进行回应, 输出该区间内最大数和最小数的差

样例输入

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

样例输出

```
6
3
0
```

分析

本题是“区间求和”, 只需要“线段树构造”和“区间查询”两个操作。

代码

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

balanced_lineup.c

```

#define MAXN 50001
#define INF INT_MAX
#define max(a,b) ((a)>(b)?(a):(b))
#define min(a,b) ((a)<(b)?(a):(b))
#define L(a) ((a)<<1)
#define R(a) (((a)<<1)+1)

typedef struct node_t {
    int left, right; /* 区间 */
    int max, min; /* 本区间里的最大值和最小值 */
} node_t;

int A[MAXN]; /* 输入数据, 0 位置未用 */

/* 完全二叉树, 结点编号从 1 开始, 层次从 1 开始.
 * 用一维数组存储完全二叉树, 空间约为 4N,
 * 参考 http://comzyh.tk/blog/archives/479/
 */
node_t node[MAXN * 4];

int minx, maxx; /* 存放查询的结果 */

void init() {
    memset(node, 0, sizeof(node));
}

/* 以 t 为根结点, 为区间 A[l,r] 建立线段树 */
void build(int t, int l, int r) {
    node[t].left = l, node[t].right = r;
    if (l == r) {
        node[t].max = node[t].min = A[l];
        return;
    }
    const int mid = (l + r) / 2;
    build(L(t), l, mid);
    build(R(t), mid + 1, r);
    node[t].max = max(node[L(t)].max, node[R(t)].max);
    node[t].min = min(node[L(t)].min, node[R(t)].min);
}

/* 查询根结点为 t, 区间为 A[l,r] 的最大值和最小值 */
void query(int t, int l, int r) {
    if (node[t].left == l && node[t].right == r) {
        if (maxx < node[t].max)
            maxx = node[t].max;
        if (minx > node[t].min)
            minx = node[t].min;
        return;
    }
    const int mid = (node[t].left + node[t].right) / 2;
    if (l > mid) {
        query(R(t), l, r);
    }

```

```
    } else if (r <= mid) {
        query(L(t), l, r);
    } else {
        query(L(t), l, mid);
        query(R(t), mid + 1, r);
    }
}

int main() {
    int n, q, i;

    scanf("%d%d", &n, &q);
    for (i = 1; i <= n; i++) scanf("%d", &A[i]);

    init();
    /* 建立以 tree[1] 为根结点, 区间为 A[1,n] 的线段树 */
    build(1, 1, n);

    while (q--) {
        int a, b;
        scanf("%d%d", &a, &b);
        maxx = 0;
        minx = INF;
        query(1, a, b); /* 查询区间 A[a,b] 的最大值和最小值 */
        printf("%d\n", maxx - minx);
    }
    return 0;
}
```

balanced_lineup.c

相关的题目

与本题相同的题目:

- POJ 3264 Balanced Lineup, <http://poj.org/problem?id=3264>

与本题相似的题目:

- None

4.7.3 线段树练习 1

描述

一行 $N(1 \leq N < 100000)$ 个方格, 开始每个格子里都有一个整数。现在动态地提出一些命令请求, 有两种命令, 查询和增加: 求某一个特定的子区间 $[a, b]$ 中所有元素的和; 指定某一个格子 x , 加上一个特定的值 A 。现在要求你能对每个请求作出正确的回答。

输入

输入文件第一行为一个整数 N ，接下来是 n 行每行 1 个整数，表示格子中原来的整数。接下来是一个正整数 Q ，再接下来有 Q 行，表示 Q 个询问，第一个整数表示命令代号，命令代号 1 表示增加，后面的两个数 a 和 x 表示给位置 a 上的数值增加 x ，命令代号 2 表示区间求和，后面两个整数 a 和 b ，表示要求 $[a,b]$ 之间的区间和。

输出

共 Q 行，每个整数

样例输入

```
6
4
5
6
2
1
3
4
1 3 5
2 1 4
1 1 9
2 2 6
```

样例输出

```
22
22
```

分析

单点更新 + 区间求和

代码

```
/* wikioi 1080 线段树练习 , http://www.wikioi.com/problem/1080/ */ interval_tree1.c
#include <stdio.h>
#include <string.h>

#define L(a) ((a)<<1)
#define R(a) (((a)<<1)+1)
#define MAXN 100001

typedef long long int64_t;
```

```

typedef struct node_t {
    int left, right;
    int64_t sum;
} node_t;

int A[MAXN]; /* 输入数据, 0 位置未用 */

/* 完全二叉树, 结点编号从 1 开始, 层次从 1 开始.
 * 用一维数组存储完全二叉树, 空间约为 4N,
 * 参考 http://comzyh.tk/blog/archives/479/
 */
node_t node[MAXN * 4];

void init() {
    memset(node, 0, sizeof(node));
}

/* 以 t 为根结点, 为区间 A[l,r] 建立线段树 */
void build(int t, int l, int r) {
    node[t].left = l;
    node[t].right = r;
    if (l == r) {
        node[t].sum = A[l];
        return;
    }
    const int mid = (l + r) / 2;
    build(L(t), l, mid);
    build(R(t), mid + 1, r);
    node[t].sum = node[L(t)].sum + node[R(t)].sum;
}

/* 给区间 A[l,r] 里的 pos 位置加 delta */
void update(int t, int l, int r, int pos, int64_t delta) {
    if (node[t].left > pos || node[t].right < pos) return;
    if (node[t].left == node[t].right) {
        node[t].sum += delta;
        return;
    }
    const int mid = (node[t].left + node[t].right) / 2;
    if (l > mid) update(R(t), l, r, pos, delta);
    else if (r <= mid) update(L(t), l, r, pos, delta);
    else {
        update(L(t), l, mid, pos, delta);
        update(R(t), mid + 1, r, pos, delta);
    }
    node[t].sum = node[L(t)].sum + node[R(t)].sum;
}

/* 查询根结点为 t, 区间为 A[l,r] 的和 */
int64_t query(int t, int l, int r) {
    if (node[t].left == l && node[t].right == r)

```

```

        return node[t].sum;
    const int mid = (node[t].left + node[t].right) / 2;
    if (l > mid) return query(R(t), l, r);
    else if (r <= mid) return query(L(t), l, r);
    else return query(L(t), l, mid) + query(R(t), mid + 1, r);
}

int main() {
    int i, n, q;
    scanf("%d", &n);
    for (i = 1; i <= n; i++) scanf("%d", &A[i]);

    init();
    /* 建立以 tree[1] 为根结点, 区间为 A[1,n] 的线段树 */
    build(1, 1, n);

    scanf("%d", &q);
    while (q--) {
        int cmd;
        scanf("%d", &cmd);
        if (cmd == 2) {
            int a, b;
            scanf("%d%d", &a, &b);
            printf("%lld\n", query(1, a, b)); /* 查询区间 A[a,b] 的和 */
        } else {
            int a;
            int64_t x;
            scanf("%d%lld", &a, &x);
            if (x != 0) update(1, 1, n, a, x);
        }
    }
    return 0;
}

```

interval_tree1.c

相关的题目

与本题相同的题目：

- wikioi 1080 线段树练习 1, <http://www.wikioi.com/problem/1080/>

与本题相似的题目：

- wikioi 1081 线段树练习 2, <http://www.wikioi.com/problem/1081/>。本题是“区间更新 + 单点查询”，可以转化为线段树练习 1。设原数组为 $A[N]$ ，将其转化为差分数列，然后在数组上维护一棵线段树。“区间更新”操作转化为两个“单点更新”操作：将 $A[a]$ 加上 x ，并将 $A[b + 1]$ 减去 x （也就是加上 $-x$ ）。“单点查询”操作转化为“区间求和”操作：求 A 数组 $[1..i]$ 范围内所有数的和。这样就转化成与线段树练习 1 完全相同了。标程 <https://gist.github.com/soulmachine/6449609>

4.7.4 A Simple Problem with Integers

描述

You have N integers, A_1, A_2, \dots, A_N . You need to deal with two kinds of operations. One type of operation is to add some given number to each number in a given interval. The other is to ask for the sum of numbers in a given interval.

输入

The first line contains two numbers N and Q . $1 \leq N, Q \leq 100000$.

The second line contains N numbers, the initial values of A_1, A_2, \dots, A_N . $-1000000000 \leq A_i \leq 1000000000$.

Each of the next Q lines represents an operation. "C a b c" means adding c to each of A_a, A_{a+1}, \dots, A_b . $-10000 \leq c \leq 10000$. "Q a b" means querying the sum of A_a, A_{a+1}, \dots, A_b .

输出

You need to answer all Q commands in order. One answer in a line.

样例输入

```
10 5
1 2 3 4 5 6 7 8 9 10
Q 4 4
Q 1 10
Q 2 4
C 3 6 3
Q 2 4
```

样例输出

```
4
55
9
15
```

提示

The sums may exceed the range of 32-bit integers.

分析

区间更新 + 区间求和。

树节点要存哪些信息？只存该区间的和，行不行？只存和，会导致每次加数的时候都要更新到叶子节点，速度太慢。本题节点的结构如下：

```
typedef struct node_t {
    int left, right;
    int64_t sum; /* 本区间的和实际上是 sum+inc*[right-left+1] */
    int64_t inc; /* 增量 c 的累加 */
} node_t;
```

代码

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

#define L(a) ((a)<<1)
#define R(a) (((a)<<1)+1)
#define MAXN 100001

typedef long long int64_t;

typedef struct node_t {
    int left, right;
    int64_t sum; /* 本区间的和实际上是 sum+inc*[right-left+1] */
    int64_t inc; /* 增量 c 的累加 */
} node_t;

int A[MAXN]; /* 输入数据, 0 位置未用 */

/* 完全二叉树, 结点编号从 1 开始, 层次从 1 开始.
 * 用一维数组存储完全二叉树, 空间约为 4N,
 * 参考 http://comzyh.tk/blog/archives/479/
 */
node_t node[MAXN * 4];

void init() {
    memset(node, 0, sizeof(node));
}

/* 以 t 为根结点, 为区间 A[l,r] 建立线段树 */
void build(int t, int l, int r) {
    node[t].left = l;
    node[t].right = r;
    if (l == r) {
        node[t].sum = A[l];
        return;
    }
    const int mid = (l + r) / 2;
    build(L(t), l, mid);
```

poj3468.c


```

        build(R(t), mid+1, r);
        node[t].sum = node[L(t)].sum + node[R(t)].sum;
    }

    /* 给区间 A[l,r] 里的每个元素都加 c */
    void update(int t, int l, int r, int64_t c) {
        if (node[t].left == l && node[t].right == r) {
            node[t].inc += c;
            node[t].sum += c * (r - l + 1);
            return;
        }
        if (node[t].inc) {
            node[R(t)].inc += node[t].inc;
            node[L(t)].inc += node[t].inc;
            node[R(t)].sum += node[t].inc * (node[R(t)].right - node[R(t)].left + 1);
            node[L(t)].sum += node[t].inc * (node[L(t)].right - node[L(t)].left + 1);
            node[t].inc = 0;
        }
        const int mid = (node[t].left + node[t].right) / 2;
        if (l > mid)
            update(R(t), l, r, c);
        else if (r <= mid)
            update(L(t), l, r, c);
        else {
            update(L(t), l, mid, c);
            update(R(t), mid + 1, r, c);
        }
        node[t].sum = node[L(t)].sum + node[R(t)].sum;
    }

    /* 查询根结点为 t, 区间为 A[l,r] 的和 */
    int64_t query(int t, int l, int r) {
        if (node[t].left == l && node[t].right == r)
            return node[t].sum;
        if (node[t].inc) {
            node[R(t)].inc += node[t].inc;
            node[L(t)].inc += node[t].inc;
            node[R(t)].sum += node[t].inc * (node[R(t)].right - node[R(t)].left + 1);
            node[L(t)].sum += node[t].inc * (node[L(t)].right - node[L(t)].left + 1);
            node[t].inc = 0;
        }
        const int mid = (node[t].left + node[t].right) / 2;
        if (l > mid)
            return query(R(t), l, r);
        else if (r <= mid)
            return query(L(t), l, r);
        else
            return query(L(t), l, mid) + query(R(t), mid + 1, r);
    }

    int main() {
        int i, n, q;
        char s[5];
    }

```

```

scanf("%d%d", &n, &q);
for (i = 1; i <= n; i++) scanf("%d", &A[i]);

init();
/* 建立以 tree[1] 为根结点, 区间为 A[1,n] 的线段树 */
build(1, 1, n);

while (q--) {
    int a, b;
    int64_t c;
    scanf("%s", s);
    if (s[0] == 'Q') {
        scanf("%d%d", &a, &b);
        printf("%lld\n", query(1, a, b)); /* 查询区间 A[a,b] 的和 */
    } else {
        scanf("%d%d%lld", &a, &b, &c);
        if (c != 0) update(1, a, b, c);
    }
}
return 0;
}

```

poj3468.c

相关的题目

与本题相同的题目:

- POJ 3468 A Simple Problem with Integers, <http://poj.org/problem?id=3468>

与本题相似的题目:

- None

4.7.5 约瑟夫问题

描述

有编号从 1 到 N 的 N 个小朋友在玩一种出圈的游戏。开始时 N 个小朋友围成一圈, 编号为 $i+1$ 的小朋友站在编号为 i 小朋友左边。编号为 1 的小朋友站在编号为 N 的小朋友左边。首先编号为 1 的小朋友开始报数, 接着站在左边的小朋友顺序报数, 直到数到某个数字 M 时就出圈。直到只剩下 1 个小朋友, 则游戏完毕。

现在给定 N, M , 求 N 个小朋友的出圈顺序。

输入

唯一的一行包含两个整数 $N, M (1 \leq N, M \leq 30000)$ 。

输出

唯一的一行包含 N 个整数，每两个整数中间用空格隔开，第 i 个整数表示第 i 个出圈的小朋友的编号。

样例输入

5 3

样例输出

3 1 5 2 4

分析

约瑟夫问题的难点在于，每一轮都不能通过简单的运算得出下一轮谁淘汰，因为中间有人已经退出了。因此一般只能模拟，效率很低。

现在考虑，每一轮都令所有剩下的人从左到右重新编号，例如 3 退出后，场上还剩下 1、2、4、5，则给 1 新编号 1，2 新编号 2，4 新编号 3，5 新编号 4。不妨称这个编号为“剩余队列编号”。如下所示，括号内为原始编号：

```
1(1) 2(2) 3(3) 4(4) 5(5) --> 剩余队列编号 3 淘汰，对应原编号 3
1(1) 2(2) 3(4) 4(5) --> 剩余队列编号 1 淘汰，对应原编号 1
1(2) 2(4) 3(5) --> 剩余队列编号 3 淘汰，对应原编号 5
1(2) 2(4) --> 剩余队列编号 1 淘汰，对应原编号 2
1(4) --> 剩余队列编号 1 滔天，对应原编号 4
```

一个人在当前剩余队列中编号为 i ，则说明他是从左到右数第 i 个人，这启发我们可以用线段树来解决问题。用线段树维护原编号 $[i..j]$ 内还有多少人没有被淘汰，这样每次选出被淘汰者后，在当前线段树中查找位置就可以了。

例如我们有 5 个原编号，当前淘汰者在剩余队列中编号为 3，先看左子树，即原编号 $[1..3]$ 区间内，如果剩下的人不足 3 个，则说明当前剩余编号为 3 的这个人原编号只能是在 $[4..5]$ 区间内，继续在 $[4..5]$ 上搜索；如果 $[1..3]$ 内剩下的人大于等于 3 个，则说明就在 $[1..3]$ 内，也继续缩小范围查找，这样即可在 $O(\log N)$ 时间内完成对应。问题得到圆满的解决。

代码

```
/* wikioi 1282 约瑟夫问题, http://www.wikioi.com/problem/1282/ */
#include <stdio.h>
#include <string.h>

#define L(a) ((a)<<1)
```

josephus_problem.c

```

#define R(a) (((a)<<1)+1)
#define MAXN 30001

typedef struct node_t {
    int left, right;
    int count; /* 区间内的元素个数 */
} node_t;

/* 完全二叉树, 结点编号从 1 开始, 层次从 1 开始.
 * 用一维数组存储完全二叉树, 空间约为 4N,
 * 参考 http://comzyh.tk/blog/archives/479/
 */
node_t node[MAXN * 4];

void init() {
    memset(node, 0, sizeof(node));
}

/* 以 t 为根结点, 为区间 [l,r] 建立线段树 */
void build(int t, int l, int r) {
    node[t].left = l;
    node[t].right = r;
    node[t].count = r - l + 1;
    if (l == r) return;

    const int mid = (r + l) / 2;
    build(L(t), l, mid);
    build(R(t), mid + 1, r);
}

/**
 * @brief 输出 i
 * @param[in] t 根节点
 * @param[in] i 剩余队列编号
 * @return 被删除的实际数字
 */
int delete(int t, int i) {
    node[t].count--;
    if (node[t].left == node[t].right) {
        printf("%d ", node[t].left);
        return node[t].left;
    }
    if (node[L(t)].count >= i) return delete(L(t), i);
    else return delete(R(t), i - node[L(t)].count); /* 左子树人数不足, 则在右子树查找 */
}

/**
 * @brief 返回 1 到 i 内的活人数
 * @param[in] t 根节点
 * @param[in] i 原始队列的数字
 * @return 1 到 i 内的活人数
 */
int get_count(int t, int i) {

```

```
    if (node[t].right <= i) return node[t].count;

    const int mid = (node[t].left + node[t].right) / 2;
    int s = 0;
    if (i > mid) {
        s += node[L(t)].count;
        s += get_count(R(t), i);
    } else
        s += get_count(L(t), i);
    return s;
}

int main() {
    int n, m;
    scanf("%d%d", &n, &m);

    init();
    build(1, 1, n);

    int i;
    int j = 0; /* 剩余队列的虚拟编号 */
    for (i = 1; i <= n; i++) {
        j += m;
        if (j > node[1].count)
            j %= node[1].count;
        if (j == 0) j = node[1].count;
        const int k = delete(1, j);
        j = get_count(1, k);
    }
    return 0;
}
```

josephus_problem.c

相关的题目

与本题相同的题目：

- wikioi 1282 约瑟夫问题, <http://www.wikioi.com/problem/1282/>

与本题相似的题目：

- None

第 5 章

查找

5.1 折半查找

```
/** 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。

```
/** 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。

```

/**
 * @brief 折半插入排序，时间复杂度  $O(n\log 2n)$ .
 *
 * @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;
    }
}

```

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。

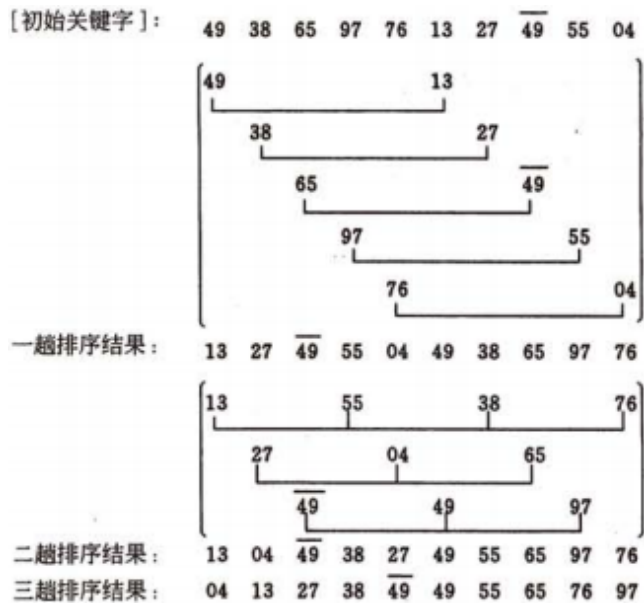


图 6-1 希尔排序

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。

```

/** 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

```

bubble_sort.c

```
*/  
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。

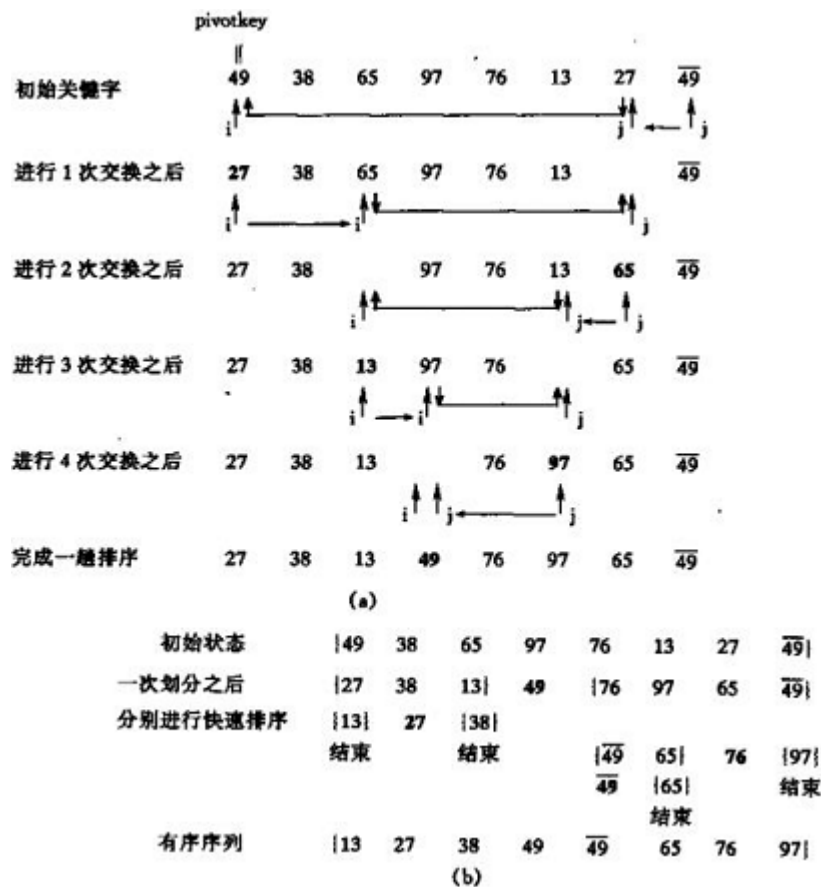


图 6-2 快速排序示例

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;
```

```

    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 $n-i$ ($i=1, 2, \dots, n-2$) 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 $a[i] \dots a[n-1]$ 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 $a[i+1] \dots a[n-1]$ 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。

```

/** 数组元素的类型 */
typedef int elem_t;

```

simple_selection_sort.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。

```

#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;

```

heap_sort.c

```

    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。

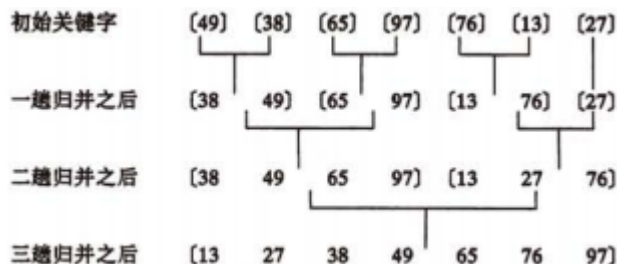


图 6-3 二路归并排序示例

二路归并排序的 C 语言实现如下。

```

/** 数组元素的类型 */
typedef int elem_t;

/** 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

```

merge_sort.c

```

    * @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);
    }
}

```

merge_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。

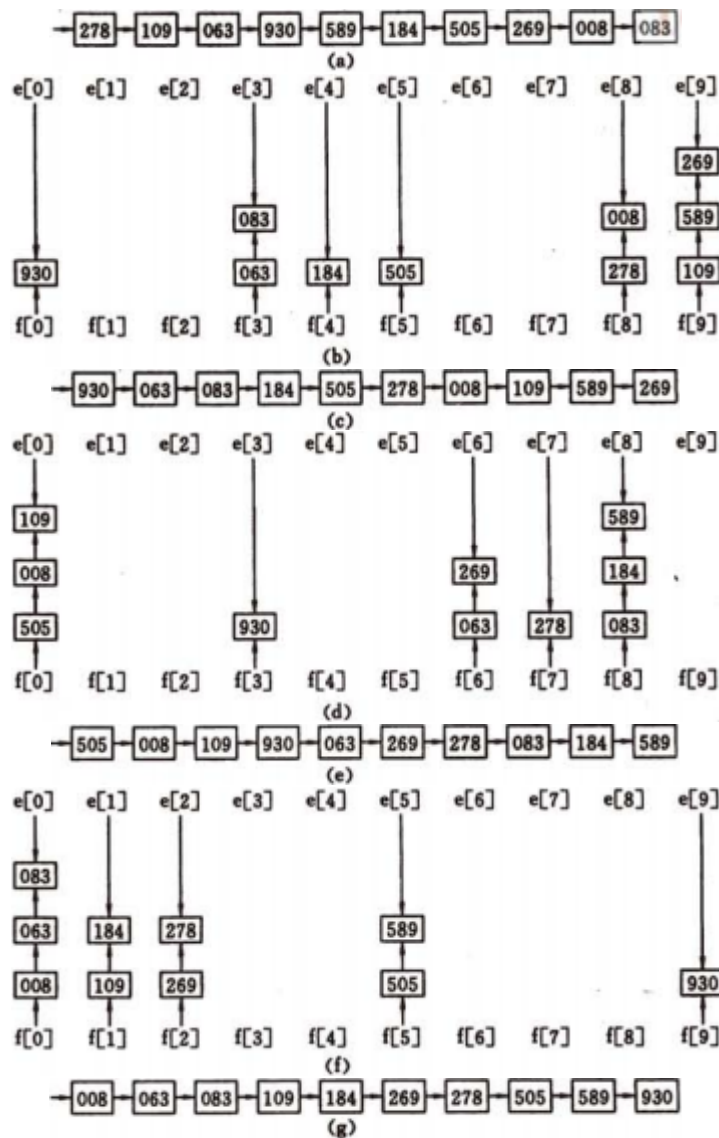


图 6-4 LSD 链式基数排序示例

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
    */
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);
}

```

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

暴力枚举法 (brute force enumeration) 又称为暴力搜索法 (Brute-force search), 详细定义见 Wikipedia, http://en.wikipedia.org/wiki/Brute_force_search

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

#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); // 不能正确处理可重复集
    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

```

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

```

print_permutation.c


```

/*
 * @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;
}

```

7.1.3 下一个排列

还可以利用 STL 中的 `std::next_permutation()` 或第 §17.1 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

```

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

```

subset.c

```

/**
 * @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

```

/**
 * @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

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 B_1 和 B_2 ，则 $B_1|B_2$, $B_1\&B_2$, $B_1\sim B_2$ 分别对应集合的并、交、对称差。

代码

```

/**
 * @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
 */

```

subset.c

```
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;
}
```

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 $2^k \times 2^k$ ($1 \leq k \leq 100$) 的棋盘上，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)，去覆盖棋盘上所有的白色方格，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。

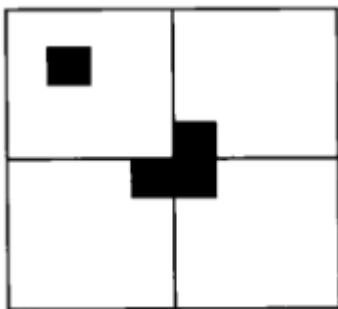


图 8-2 棋盘覆盖问题的递归解法

本题只要求总数，不要求具体怎么摆放，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 §15.3 节。

代码

```
#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.
```

chessboard_cover.c

```

* @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

```
#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

```
// 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
```

poj1521_entropy.cpp

```

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

^②CLRS, 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”和“自底向上 (bottom-up)+ 表格”。前者属于一种 memorization（翻译为记忆化搜索），后者才是正宗的动规。

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 动规和记忆化搜索的区别

动规 (dynamic programming) 一定是自底向上的, 记忆化搜索 (memorization) 一定是自顶向下的。

动规不是 lazy 的, memorization 是 lazy 的, 是按需 (on-demand) 计算的。所以, 如果所有的子问题至少会碰到一次, 则动规有优势; 如果有些子问题在搜索过程中不会被碰到 (即有剪枝), 则 memorization 有优势。更详细的解释请参考 StackOverflow 上的这个帖子 <http://t.cn/z80ZS6B>。

记忆化搜索可以实现跟动规类似的功能, 但它不是动规。两者的方向是反的, 一个是自顶向下, 一个自底向上, 我们应该区分开这两个概念。本书后面提到的动规, 都是指自底向上的动规。

10.2 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 $X = (x_1, x_2, \dots, x_m)$, 则另一个序列 $Z = (z_1, z_2, \dots, z_k)$, 是 X 的子序列是指存在一个严格递增下标序列 (i_1, i_2, \dots, i_k) 使得对于所有 $j = 1, 2, \dots, k$ 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 $X = (x_1, x_2, \dots, x_m)$ 和 $Y = (y_1, y_2, \dots, y_n)$ 的最长公共子序列为 $Z = (z_1, z_2, \dots, z_k)$, 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, $X_{m-1} = (x_1, x_2, \dots, x_{m-1})$, $Y_{n-1} = (y_1, y_2, \dots, y_{n-1})$, $Z_{k-1} = (z_1, z_2, \dots, z_{k-1})$ 。

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:

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

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

```
#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++) {
```

lcs.c

```

        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>

10.3 最大连续子序列和

描述

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

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。

^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

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
```

分析

当我们从头到尾遍历这个数组的时候，对于数组里的一个整数，它有几种选择呢？它只有两种选择：1、加入之前的 SubArray；2、自己另起一个 SubArray。那什么时候会出现这两种情况呢？

如果之前 SubArray 的总体和大于 0 的话，我们认为其对后续结果是有贡献的。这种情况下我们选择加入之前的 SubArray

如果之前 SubArray 的总体和为 0 或者小于 0 的话，我们认为其对后续结果是没有贡献，甚至是有害的（小于 0 时）。这种情况下我们选择以这个数字开始，另起一个 SubArray。

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：

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

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

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

mcSS.C

```

#define max(a,b) ((a)>(b)?(a):(b))
#define min(a,b) ((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, result, cur_min;
    int *sum = (int*) malloc((n + 1) * sizeof(int)); // 前 n 项和

    sum[0] = 0;
    result = INT_MIN;
    cur_min = sum[0];
    for (i = 1; i <= n; i++) { /* 变成前缀和 */
        sum[i] = sum[i - 1] + S[i - 1];
    }
    for (i = 1; i <= n; i++) {
        result = max(result, sum[i] - cur_min);
        cur_min = min(cur_min, sum[i]);
    }
    free(sum);
    return result;
}

/**
 * @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 §10.4 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：

- LeetCode Maximum Subarray, http://leetcode.com/onlinejudge#question_53

与本题相似的题目：

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

10.4 最大 M 子段和

描述

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

现给定一个正整数 m ，找出 m 对 i 和 j ，使得 $sum(i_1, j_1) + sum(i_2, j_2) + \dots + sum(i_m, j_m)$ 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:

$$d[i, j] = \max \{d[i, j-1] + S[j], \max \{d[i-1, t] + S[j]\}\}, \text{ 其中 } i \leq j \leq n, i-1 \leq t < j$$

$$target = \max \{d[m, j]\}, \text{ 其中 } m \leq j \leq n$$

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

```
#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) {
```

mmss.c


```
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>

10.5 背包问题

背包问题 (Knapsack problem^①) 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。

样例输入

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

样例输出

```
14
```

^①Knapsack problem, http://en.wikipedia.org/wiki/Knapsack_problem

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 10-1 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	j= 1	1	0	0	0	4	4	4	4	4	4	4	4
4	5	i-n 2	2	0	0	0	4	5	5	5	9	9	9	9
5	6	3w 3	3	0	0	0	4	5	6	6	9	10	11	11

图 10-1 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

```
#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];
            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() {
```

^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

```

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。

```

#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 */

    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);

```

01knapsack2.c

```
        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
- HDU 2602 Bone Collector, <http://acm.hdu.edu.cn/showproblem.php?pid=2602>

与本题相似的题目：

- wikioi 1014 装箱问题, <http://www.wikioi.com/problem/1014/>,
参考代码 <https://gist.github.com/soulmachine/6248518>

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, 储钱罐的重量不会超过 10kg, 即 $1 \leq E \leq F \leq 10000$ 。第二行是一个整数 N ($1 \leq N \leq 500$), 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 ($1 \leq v \leq 50000, 1 \leq w \leq 10000$), 分别表示硬币的面值和重量。

^①刘汝佳, 算法竞赛入门经典, 清华大学出版社, 2009

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 $\sum \frac{W}{w[i]}$ 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:

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

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 $2^k * w[i] \leq W$ 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 $1, 2, 4, \dots, 2^k$ 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：

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

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 $1, 2, 2^2, \dots, 2^{k-1}, c[i] - 2^k + 1$ ，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

```

#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;
}
```

bkc.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：

- None

10.6 序列型动态规划

对于所有动规题目，如果把状态转移图画出来，一定是一个有向无环图 (DAG)。再进一步细分类别，有序列型动态规划，棋盘型动态规划，树型动态规划等等。

10.6.1 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, \dots, a_N) ，我们可以得到一些上升的子序列 $(a_{i1}, a_{i2}, \dots, a_{iK})$ ，这里 $1 \leq i1 < i2 < \dots < iK \leq N$ 。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。

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

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

int N;
int a[MAXN];
```

lis.c

```
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>
- wikioi 1576, 最长严格上升子序列, <http://www.wikioi.com/problem/1576/>

与本题相似的题目：

- wikioi 1044 拦截导弹, <http://www.wikioi.com/problem/1044/>,

参考代码 <https://gist.github.com/soulmachine/6248489>

10.6.2 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 90° ）。例如 (1,5) 可

^①李文新, 程序设计导引及在线实践, 清华大学出版社, 2007

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整数 n ,表示该组测试数据中含有矩形的个数 ($n \leq 1000$) 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:

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

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 记忆化搜索.
 * @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：

- None

10.6.3 线段覆盖 2

描述

数轴上有 n ($n \leq 1000$) 条线段，线段的两端都是整数坐标，坐标范围在 $0 \sim 1000000$ ，每条线段有一个价值，请从 n 条线段中挑出若干条线段，使得这些线段两两不覆盖（端点可以重合）且线段价值之和最大。

^①刘汝佳, 算法竞赛入门经典, 清华大学出版社, 2009

输入

第一行一个整数 n ，表示有多少条线段。

接下来 n 行每行三个整数 a_i, b_i, c_i ，分别代表第 i 条线段的左端点 a_i ，右端点 b_i （保证左端点 < 右端点）和价值 c_i 。

输出

输出能够获得的最大价值

样例输入

```
3
1 2 1
2 3 2
1 3 4
```

样例输出

```
4
```

分析

先将线段排序。按照右端点从小到大排序。原因是循环结构中是 i 从 1 到 n , i 比较小的时候尽可能选右端点比较小的，这样才可以为后面的线段留下更大的空间。

设状态为 $f[i]$ ，表示前 i 条线段时，选上第 i 条线段，能获得的最大价值。状态转移方程如下：

$$f[i] = \max \{f[j]\} + c[i], 2 \leq i \leq n, 1 \leq j \leq i-1, \text{且 } b[j] \leq a[i]$$

$b[j] \leq a[i]$ 表示 j 的右端点在 i 的左端点左边，即不重合。

输出 $f[i]$ 数组中的最大值。

代码

```
/* http://www.wikioi.com/problem/3027/ */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define MAXN 1000

typedef struct line_t {
    int a, b, c;
```

```
} line_t;

int line_cmp(const void *a, const void *b) {
    line_t *la = (line_t*)a;
    line_t *lb = (line_t*)b;
    return la->b - lb->b;
}

int n;
line_t line[MAXN];
int f[MAXN];

void dp() {
    int i, j;
    qsort(line, n, sizeof(line_t), line_cmp);
    f[0] = line[0].c;

    for (i = 1; i < n; i++) {
        int max = 0;
        for (j = 0; j < i; j++) {
            if (line[j].b <= line[i].a) max = max > f[j] ? max : f[j];
        }
        f[i] = max + line[i].c;
    }
}

static int max_element(const int a[], int begin, int end) {
    int i;
    int max_value = INT_MIN;
    int max_pos = -1;
    for (i = begin; i < end; i++) {
        if (max_value < a[i]) {
            max_value = a[i];
            max_pos = i;
        }
    }
    return max_pos;
}

int main() {
    int i;
    scanf("%d", &n);
    for (i = 0; i < n; i++) scanf("%d%d%d", &line[i].a, &line[i].b, &line[i].c);

    dp();

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

相关的题目

与本题相同的题目：

- wikioi 3027 线段覆盖 2, <http://www.wikioi.com/problem/3027/>

与本题相似的题目：

- None

10.6.4 硬币问题

描述

有 n 种硬币，面值为别为 $v_1, v_2, v_3, \dots, v_n$ ，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。 $1 \leq n \leq 100, 1 \leq S \leq 10000, 1 \leq v_i \leq S$ 。

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：

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

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，记忆化搜索。

coin_change.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 max
 * @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：

- None

^①刘汝佳, 算法竞赛入门经典, 清华大学出版社, 2009

10.7 区间型动态规划

10.7.1 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, \dots, 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}, \dots, 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

```
#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/zH2pchg>
- ZOJ 1276 Optimal Array Multiplication Sequence, <http://t.cn/zH2gW1P>

与本题相似的题目：

- wikioi 1154, 能量项链, <http://www.wikioi.com/problem/1154>,
参考代码 <https://gist.github.com/soulmachine/6248459>
- Matrix67 - 十个利用矩阵乘法解决的经典题目, <http://www.matrix67.com/blog/archives/276>

10.7.2 石子合并

描述

在一条直线上摆着 N 堆石子。现要将石子有次序地合并成一堆。规定每次只能选相邻的 2 堆石子合并成新的一堆，并将新的一堆石子数记为该次合并的得分。

试设计一个算法，计算出将 N 堆石子合并成一堆的最小得分。

输入

第一行是一个数 $N, 1 \leq N \leq 40000$ ，表示堆数。

接下来的 N 行每行一个数，表示该堆的石子数目。

输出

一个整数，即 N 堆石子合并成一堆的最小得分。

样例输入

```
4
1
1
1
1
```

样例输出

```
8
```

分析

这题与前面的矩阵链乘非常相似，只是计算代价的方式不同。设状态为 $d(i, j)$ ，表示子问题第 i 堆到第 j 堆石子的最优解，状态转移方程如下：

$$d(i, j) = \min \{d(i, k) + d(k + 1, j) + \text{sum}(i, j)\}$$

$\text{sum}(i, j)$ 表示从第 i 堆到第 j 堆的石子总数。代码见 <https://gist.github.com/soulmachine/6195139>

上面的动规算法可以用四边形不等式进行优化，将时间复杂度从 $O(N^3)$ 下降到 $O(N^2)$ 。代码见 <https://TODO>

无论如何，时间复杂度都是 $O(N^2)$ ，本题中 N 范围特别大，开不了这么大的二维数组。所以动规算法只能处理小规模的情况。下面介绍第三种解法，Garsia-Wachs 算法^①。

^①Donald E. Knuth, The art of computer programming, volume 3, 2nd ed, p446

假设我们只对 3 堆石子 a,b,c 进行合并, 先合并哪两堆, 能使得得分最小? 有两种方案:

$\text{score1} = (a+b) + ((a+b)+c)$

$\text{score2} = (b+c) + ((b+c)+a)$

假设 $\text{score1} \leq \text{score2}$, 可以推导出 $a \leq c$, 反过来, 只要 a 和 c 的关系确定, 合并的顺序也确定。这就是 2-递减性质。

Garsia-Wachs 算法, 就是基于这个性质, 找出序列中满足 $A[i-1] \leq A[i+1]$ 最小的 i, 合并 $\text{temp} = A[i] + A[i+1]$, 接着往前面找是否有满足 $A[j] > \text{temp}$, 把 temp 值插入 $A[j]$ 的后面 (数组的右边)。循环这个过程一直到只剩下一堆石子结束。

例如,

13 9 5 7 8 6 14

首先, 找第一个满足 $A[i-1] \leq A[i+1]$ 的三个连续点, 本例中就是 5 7 8, 5 和 7 合并得到 12, 12 不是丢在原来的位置, 而是将它插入到第一个比它大的元素的后面, 得到

13 12 9 8 6 14

接着来, 从左到右搜索三个连续点, 找到 8 6 14, 合并 8 和 6 的到 14, 将 14 插入到适当位置, 得到 14 13 12 9 14

找到 12 9 14, 合并 12 和 9 得到 21, 将 21 插入到适当位置, 得到 21 14 13 14

找到 14 13 14, 合并 14 和 13 得到 27, 将 27 插入到适当位置, 得到 27 21 14

因为 $27 < 14$, 先合并 21 和 14, 得到 35, 最后合并 27 和 35, 得到 62, 就是最终答案。

为什么要将 temp 插入 $A[j]$ 的后面, 可以理解为是为了保持 2-递减性质。从 $A[j+1]$ 到 $A[i-2]$ 看成一个整体 $A[\text{mid}]$, 现在 $A[j]$, $A[\text{mid}]$, $\text{temp}(A[i-1]+A[i])$, 因为 $\text{temp} < A[j]$, 因此不管怎样都是 $A[\text{mid}]$ 和 temp 先合并, 所以将 temp 值插入 $A[j]$ 的后面不会影响结果。

代码

```
#include <stdio.h>
#define MAXN 55555

int N, A[MAXN]; /* 堆数, 每堆的石头个数 */
int num, result; /* 数组实际长度, 结果 */

void combine(int k) { /* 前提 A[k-1] < A[k+1] */
    int i, j;

    /* 合并 k-1 和 k */
    int temp=A[k] + A[k-1];
    result += temp;
    /* 紧缩 */
    for(i = k; i < num - 1; i++) A[i] = A[i+1];
    num--;
    /* 插入 temp 到合适位置 */
}
```

stone_merge.c


```
for(j = k-1; j > 0 && A[j-1] < temp; j--) A[j] = A[j-1];
A[j] = temp;

/* why? */
while(j >= 2 && A[j] >= A[j-2]) {
    int d = num - j;
    combine(j - 1);
    j = num - d;
}

int main() {
    int i;

    scanf("%d",&N);
    if(N == 0) return 0;
    for(i = 0; i < N; i++) scanf("%d", &A[i]);

    num=2;
    result=0;
    for(i = 2; i < N; i++) {
        A[num++] = A[i];
        while(num >= 3 && A[num-3] <= A[num-1]) combine(num-2);
    }
    while(num > 1) combine(num-1);
    printf("%d\n",result);
    return 0;
}
```

stone_merge.c

相关的题目

与本题相同的题目：

- WIKIOI 1048 石子归并（数据规模很小），<http://www.wikioi.com/problem/1048/>
- WIKIOI 2298 石子合并（数据规模很大），<http://www.wikioi.com/problem/2298/>
- POJ 1738 An old Stone Game, <http://poj.org/problem?id=1738>

与本题相似的题目：

- None

10.7.3 矩阵取数游戏

描述

帅帅经常跟同学玩一个矩阵取数游戏：对于一个给定的 $n * m$ 的矩阵，矩阵中的每个元素 a_{ij} 均为非负整数。游戏规则如下：

- 每次取数时须从每行各取走一个元素，共 n 个， m 次后取完矩阵所有元素；
- 每次取数时须从每行各取走一个元素，共 n 个， m 次后取完矩阵所有元素；每次取走的各个元素只能是该元素所在行的行首或行尾；
- 每次取数时须从每行各取走一个元素，共 n 个， m 次后取完矩阵所有元素；每次取数都有一个得分值，为每行取数的得分之和，每行取数的得分 = 被取走的元素值 $*2i$ ，其中 i 表示第 i 次取数（从 1 开始编号）；
- 每次取数时须从每行各取走一个元素，共 n 个， m 次后取完矩阵所有元素；游戏结束总得分为 m 次取数得分之和。

帅帅想请你帮忙写一个程序，对于任意矩阵，可以求出取数后的最大得分。

输入

第 1 行为两个用空格隔开的整数 n 和 m , $1 \leq n, m \leq 90$ 。第 2 ~ $n+1$ 行为 $n * m$ 矩阵，其中每行有 m 个用单个空格隔开的非负整数 a_{ij} , $0 \leq a_{ij} \leq 1000$ 。

输出

输出仅包含 1 行，为一个整数，即输入矩阵取数后的最大得分。

样例输入

```
2 3
1 2 3
3 4 2
```

样例输出

```
82
```

样例解释

第 1 次：第 1 行取行首元素，第 2 行取行尾元素，本次得分为 $1*21+2*21=6$

第 2 次：两行均取行首元素，本次得分为 $2*22+3*22=20$

第 3 次：得分为 $3*23+4*23=56$ 。总得分为 $6+20+56=82$

分析

首先，每行之间是互相独立的，因此可以分别求出每行的最大得分，最后相加起来。
设状态为 $f(i, j)$ ，表示单行的区间 $[i, j]$ 的最大值，转移方程为

$$f(i, j) = \max \{f(i+1, j) + a(i) * 2^x, f(i, j-1) + a(j) * 2^x\}, 1 \leq x \leq m$$

等价于

$$f(i, j) = 2 * \max \{f(i+1, j) + a(i), f(i, j-1) + a(j)\}$$

同时，注意到，最大得分可能会非常大，因此需要用到大整数运算。预估一下最大得分，大概是 $1000 * 2^{80}$ ，用计算器算一下，有 28 位。

代码

wiki01166_matrix_game.c

```
#include <stdio.h>
#include <memory.h>
#include <limits.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_RADIX 10000
#define RADIX_LEN 4
/* 1000*2^80 有 28 位 */
#define MAX_LEN (28/RADIX_LEN+1) /* 整数的最大位数 */

/**
 * @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 一个 32 位整数转化为大整数 bigint.
 */
```

```

* @param[in] n 32 位整数
* @param[out] x 大整数
* @return 大整数
*/
int* int_to_bigint(int n, int x[]) {
    int i = 0;
    memset(x, 0, MAX_LEN * sizeof(int));
    while (n > 0) {
        x[i++] = n % BIGINT_RADIX;
        n /= BIGINT_RADIX;
    }
    return x;
}

/**
* @brief 大整数比较.
* @param[in] x x
* @param[in] y y
* @return x<y, 返回-1, 相等于返回 0, 大于返回 1.
*/
int bigint_cmp(const int x[], const int y[]) {
    int i, lenx = 0, leny = 0;
    for (i = 0; i < MAX_LEN; i++) {
        if (x[i] != 0) lenx++;
        else break;
    }
    for (i = 0; i < MAX_LEN; i++) {
        if (y[i] != 0) leny++;
        else break;
    }
    if (lenx < leny) return -1;
    else if (lenx > leny) return 1;
    else {
        for (i = lenx - 1; i >= 0; i--) {
            if (x[i] == y[i]) continue;
            else if (x[i] > y[i]) return 1;
            else return -1;
        }
        return 0;
    }
}

/**
* @brief 大整数加上普通整数.
* @param[inout] x 大整数
* @param[in] y 32 位整数
* @return 大整数
*/
int* bigint_add(int x[], const int y) {
    int i;

    x[0] += y;

```

```

    for (i = 0; i < MAX_LEN; i++) {
        if (x[i] >= BIGINT_RADIX) {
            x[i+1] += x[i] / BIGINT_RADIX;
            x[i] %= BIGINT_RADIX;
        }
    }
    return x;
}

/**
 * @brief 两个大整数相加, in place.
 * @param[inout] x x=x+y
 * @param[in] y y
 * @return 无
 */
void bigint_add1(int x[], const int y[]) {
    int i;
    int c = 0;

    for (i = 0; i < MAX_LEN; i++) { /* 逐位相加 */
        const int tmp = x[i] + y[i] + c;
        x[i] = tmp % BIGINT_RADIX;
        c = tmp / BIGINT_RADIX;
    }
}

/**
 * @brief 大整数乘法, x = x*y.
 * @param[inout] x x
 * @param[in] y y
 * @return 无
 */
void bigint_mul(int x[], const int y) {
    int i;
    int c = 0; /* 保存进位 */

    for (i = 0; i < MAX_LEN; i++) { /* 用 y, 去乘以 x 的各位 */
        const int tmp = x[i] * y + c;
        x[i] = tmp % BIGINT_RADIX;
        c = tmp / BIGINT_RADIX;
    }
}

#define MAX 80 /* n,m 的最大值 */

int n, m;
int matrix[MAX][MAX]; /* 输入矩阵 */
/* f[i][j] 表示某一行区间 [i,j] 的最大值, 转移方程
 * f[i][j] = f[i+1][j]+a[i]*2^x, f[i][j-1]+a[j]*2^x, 1<=x<=m
 * 等价于 f[i][j] = 2*max(f[i+1][j]+a[i], f[i][j-1]+a[j])
 */

```

```

int f[MAX][MAX][MAX_LEN];
int tmp[MAX_LEN];

/**
 * @brief 计算单行.
 * @param[in] a 某一行
 * @return 此行能获得的最大得分
 */
int* dp(int a[MAX], int l, int r) {
    if (l == r) {
        int_to_bigint(a[l] * 2, f[l][r]);
        return f[l][r];
    }
    if (f[l][r][0] >= 0) return f[l][r];

    if (l < r) {
        memcpy(f[l][r], dp(a, l+1, r), sizeof(f[l][r]));
        bigint_mul(bigint_add(f[l][r], a[l]), 2);
        memcpy(tmp, dp(a, l, r-1), sizeof(tmp));
        bigint_mul(bigint_add(tmp, a[r]), 2);

        if (bigint_cmp(f[l][r], tmp) < 0) memcpy(f[l][r], tmp, sizeof(tmp));
    }
    return f[l][r];
}

int main() {
    int i, j, result[MAX_LEN];
    scanf("%d%d", &n, &m);

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

    memset(result, 0, sizeof(result));
    for (i = 0; i < n; i++) {
        memset(f, 0, sizeof(f));
        for (j = 0; j < m; j++) {
            int k;
            for (k = 0; k < m; k++) {
                f[j][k][0] = -1;
            }
        }
        bigint_add1(result, dp(matrix[i], 0, m-1));
    }
    bigint_print(result, MAX_LEN);
    return 0;
}

```

相关的题目

与本题相同的题目：

- WIKIOI 1166 矩阵取数游戏, <http://www.wikioi.com/problem/1166/>

与本题相似的题目：

- None

10.8 棋盘型动态规划

10.8.1 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 10-2所示。

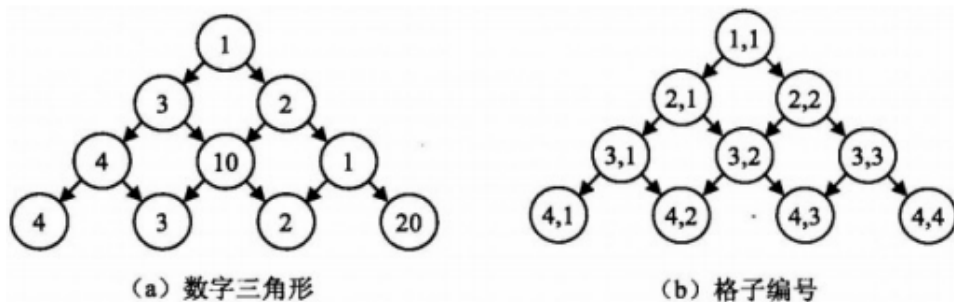


图 10-2 数字三角形问题

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，记忆化搜索。

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

#define MAXN 100

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

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

/**
 * @brief 求从位置 (i,j) 出发时能得到的最大和
 *
 * numbers_triangle1.c
```



```

* @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。

```

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

#define MAXN 100

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

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

/**
* @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));

```

numbers_triangle2.c

```
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/>
- wikioi 1220 数字三角形, <http://www.wikioi.com/problem/1220/>

与本题相似的题目：

- None

10.8.2 过河卒

描述

如图，A 点有一个过河卒，需要走到目标 B 点。卒行走规则：可以向下、或者向右。同时在棋盘上的任一点有一个对方的马（如上图的 C 点），该马所在的点和所有跳跃一步可达的点称为对方马的控制点。例如上图 C 点上的马可以控制 9 个点（图中的 P1, P2 … P8 和 C）。卒不能通过对方马的控制点。

^①刘汝佳, 算法竞赛入门经典, 清华大学出版社, 2009

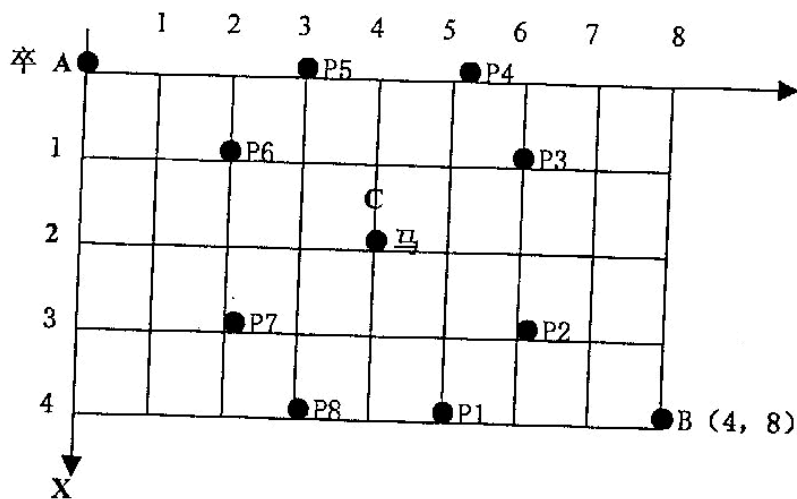


图 10-3 过河卒

棋盘用坐标表示, A 点 (0, 0)、B 点 (n,m) (n,m 为不超过 20 的整数, 并由键盘输入), 同样马的位置坐标是需要给出的 (约定: C 不等于 A, 同时 C 不等于 B)。现在要求你计算出卒从 A 点能够到达 B 点的路径的条数。

输入

B 点的坐标 (n,m) 以及对方马的坐标 (X,Y)

输出

一个整数 (路径的条数)

样例输入

6 6 3 2

样例输出

17

分析

这是一个棋盘形动态规划。

设状态为 $f[i][j]$, 表示从 (0,0) 到 (i,j) 的路径的条数, 状态转移方程为

$$f[i][j] = f[i-1][j] + f[i][j-1]$$

代码

```

#include <stdio.h>
#include <memory.h>

#define MAX 21

int n, m;
int x, y; /* 马儿的坐标 */

/* 八个控制点, 顺时针方向 */
int dx[] = {-2, -1, 1, 2, 2, 1, -1, -2};
int dy[] = {1, 2, 2, 1, -1, -2, -2, -1};

int g[MAX][MAX]; /* 棋盘, 马儿以及马儿的控制点为 1, 其余为 0 */
int f[MAX][MAX]; /* f[i][j] = f[i-1][j] + f[i][j-1] */

void dp() {
    int i, j;
    memset(g, 0, sizeof(g));
    memset(f, 0, sizeof(f));

    g[x][y] = 1;
    for (i = 0; i < 8; i++) if(x+dx[i] >= 0 && x+dx[i] <= n &&
        y+dy[i] >= 0 && y+dy[i] <= m){
        g[x+dx[i]][y+dy[i]] = 1;
    }

    f[0][0] = 1; /* 起点 */
    /* 初始化边界 */
    for (i = 1; i <= n; i++) if (!g[i][0]) f[i][0] = f[i-1][0];
    for (j = 1; j <= m; j++) if (!g[0][j]) f[0][j] = f[0][j-1];

    for (i = 1; i <= n; i++) {
        for (j = 1; j <= m; j++) {
            if (!g[i][j]) f[i][j] = f[i-1][j] + f[i][j-1];
        }
    }
}

int main() {
    scanf("%d%d%d", &n, &m, &x, &y);
    dp();
    printf("%d\n", f[n][m]);
    return 0;
}

```

相关的题目

与本题相同的题目：

- wikioi 1010, 过河卒, <http://www.wikioi.com/problem/1010/>

与本题相似的题目：

- null

10.8.3 传纸条

描述

小渊和小轩是好朋友也是同班同学，他们在一起总有谈不完的话题。一次素质拓展活动中，班上同学安排做成一个 m 行 n 列的矩阵，而小渊和小轩被安排在矩阵对角线的两端，因此，他们就无法直接交谈了。幸运的是，他们可以通过传纸条来进行交流。纸条要经由许多同学传到对方手里，小渊坐在矩阵的左上角，坐标 $(1, 1)$ ，小轩坐在矩阵的右下角，坐标 (m, n) 。从小渊传到小轩的纸条只可以向下或者向右传递，从小轩传给小渊的纸条只可以向上或者向左传递。

在活动进行中，小渊希望给小轩传递一张纸条，同时希望小轩给他回复。班里每个同学都可以帮他们传递，但只会帮他们一次，也就是说如果此人在小渊递给小轩纸条的时候帮忙，那么在小轩递给小渊的时候就不会再帮忙。反之亦然。

还有一件事情需要注意，全班每个同学愿意帮忙的好感度有高有低（注意：小渊和小轩的好心程度没有定义，输入时用 0 表示），可以用一个 $0 \sim 100$ 的自然数来表示，数越大表示越好心。小渊和小轩希望尽可能找好心程度高的同学来帮忙传纸条，即找到来回两条传递路径，使得这两条路径上同学的好心程度只和最大。现在，请你帮助小渊和小轩找到这样的两条路径。

输入

输入的第一行有 2 个用空格隔开的整数 m 和 n ，表示班里有 m 行 n 列 ($1 \leq m, n \leq 50$)。接下来的 m 行是一个 $m * n$ 的矩阵，矩阵中第 i 行 j 列的整数表示坐在第 i 行 j 列的学生的好心程度。每行的 n 个整数之间用空格隔开

输出

输出共一行，包含一个整数，表示来回两条路上参与传递纸条的学生的好心程度之和的最大值。

样例输入

```
3 3
0 3 9
2 8 5
5 7 0
```

样例输出

```
34
```

分析

这是一个棋盘形动态规划。用矩阵 `int g[MAX][MAX]` 存储输入数据，即每个学生的好心值。

题目等价于为从 $(1, 1)$ 传两张纸条到 (m, n) 。每个纸条都是由上面或左边传递过来的，所以有四种情况。设状态为 $f[i][j][k][l]$ ，表示纸条一在 (i, j) ，纸条二在 (k, l) 的好心程度之和，状态转移方程为

$$f[i][j][k][l] = \max \{ \begin{aligned} &f[i-1][j][k-1][l], \\ &f[i-1][j][k][l-1], \\ &f[i][j-1][k][l-1], \\ &f[i][j-1][k-1][l] \end{aligned} \} + g[i][j] + g[k][l]$$

代码

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

#define MAX 51 /* m,n 的最大值是 50，应该有笔误，实际是 51 */

int m, n;          /* 行，列 */
int g[MAX][MAX];   /* 好心值 */
/** 题目等价于为从 (1, 1) 传两张纸条到 (m,n),
 * f[i][j][k][l] 表示纸条一在 (i,j)，纸条二在 (k,l) 的好心程度之和
 */
int f[MAX][MAX][MAX][MAX];

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

void dp() {
    int i, j, k, l;
    memset(f, 0, sizeof(f));
```

deliver_note.c

```

/* 每个纸条都是由上面或左边传递过来的， 所以有四种情况 */
for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
        for (k = 0; k < m; k++) {
            for (l = 0; l < n; l++) {
                if(i > 0 && k > 0 && (i != k || j != l))
                    f[i][j][k][l] = max(f[i][j][k][l],
                                           f[i-1][j][k-1][l] + g[i][j]+g[k][l]);
                if(i > 0 && l < n && (i-1 != k || j != l-1))
                    f[i][j][k][l] = max(f[i][j][k][l],
                                           f[i-1][j][k][l-1] + g[i][j]+g[k][l]);
                if(j < n && l < n && (i != k || j != l))
                    f[i][j][k][l] = max(f[i][j][k][l],
                                           f[i][j-1][k][l-1] + g[i][j]+g[k][l]);
                if(j < n && k > 0 && (i != k-1 || j-1 != l))
                    f[i][j][k][l] = max(f[i][j][k][l],
                                           f[i][j-1][k-1][l] + g[i][j]+g[k][l]);
            }
        }
    }
}

int main() {
    int i, j;
    scanf("%d%d", &m, &n);
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            scanf("%d", &g[i][j]);
        }
    }
    dp();
    printf("%d\n", f[m-1][n-1][m-1][n-1]);
    return 0;
}

```

deliver_note.c

TODO: 本题可以优化为三维

相关的题目

与本题相同的题目：

- wikioi 1169, 传纸条, <http://www.wikioi.com/problem/1169/>

与本题相似的题目：

- null

10.8.4 骑士游历

描述

设有一个 $n * m$ 的棋盘 ($2 \leq n \leq 50, 2 \leq m \leq 50$), 如图 10-4所示, 在棋盘上有一个中国象棋马。规定:

- 马只能走日字
- 马只能向右跳

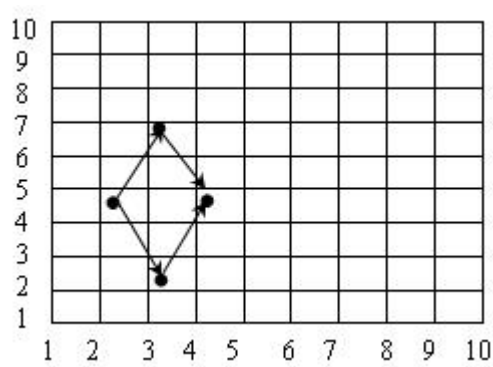


图 10-4 骑士游历

问给定起点 $(x1,y1)$ 和终点 $(x2,y2)$, 求出马从 $(x1,y1)$ 出发到 $(x2,y2)$ 的合法路径条数。

输入

第一行 2 个整数 n 和 m , 第二行 4 个整数 $x1,y1,x2,y2$

输出

合法路径条数

样例输入

30 30
1 15 3 15

样例输出

2

分析

这是一道棋盘型动态规划。

设状态为 $f[i][j]$ ，表示从起点 $(x1,y1)$ 到 (i,j) 的合法路径条数，状态转移方程为

$$f[i][j] = f[i-1][j-2] + f[i-1][j+2] + f[i-2][j-1] + f[i-2][j+1]$$

注意，合法路径条数有可能非常大，32 位整数存不下，需要用 64 位整数。

代码

```
#include <stdio.h>
#include <memory.h>
#include <stdint.h>

#define MAX 51 /* n,m 最大值，本题范围又有笔误，应该是 51 */

int n, m;
int x1, y1, x2, y2;
/*f[i][j] 表示从 (x1,y1) 到 (i,j) 的合法路径的条数 */
int64_t f[MAX+1][MAX+1];

void dp() {
    int i, j;
    memset(f, 0, sizeof(f));
    f[x1][y1] = 1;
    for (i = x1+1; i <= x2; i++) {
        for (j = 1; j <= m; j++) {
            f[i][j] = f[i-1][j-2] + f[i-1][j+2] + f[i-2][j-1] + f[i-2][j+1];
        }
    }
}

int main() {
    scanf("%d%d", &n, &m);
    scanf("%d%d%d%d", &x1, &y1, &x2, &y2);
    dp();
    printf("%lld\n", f[x2][y2]);
    return 0;
}
```

horse.c

horse.c

相关的题目

与本题相同的题目：

- wikioi 1219, 骑士游历, <http://www.wikioi.com/problem/1219/>

与本题相似的题目：

- null

10.9 划分型动态规划

10.9.1 乘积最大

描述

设有一个长度为 N 的数字串，要求选手使用 K 个乘号将它分成 $K + 1$ 个部分，找出一种分法，使得这 $K + 1$ 个部分的乘积能够为最大。

举个例子：有一个数字串：312，当 $N = 3$ ， $K = 1$ 时会有以下两种分法：

- $3*12=36$
- $31*2=62$

这时，符合题目要求的结果是： $31*2=62$ 。

输入

输入共有两行：

第一行共有 2 个自然数 $N, K (6 \leq N \leq 40, 1 \leq K \leq 6)$

第二行是一个长度为 N 的数字串。

输出

输出最大乘积

样例输入

```
4 2
1231
```

样例输出

```
62
```

分析

首先，本题可以用区间型动态规划的思路解决。设状态为 $f[i][j][k]$ ，表示 i 到 j 这一段用 k 个乘号的最大乘积，状态转移方程如下：

$$f[i][j][k] = \max \{f[i][s][t] * f[s+1][j][k-t-1]\}, i \leq s < j, 0 \leq t < k$$

复杂度是 $O(N^3 K^2)$ ，代码如下：

```

/** wikioi 1017 乘积最大, http://www.wikioi.com/problem/1017 */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#define MAXN 40
#define MAXK 6

int N, K;
char str[MAXN];

/** f[i][j][k] 表示 i 到 j 这一段用 k 个乘号的最大乘积. */
int64_t f[MAXN][MAXN][MAXK+1];

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

/** 计算 str[l,r] 字符串的值, int64 可以过, 不用高精度. */
int64_t num(int l, int r) {
    int64_t ret = 0;
    int i;
    for (i = l; i <= r; i++) {
        ret = ret * 10 + str[i] - '0';
    }
    return ret;
}

/**
 * 区间型动态规划, 复杂度是  $O(n^3 k^2)$ .
 *
 *  $f[i][j][k] = \max(f[i][s][t] * f[s+1][j][k-t-1])$ ,
 *  $i \leq s < j$ ,  $0 \leq t < k$ 
 */
void dp() {
    int i, j, k, s, t;
    memset(f, 0, sizeof(f));
    for (i = 0; i < N; i++) {
        for (j = i; j < N; j++) {
            f[i][j][0] = num(i, j);
        }
    }

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

```

```

        for (j = i; j < N; j++) {
            for (k = 1; k <= K; k++) {
                for (s = i; s < j; s++) {
                    for (t = 0; t < k; t++) {
                        f[i][j][k] = max(f[i][j][k],
                                           f[i][s][t] * f[s + 1][j][k - t - 1]);
                    }
                }
            }
        }
    }

int main() {
    scanf("%d%d", &N, &K);
    scanf("%s", str);
    dp();
    printf("%lld\n", f[0][N-1][K]);
    return 0;
}

```

尽管上面的代码可以 AC 了，但还有更高效思路。设状态为 $f[k][j]$ ，表示在区间 $[0, j]$ 放入 k 个乘号的最大乘积，状态转移方程如下：

$$f[k][j] = \max \{f[k][j], f[k-1][i] * num(i+1, j), 0 \leq i < j\}$$

复杂度是 $O(N^2 K)$ 。代码如下。

代码

```

/** wikioi 1017 乘积最大, http://www.wikioi.com/problem/1017 */
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

#define MAXN 40
#define MAXK 6

int N, K;
char str[MAXN];

/** f[k][j] 在区间 [0, j] 放入 k 个乘号的最大乘积. */
int64_t f[MAXK+1][MAXN];

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

/** 计算 str[l,r] 字符串的值, int64 可以过, 不用高精度. */
int64_t num(int l, int r) {
    int64_t ret = 0;

```

maximal_product.c

```
int i;
for (i = 1; i <= r; i++) {
    ret = ret * 10 + str[i] - '0';
}
return ret;
}

/** 划分型动态规划，复杂度是  $O(n^3 \cdot k^2)$ 。 */
void dp() {
    int i, j, k;
    memset(f, 0, sizeof(f));
    for (j = 0; j < N; j++) {
        f[0][j] = num(0, j);
    }

    for (k = 1; k <= K; k++) {
        for (j = 0; j < N; j++) {
            for (i = 0; i < j; i++) {
                f[k][j] = max(f[k][j], f[k-1][i] * num(i+1, j));
            }
        }
    }
}

int main() {
    scanf("%d%d", &N, &K);
    scanf("%s", str);
    dp();
    printf("%lld\n", f[K][N-1]);
    return 0;
}
```

maximal_product.c

相关的题目

与本题相同的题目：

- wikioi 1017, 乘积最大, <http://www.wikioi.com/problem/1017/>

与本题相似的题目：

- null

10.9.2 数的划分

描述

将整数 n 分成 k 份，且每份不能为空，任意两种划分方案不能相同 (不考虑顺序)。

例如： $n = 7$ ， $k = 3$ ，下面三种划分方案被认为是相同的。

```
1 1 5
1 5 1
5 1 1
```

问有多少种不同的分法。

输入

$n, k (6 < n \leq 200, 2 \leq k \leq 6)$

输出

一个整数，即不同的分法。

样例输入

```
7 3
```

样例输出

```
4
```

分析

设状态为 $f[k][i]$ ，表示将整数 i 分成 k 份的方案数。有两种情况，一种划分中有至少一个 1，另外一种划分中所有的数都大于 1。对于第一种划分，去掉一个 1，就变成了一个子问题 $f[n-1][k-1]$ ，对于第二种划分，把划分中的每个数都减去 1，就变成了一个子问题 $f[k][n-k]$ 。因此，状态转移方程如下：

$$f[k][i] = f[k-1][i-1] + f[k][i-k]$$

代码

```
integer_partition.c

/** wikioi 1039 数的划分, http://www.wikioi.com/problem/1039 */
#include <stdio.h>
#include <memory.h>

#define MAXN 300
#define MAXK 10

int N, K;

/** f[k][i] 表示将整数 i 分成 k 份的方案数. */
int f[MAXK+1][MAXN+1];
```

```
/** 划分型动态规划, 复杂度是  $O(K*N)$ . */
void dp() {
    int i, k;
    memset(f, 0, sizeof(f));
    for (i = 1; i <= N; i++) {
        f[1][i] = 1;
        if (i <= K) f[i][i] = 1;
    }

    for (k = 2; k <= K; k++) {
        for (i = k+1; i <= N; i++) {
            f[k][i] = f[k-1][i-1] + f[k][i-k];
        }
    }
}

int main() {
    scanf("%d%d", &N, &K);
    dp();
    printf("%d\n", f[K][N]);
    return 0;
}
```

integer_partition.c

相关的题目

与本题相同的题目：

- wikioi 1039, 数的划分, <http://www.wikioi.com/problem/1039/>

与本题相似的题目：

- None

10.10 树型动态规划

10.10.1 访问艺术馆

描述

皮尔是一个出了名的盗画者，他经过数月的精心准备，打算到艺术馆盗画。艺术馆的结构，每条走廊要么分叉为二条走廊，要么通向一个展览室。皮尔知道每个展室里藏画的数量，并且他精确地测量了通过每条走廊的时间，由于经验老道，他拿下一副画需要 5 秒的时间。你的任务是设计一个程序，计算在警察赶来之前（警察到达时皮尔回到了入口也算），他最多能偷到多少幅画。

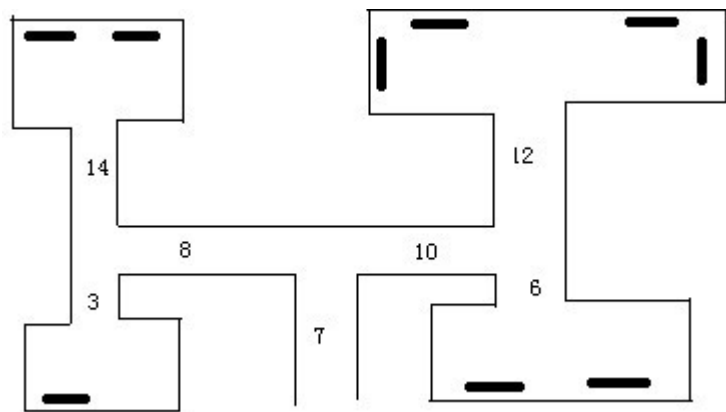


图 10-5 艺术馆

输入

第 1 行是警察赶到得时间 $s(s \leq 600)$ ，以秒为单位。第 2 行描述了艺术馆得结构，是一串非负整数，成对地出现：每一对得第一个数是走过一条走廊得时间，第 2 个数是它末端得藏画数量；如果第 2 个数是 0，那么说明这条走廊分叉为两条另外得走廊。走廊的数目 ≤ 100 。数据按照深度优先得次序给出，请看样例

输出

输出偷到得画得数量

样例输入

```
60
7 0 8 0 3 1 14 2 10 0 12 4 6 2
```

样例输出

```
2
```

分析

设状态为 $f[root][time]$ ，表示在 $root$ 节点，剩下 $time$ 时间内能偷到的画数，则状态转移方程如下：

$$f[root][time] = \max \{ f[left][t] + f[right][ct - t], ct = time - T[root], 0 \leq t \leq ct \}$$

代码

art_gallery.c

```

/* wikioi 1163 访问艺术馆, http://www.wikioi.com/problem/1163/ */
#include<stdio.h>
#include <memory.h>

#define max(a,b) ((a)>(b)?(a):(b))
#define MAXN 1010 /* 走廊的最大数目 */

/* 输入数据, 警察到达时间, 走廊时间, 藏画数量 */
int LIMIT, T[MAXN], P[MAXN];
/* 二叉树, 节点个数, 左孩子, 右孩子 */
int n, L[MAXN], R[MAXN]; /* 0 位置未用 */

/* f[root][time] 表示在 root 节点, 剩下 time 时间内能偷到的画数 */
int f[MAXN][MAXN];

/**
 * @brief 输入的数据是前序遍历序列, 依次读入, 递归建树
 * @return 根节点
 */
int build_tree() {
    n++;
    scanf("%d%d", &T[n], &P[n]);
    T[n] *= 2; /* 走廊花费时间要 *2, 因为还要出来 */
    int now = n; /* 当前节点 */
    if (P[now] == 0) {
        L[now] = build_tree(); /* 建立节点 now 的左子树 */
        R[now] = build_tree(); /* 建立节点 now 的右子树 */
    }
    return now;
}

/**
 * 动规过程, 记忆化搜索.
 * @param[in] root 开始节点
 * @param[in] 剩余时间
 * @return 偷到得画的数量
 */
int dp(int root, int time) {
    int t;
    if (f[root][time] != -1)
        return f[root][time];
    if (!time)
        return f[root][time] = 0;

    const int ct = time - T[root]; /* 留给左右子节点的时间 */

    /* 到达叶子节点, 即展览室 */
    if (!L[root] && !R[root]) {
        /* 不能 tt / 5, 因为时间够, 但是可能画不够 */
        if (P[root] * 5 <= ct)

```

```
        return f[root][time] = P[root];
    else
        return f[root][time] = ct / 5;
}

f[root][time] = 0;
for (t = 0; t <= ct; t++) {
    int lp = dp(L[root], t);
    int rp = dp(R[root], ct - t);
    f[root][time] = max(f[root][time], lp + rp);
}
return f[root][time];
}

int main() {
    scanf("%d", &LIMIT);
    build_tree();
    memset(f, -1, sizeof(f));
    printf("%d\n", dp(1, LIMIT));
    return 0;
}
```

art_gallery.c

相关的题目

与本题相同的题目：

- wikioi 1163, 访问艺术馆, <http://www.wikioi.com/problem/1163/>

与本题相似的题目：

- None

10.10.2 没有上司的舞会

描述

Ural 大学有 N 个职员，编号为 $1 \sim N$ 。他们有从属关系，也就是说他们的关系就像一棵以校长为根的树，父结点就是子结点的直接上司。每个职员有一个快乐指数。现在有个周年庆宴会，要求与会职员的快乐指数最大。但是，没有职员愿和直接上司一起与会。

输入

第一行一个整数 N ($1 \leq N \leq 6000$)。

接下来 N 行，第 $i+1$ 行表示 i 号职员的快乐指数 R_i ($-128 \leq R_i \leq 127$)。

接下来 $N-1$ 行，每行输入一对整数 L, K 。表示 K 是 L 的直接上司。

最后一行输入 0,0，表示结束。

输出

输出最大的快乐指数

样例输入

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

样例输出

```
5
```

分析

设状态为 $f[k][0]$ 和 $f[k][1]$, $f[k][0]$ 表示第 i 个人不参加时的最大值, $f[k][1]$ 表示第 k 个人参加的最大值, 则状态转移方程如下:

$$f[k][0] = \sum_l \max \{f[l][0], f[l][1]\}, \text{ 其中 } k \text{ 是 } l \text{ 的直接上司}$$

$$f[k][1] = \sum_l f[l][0] + r[k]$$

代码

```
/* wikioi 1380 没有上司的舞会, http://www.wikioi.com/problem/1380 */
#include<stdio.h>
#include<string.h>

#define MAXN 6001 /* 0 位置未用 */
#define max(a,b) ((a)>(b)?(a):(b))
typedef char bool;

int n;
```

ball.c

```
/* 快乐指数 */
int r[MAXN];

/* 静态链表节点. */
typedef struct edge_t {
    int v;
    int prev; /* 上一条边, 倒着串起来 */
} edge_t;

/* 多个静态链表在一个数组里, 实质上是树的孩子表示法 */
edge_t edge[MAXN-1];

/* head[root] 指向表头, 即最后一条边 */
int head[MAXN];

/* has_boss[i] 表示第 i 个人是否有上司 */
bool has_boss[MAXN];

int f[MAXN][2];

int cnt; /* 边个数-1 */
void add_edge(int u, int v) {
    edge[cnt].v = v;
    edge[cnt].prev = head[u];
    head[u] = cnt++;
}

void dp(int k) {
    int p;
    f[k][1] = r[k];
    f[k][0] = 0;
    for (p = head[k]; p != -1; p = edge[p].prev) {
        int l = edge[p].v;
        dp(l);
        f[k][1] += f[l][0];
        f[k][0] = f[k][0] + max(f[l][1], f[l][0]);
    }
}

int solve() {
    int k;
    memset(f, 0, sizeof(f));
    for (k = 1; k <= n; k++) if (!has_boss[k]) {
        break;
    }

    dp(k);
    return max(f[k][0], f[k][1]);
}

int main() {
    int i;
    scanf("%d", &n);
```

```
for (i = 1; i <= n; i++) scanf("%d", &r[i]);

cnt = 0;
memset(has_boss, 0, sizeof(has_boss));
memset(head, -1, sizeof(head));
int x, y;
for (i = 1; i < n; i++) {
    scanf("%d%d", &x, &y);
    has_boss[x] = 1;
    add_edge(y, x);
}
scanf("%d%d", &x, &y);

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

ball.c

相关的题目

与本题相同的题目：

- wikioi 1380, 没有上司的舞会, <http://www.wikioi.com/problem/1380/>

与本题相似的题目：

- None

10.11 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。

10.11.1 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 $S \times S$ ，所以时间复杂度为 $O(S^2)$ ，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

```

/**
 * 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*) calloc(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], h[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>

10.11.2 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

广度优先搜索

当题目看不出任何规律，既不能用分治，贪心，也不能用动规时，这时候万能方法——搜索，就派上用场了。搜索分为广搜和深搜，广搜里面又有普通广搜，双向广搜， A^* 搜索等。深搜里面又有普通深搜，回溯法等。

广搜和深搜非常类似（除了在扩展节点这部分不一样），二者有相同的框架，如何表示状态？如何扩展状态？如何判重？尤其是判重，解决了这个问题，基本上整个问题就解决了。

11.1 算法框架

思考的步骤：

1. 如何表示状态？即一个状态需要存储哪些些必要的的数据，就可以完整表示该状态？这一步不考虑是求长度还是求路径的问题，纯粹考虑状态本身的数据。
2. 是求最短路径长度，还是最短路径（或动作序列）？如果是求长度，则状态里面要存路径长度；如果是求路径或动作序列，则要用一棵树存储宽搜过程中的路径。
 - (a) 如果是求长度，则状态里面要存路径长度
3. 是求最短路径长度，还是最短路径（或动作序列）？如果是求路径或动作序列
 - (a) 要用一棵树存储宽搜过程中的路径
 - (b) 是否可以预估状态个数的上限？能够预估状态总数，则开一个大数组，用树的双亲表示法；如果不能预估状态总数，则要使用一棵通用的树（自己实现或用标准库）。这一步也是第 4 步的必要不充分条件。
4. 关于判重，状态是否存在完美哈希方案？即将状态一一映射到整数，互相之间不会冲突。

- (a) 如果不存在, 则需要使用通用的哈希表 (自己实现或用标准库, 例如 `std::unordered_set`) 来判重; 自己实现哈希表的话, 如果能够预估状态个数的上限, 则可以开两个数组, `head` 和 `next`, 表示哈希表, 参考第 §11.3 节方案 2。
- (b) 如果存在, 则可以开一个大布尔数组, 作为哈希表来判重, 且此时可以精确计算出状态总数, 而不仅仅是预估上限。

bfs_template.c

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

/***** 输入数据, 用全局变量存放 *****/
...
/*
例如
int m = MAXN, n = MAXN; // 迷宫的行数, 列数
// 迷宫, 0 表示空地, 1 表示障碍物
int map[MAXN][MAXN]; // 迷宫, 0 表示空地, 1 表示障碍物
*/

/***** 一些常量 *****/
...
/* 例如
// 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
*/

#ifdef __cplusplus
typedef char bool;
#define false 0
#define true 1
#endif

/**
 * @strut 状态
 */
typedef struct state_t {
    ... data1; /** 状态的数据, 可以有多个字段. */
    ... data2; /** 状态的数据, 可以有多个字段. */
    int action; /** 由父状态移动到本状态的动作 */
    int index; /** 本状态在 path[] 中的下标, 求路径或动作序列时需要。
                如果存在完美哈希, 则不需要本字段 (此时将
                状态的哈希值作为下标, 而哈希值可以直接计算出) */
    int father; /** 父状态在 path[] 中的下标, 求路径或动作序列时需要 */
    int count; /** 所花费的步骤数 (也即路径长度-1), 求路径长度时需要 */
} state_t;

/***** 如果题目要求输出路径或动作序列, 则需要下面的变量和函数 *****/
```

```

#define STATE_MAX ... /* 状态总数 */
/**
 * 记录动作序列，树的双亲表示法。
 * 如果不能预估状态个数的上限，则不能用数组，要用通用树
 * 一般此时会有完美哈希方案，开一个大数组存储每个状态的动作，下标就是状态的哈希值
 * 其实一般用不了这么大的数组，广搜很很快就会结束
 */
state_t path[STATE_MAX];
int path_index = 0; /** 每出现一个新状态，就增 1，将状态存到该位置 */

/**
 * @brief 打印动作序列。
 * 如果有完美哈希方案，状态的哈希值就是下标。
 * 起点状态没有父亲和动作为-1，因为它没有父状态，也就没有所谓的从父节点到它的动作。
 * @param[in] end 终点状态的下标
 * @return 父状态
 */
void print_action(const int end) {
    if (path[end].father == -1) return;

    print_action(path[end].father);
    putchar(name[path[end].action]);
}

/**
 * @brief 打印坐标序列。
 * 如果有完美哈希方案，状态的哈希值就是下标。
 * 起点状态没有父亲和动作为-1，因为它没有父状态，也就没有所谓的从父节点到它的动作。
 * @param[in] end 终点状态的哈希值
 * @return 父状态
 */
void print_path(const int end) {
    if (path[end].father == -1) {
        printf("(%d, %d)\n", end / n, end % n);
        return;
    }
    print_path(path[end].father);
    printf("(%d, %d)\n", end / n, end % n);
}

/***** 如果题目要求输出路径或动作序列，则需要上面的变量和函数 *****/

/** 计算状态的哈希值，如果存在完美哈希方案，则定义本函数，否则不需要。
 * 哈希值只依赖 state_t 的数据字段。
 */
int state_hash(const state_t *s);

/** 初始化查找表。 */
void history_init();

/**
 * @brief 状态判重。

```

```

* 一般用哈希表 (set::unordered_set), 如果存在完美哈希, 则用数组
* @param[in] s 状态
* @return 已经访问过, 返回 true, 否则 false
*/
bool history_contains(const state_t *s);

/**
* @brief 标记该状态已经被访问
* @param[in] s 状态
* @return 无
*/
void history_insert(const state_t *s);

/**
* @brief 扩展第一个状态.
* @param[in] s 状态
* @param[out] next 第一个状态
* @return 如果还有下一个状态, 返回 true, 否则返回 false
*/
void state_extend_init(const state_t *s);

/**
* @brief 扩展下一个状态.
* @param[in] s 状态
* @param[out] next 下一个状态
* @return 如果还有下一个状态, 返回 true, 否则返回 false
*/
bool state_extend(const state_t *s, state_t *next);

/**
* @brief 判断状态是否为目标.
* @param[in] s 状态
* @return 如果已经达到目标状态, 返回 true, 否则 false
*/
bool state_is_target(const state_t *s);

typedef state_t queue_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 "queue.c" /* 见“栈和队列->队列”这节, 如果是 C++, 则使用 std::queue */
// 队列
queue_t q;

/*
* @brief 广搜
*
* @param[in] x 入口的 x 坐标
* @param[in] y 入口的 y 坐标
* @return 目标状态在 path[] 中的下标, 如果不需要求路径, 则声明为 void
*/
int bfs(state_t *start) {
    queue_init(&q, 16);
    history_init();

```



```

start->action = -1;    /* 起点状态没有动作 */
start->index = path_index++;
start->father = -1;    /* 起点状态没有父状态 */
start->count = 0;

path[state_hash(start) or start->index] = *start;
history_insert(start); // 千万别忘了标记此处的访问记录
if (state_is_target(start)) /* 如果起点就是终点, 返回 */
    return state_hash(start) or start->index;
queue_push(&q, *start);

while (!queue_empty(&q)) {
    const state_t s = queue_front(&q);
    state_t next;
    queue_pop(&q);

    state_extend_init(&s);
    while (state_extend(&s, &next)) {
        if (state_is_target(&next)) {
            // printf("%d\n", next.count);
            queue_uninit(&q);
            return next.index or state_hash(&next);
        }
        queue_push(&q, next);
        history_insert(&next);
    }
}
queue_uninit(&q);
return -1;
}

int main(void) {
    state_t start;
    int end; /* 目标状态在 path[] 中的下标 */
    input(&start);

    end = bfs(&start);

    print_action(end);
    printf("\n");
    print_path(end);
    return 0;
}

/***** functions implement *****/

/* 哈希表容量, 要大于状态总数, 若存在完美哈希方案, 则等于状态总数 */
#define HASH_CAPACITY ...

/** 哈希表, 标记状态是否已访问过。
 * 如果存在完美哈希方案, 则用数组作为哈希表, 否则用 std::unordered_set
 */
... visited

```

```

// 例如 bool visited[HASH_MAX];

int state_hash(const state_t *s) {
    ...
}

void history_init() {
    /* 如果是数组 */
    memset(visited, 0, sizeof(visited));
    /* 如果是 std::unordered_set */
    visited.clear();
}

bool history_contains(const state_t *s) {
    /* 如果是数组 */
    return visited[state_hash(s)] == true;
    /* 如果是 std::unordered_set */
    return visited.count(s) > 0;
}

void history_insert(const state_t *s) {
    /* 如果是数组 */
    visited[state_hash(s)] = true;
    /* 如果是 std::unordered_set */
    visited.insert(s);
}

/** 扩展节点时，记录当前到了什么哪一步。
 * 是 state_extend_init() 和 state_extend() 的共享变量
 */
int action_cur;
/* 动作的范围 */
#define ACTION_BEGIN ...
#define ACTION_END ...

/* 扩展点的位置，是 state_extend_init() 和 state_extend() 的共享变量 */
int extend_pos;

void state_extend_init(const state_t *s) {
    action_cur = ACTION_BEGIN;
    extend_pos = // 根据 s 进行计算
}

bool state_extend(const state_t *s, state_t *next) {
    // extract values from s
    value1 = ...
    value2 = ...
    while(action_cur < ACTION_END) {
        // apply action_cur to values
        value1 = ...
        value2 = ...
        next->count = s->count + 1;
        if (values are valid) {

```

```

        // assign new values to next's data fields
        next->data1 = value1
        next->data2 = value2

        if (!history_contains(next)) { /* 判重 */
            next->action = action_cur;
            next->index = path_index++;
            next->father = s->index or state_hash(s);
            /* 记录路径 */
            path[next->index or state_hash(next)] = *next;
            action_cur++; /* return 前别忘了增 1 */
            return true;
        }
        action_cur++;
    }
    return false;
}

// 目标状态
const state_t END = {..., 0};

bool state_is_target(const state_t *s) {
    ...
    /* 例如 return memcmp(s->data, END.data, xx * sizeof(yy)) == 0; */
    /* 例如 return s->data == END.data; */
}

```

bfs_template.c

11.2 走迷宫

描述

一个迷宫由一个 01 矩阵表示，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路径。任何时候都不能在障碍物格子中，也不能走到迷宫之外。只能横着走或竖着走，不能斜着走。数据保证有唯一解。

输入

一个 5×5 的二维数组

输出

左上角到右下角的最短路径

样例输入

```
0 1 0 0 0
0 1 0 1 0
0 0 0 0 0
0 1 1 1 0
0 0 0 1 0
```

样例输出

(0, 0) (1, 0) (2, 0) (2, 1) (2, 2) (2, 3) (2, 4) (3, 4) (4, 4)

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 11-1(a) 所示），以及这些路径上每个节点的前驱（如图 11-1(b) 所示）。

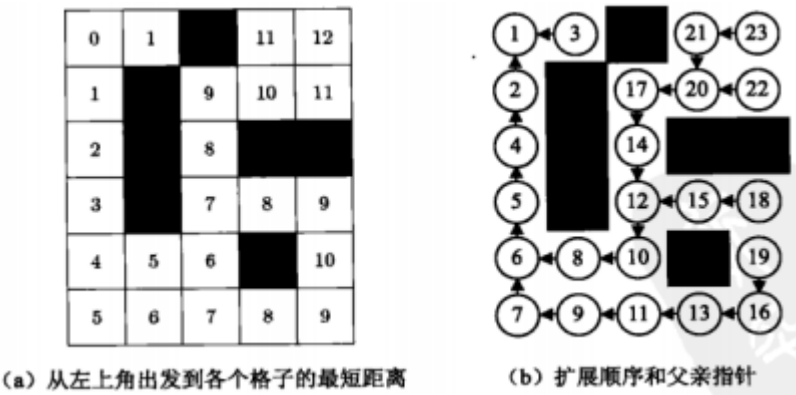


图 11-1 用 BFS 求迷宫中最短路径

代码

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

#define MAXN 5

// 迷宫的行数，列数
int m = MAXN, n = MAXN;
// 迷宫，0 表示空地，1 表示障碍物
int map[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

#ifdef __cplusplus
typedef char bool;
#define false 0
#define true 1
#endif

typedef struct state_t {
    int data;
    int action;
    int father;
} state_t;

#define STATE_MAX MAXN*MAXN /* 状态总数 */

state_t path[STATE_MAX];

void print_action(const int end) {
    if (path[end].father == -1) return;

    print_action(path[end].father);
    putchar(name[path[end].action]);
}

void print_path(const int end) {
    if (path[end].father == -1) {
        printf("(%d, %d)\n", end / n, end % n);
        return;
    }
    print_path(path[end].father);
    printf("(%d, %d)\n", end / n, end % n);
}

int state_hash(const state_t *s);

void history_init();

bool history_contains(const state_t *s);

void history_insert(const state_t *s);

void state_extend_init(const state_t *s);

bool state_extend(const state_t *s, state_t *next);

bool state_is_target(const state_t *s);

typedef state_t queue_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 "queue.c" /* 见“栈和队列->队列”这节，如果是 C++，则使用 std::queue */
queue_t q;

int bfs(state_t *start) {
    queue_init(&q, 16);
    history_init();

    start->action = -1;
    start->father = -1;

    path[state_hash(start)] = *start;
    history_insert(start);
    if (state_is_target(start))
        return state_hash(start);
    queue_push(&q, *start);

    while (!queue_empty(&q)) {
        const state_t s = queue_front(&q);
        queue_pop(&q);
        state_t next;
        state_extend_init(&s);
        while (state_extend(&s, &next)) {
            if (state_is_target(&next)) {
                queue_uninit(&q);
                return state_hash(&next);
            }
            queue_push(&q, next);
            history_insert(&next);
        }
    }
    queue_uninit(&q);
    return -1;
}

int main(void) {
    int i, j;
    state_t start = {0, -1, -1}; /* 左上角为起点 */
    int end;

    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++) {
            scanf("%d", &map[i][j]);
        }
    }

    end = bfs(&start);
    print_path(end);
    return 0;
}

/***** functions implement *****/
/* 哈希表容量，要大于状态总数，若存在完美哈希方案，则等于状态总数 */
#define HASH_CAPACITY STATE_MAX

```

```

bool visited[HASH_CAPACITY];

int state_hash(const state_t *s) {
    return s->data;
}

void history_init() {
    memset(visited, 0, sizeof(visited));
}

bool history_contains(const state_t *s) {
    return visited[state_hash(s)] == true;
}

void history_insert(const state_t *s) {
    visited[state_hash(s)] = true;
}

int action_cur;
#define ACTION_BEGIN 0
#define ACTION_END 4
/** 扩展点, 即当前位置 */
int x, y;

void state_extend_init(const state_t *s) {
    action_cur = ACTION_BEGIN;
    x = s->data / n;
    y = s->data % n;
}

bool state_extend(const state_t *s, state_t *next) {
    while(action_cur < ACTION_END) {
        const int nx = x + dx[action_cur];
        const int ny = y + dy[action_cur];

        if (nx >= 0 && nx < m && ny >= 0 && ny < n && !map[nx][ny]) {
            next->data = nx * n + ny;

            if (!history_contains(next)) { /* 判重 */
                /* 记录路径 */
                next->action = action_cur;
                next->father = state_hash(s);
                path[state_hash(next)] = *next;

                action_cur++; /* return 前别忘了增 1 */
                return true;
            }
        }
        action_cur++;
    }
    return false;
}

```

```
const state_t END = {24, -1, -1};
bool state_is_target(const state_t *s) {
    return s->data == END.data;
}
```

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>

11.3 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 11-2 所示。

2	6	4
1	3	7
	5	8

8	1	5
7	3	6
4		2

图 11-2 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。

^①刘汝佳, 算法竞赛入门经典, 清华大学出版社, 2009

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"。

样例输入

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

样例输出

```
ullddrurddlurdruldr
```

分析

计算“最短”，很自然的想到 BFS。

如何表示一个状态？一个 3*3 的棋盘，首先可以想到用一个数组 `int data[9]` 来表示。把每个格子看做是整数的一位的话，可以用一个整数来表示，最大的棋盘 87654321x，表示成整数就是 876543210，没有超过 32 位整数的范围。

如何判重？用哈希表，自己实现或者用 STL。C++ STL 里有 `std::set`，C++ 11 新加入了 `std::unordered_set`。建议先用 STL 写一个版本，确保主算法正确，然后把 `std::set` (或 `std::unordered_set`) 替换成自己写的哈希表。

由于本题的特殊性，存在一种完美哈希 (perfect hashing) 方案。即康托展开。棋盘本身是一个排列，将排列转化为序数，用序数作为 hash 值。例如，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。

此题更优的解法还有双向 BFS (见 §11.5)，A* 算法 (见 §11.6)。

代码

方案 1，完美哈希，使用康托展开。

```
/* POJ 1077 Eight, http://poj.org/problem?id=1077 */
#include <stdio.h>
#include <string.h>
```

eight_digits_bfs.c

```

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

/***** 一些常量 *****/
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 name[] = { 'u', 'd', 'l', 'r' };

#ifndef __cplusplus
typedef char bool;
#define false 0
#define true 1
#endif

typedef char int8_t;

/**
 * @strut 状态
 */
typedef struct state_t {
    int8_t data[DIGITS]; /** 状态的数据. */
    int action; /** 由父状态移动到本状态的动作 */
    int father; /** 父状态在 path[] 中的下标，也即父状态的哈希值 */
    int count; /** 所花费的步骤数（也即路径长度-1） */
} state_t;

// 3x3 的棋盘，状态最多有 9! 种
#define STATE_MAX 362880 /** 状态总数 */

state_t path[STATE_MAX+1];

/**
 * @brief 打印动作序列.
 * @param[in] end 终点状态的哈希值
 * @return 父状态
 */
void print_action(const int end) {
    if (path[end].father == -1) return;

    print_action(path[end].father);
    putchar(name[path[end].action]);
}

int state_hash(const state_t *s);

void history_init();

bool history_contains(const state_t *s);

void history_insert(const state_t *s);

```

```

void state_extend_init(const state_t *s);

bool state_extend(const state_t *s, state_t *next);

bool state_is_target(const state_t *s);

typedef state_t queue_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 "queue.c" /* 见“栈和队列->队列”这节, 如果是 C++, 则使用 std::queue */
// 队列
queue_t q;

int bfs(state_t *start) {
    queue_init(&q, 16);
    history_init();

    start->action = -1;
    start->father = -1;
    start->count = 0;

    path[state_hash(start)] = *start;
    history_insert(start);
    if (state_is_target(start))
        return state_hash(start);
    queue_push(&q, *start);

    while (!queue_empty(&q)) {
        const state_t s = queue_front(&q);
        state_t next;
        queue_pop(&q);

        state_extend_init(&s);
        while (state_extend(&s, &next)) {
            if (state_is_target(&next)) {
                // printf("%d\n", next.count);
                queue_uninit(&q);
                return state_hash(&next);
            }
            queue_push(&q, next);
            history_insert(&next);
        }
    }
    queue_uninit(&q);
    return -1;
}

/**
 * @brief 输入.
 * @return 无
 */
void input(state_t *start) {
    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') start->data[i] = 0; // x 映射成数字 0
        else
            start->data[i] = ch - '0';
    }
}

/** for wikioi 1225 */
void input1(state_t *start) {
    int i, n;
    scanf("%d", &n);

    /* 将整数转化为棋盘 */
    for(i = DIGITS-1; i >= 0; i--) {
        start->data[i] = n % 10;
        n /= 10;
    }
}

int main(void) {
    state_t start;
    int end; /* 目标状态在 path[] 中的下标 */
    input(&start);

    end = bfs(&start);

    print_action(end);
    printf("\n");
    return 0;
}

/***** functions implement *****/

/***** 方案 1, 完美哈希, 使用康托展开 *****/

// 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};
/* 哈希表容量, 要大于状态总数, 若存在完美哈希方案, 则等于状态总数 */
#define HASH_CAPACITY STATE_MAX

bool visited[HASH_CAPACITY];

int state_hash(const state_t *s) {
    int i, j;
    int key = 0;
    for (i = 0; i < DIGITS; i++) {
        int cnt = 0; /* 逆序数 */

```

```

        for (j = i + 1; j < DIGITS; j++) if (s->data[i] > s->data[j]) cnt++;
        key += fac[i] * cnt;
    }
    return key;
}

void history_init() {
    memset(visited, 0, sizeof(visited));
}

bool history_contains(const state_t *s) {
    return visited[state_hash(s)] == true;
}

void history_insert(const state_t *s) {
    visited[state_hash(s)] = true;
}

int action_cur;
#define ACTION_BEGIN 0
#define ACTION_END 4

/* 扩展点, 即 0 的位置 */
int z;

void state_extend_init(const state_t *s) {
    action_cur = ACTION_BEGIN;
    for (z = 0; z < DIGITS; z++) {
        if (s->data[z] == SPACE_NUMBER) {
            break; // 找 0 的位置
        }
    }
}

bool state_extend(const state_t *s, state_t *next) {
    const int x = z / MATRIX_EDGE; // 行
    const int y = z % MATRIX_EDGE; // 列

    while (action_cur < ACTION_END) {
        const int newx = x + dx[action_cur];
        const int newy = y + dy[action_cur];
        const int newz = newx * MATRIX_EDGE + newy;

        if (newx >= 0 && newx < MATRIX_EDGE && newy >= 0 &&
            newy < MATRIX_EDGE) { // 没有越界
            *next = *s;
            next->data[newz] = SPACE_NUMBER;
            next->data[z] = s->data[newz];
            next->count = s->count + 1;
            if (!history_contains(next)) { /* 判重 */
                next->action = action_cur;
                next->father = state_hash(s);
                /* 记录路径 */
            }
        }
        action_cur++;
    }
}

```

```

        path[state_hash(next)] = *next;
        action_cur++; /* return 前别忘了增 1 */
        return true;
    }
}
    action_cur++;
}
return false;
}
}

// 目标状态
const state_t END = {{1, 2, 3, 4, 5, 6, 7, 8, 0}, -1, -1};
// for wikioi 1225
const state_t END1 = {{1, 2, 3, 8, 0, 4, 7, 6, 5}, -1, -1};

bool state_is_target(const state_t *s) {
    return memcmp(s->data, END.data, DIGITS * sizeof(int8_t)) == 0;
}

```

eight_digits_bfs.c

方案 2，不知道是否存在完美哈希方案，但能够预估状态个数的上限，用树双亲表示法存储路径，用自己实现的哈希表判重。

```

/* POJ 1077 Eight, http://poj.org/problem?id=1077 */
#include <stdio.h>
#include <string.h>
#include <assert.h>

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

/***** 一些常量 *****/
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 name[] = { 'u', 'd', 'l', 'r' };

#ifndef __cplusplus
typedef char bool;
#define false 0
#define true 1
#endif

typedef char int8_t;

/**
 * @strut 状态
 */
typedef struct state_t {
    int8_t data[DIGITS]; /** 状态的数据. */
    int action; /* 由父状态移动到本状态的动作 */
}

```

eight_digits_bfs2.c

```

    int index; /** 本状态在 path[] 中的下标 */
    int father; /** 父状态在 path[] 中的下标 */
    int count; /** 所花费的步骤数 (也即路径长度-1) */
} state_t;

// 3x3 的棋盘, 状态最多有 9! 种
#define STATE_MAX 362880 /** 状态总数 */

state_t path[STATE_MAX+1];
int path_index = 0;

/**
 * @brief 打印动作序列.
 * @param[in] end 终点状态的哈希值
 * @return 父状态
 */
void print_action(const int end) {
    if (path[end].father == -1) return;

    print_action(path[end].father);
    putchar(name[path[end].action]);
}

void history_init();

bool history_contains(const state_t *s);

void history_insert(const state_t *s);

void state_extend_init(const state_t *s);

bool state_extend(const state_t *s, state_t *next);

bool state_is_target(const state_t *s);

typedef state_t queue_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 "queue.c" /** 见“栈和队列->队列”这节, 如果是 C++, 则使用 std::queue */
// 队列
queue_t q;

int bfs(state_t *start) {
    queue_init(&q, 16);
    history_init();

    start->action = -1;
    start->index = path_index++;
    start->father = -1;
    start->count = 0;

    path[start->index] = *start;
    history_insert(start);
    if (state_is_target(start))

```

```

        return state_hash(start);
    queue_push(&q, *start);

    while (!queue_empty(&q)) {
        const state_t s = queue_front(&q);
        state_t next;
        queue_pop(&q);

        state_extend_init(&s);
        while (state_extend(&s, &next)) {
            if (state_is_target(&next)) {
                // printf("%d\n", next.count);
                queue_uninit(&q);
                return next.index;
            }
            queue_push(&q, next);
            history_insert(&next);
        }
    }
    queue_uninit(&q);
    return -1;
}

/**
 * @brief 输入.
 * @return 无
 */
void input(state_t *start) {
    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') start->data[i] = 0; // x 映射成数字 0
        else
            start->data[i] = ch - '0';
    }
}

/** for wikioi 1225 */
void input1(state_t *start) {
    int i, n;
    scanf("%d", &n);

    /* 将整数转化为棋盘 */
    for(i = DIGITS-1; i >= 0; i--) {
        start->data[i] = n % 10;
        n /= 10;
    }
}

int main(void) {
    state_t start;

```



```

    int end;
    input(&start);

    end = bfs(&start);

    print_action(end);
    printf("\n");
    return 0;
}

/***** functions implement *****/

/***** 方案 2 不知道完美哈希方案, 自己实现哈希表 *****/
#define HASH_CAPACITY 1000000 /* 哈希表容量, 要大于状态总数 */

int head[HASH_CAPACITY];
int next[STATE_MAX];

int state_hash(const state_t *s) {
    int i;
    int ret = 0;
    for(i = 0; i < DIGITS; i++) ret = ret * 10 + s->data[i];
    return ret % HASH_CAPACITY;
}

void history_init() {
    memset(head, 0, sizeof(head));
    memset(next, 0, sizeof(next));
}

bool history_contains(const state_t *s) {
    const int h = state_hash(s);
    int u = head[h]; // 从表头开始查找单链表
    while(u) {
        // 找到了
        if(memcmp(path[u].data, s->data,
                   DIGITS * sizeof(int8_t)) == 0) return true;
        u = next[u]; // 顺着链表继续找
    }
    return false;
}

void history_insert(const state_t *s) {
    const int h = state_hash(s);
    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(path[u].data, path[s->index].data,
                   sizeof(DIGITS * sizeof(int8_t))) == 0) return;
        u = next[u]; // 顺着链表继续找
    }
    assert(head[h] >= 0);
    next[s->index] = head[h]; /* 插入到首节点前面, 头查法 */
}

```

```

    head[h] = s->index;
    return;
}

int action_cur;
#define ACTION_BEGIN 0
#define ACTION_END 4

/* 扩展点, 即 0 的位置 */
int z;

void state_extend_init(const state_t *s) {
    action_cur = ACTION_BEGIN;
    for (z = 0; z < DIGITS; z++) {
        if (s->data[z] == SPACE_NUMBER) {
            break; // 找 0 的位置
        }
    }
}

bool state_extend(const state_t *s, state_t *next) {
    const int x = z / MATRIX_EDGE; // 行
    const int y = z % MATRIX_EDGE; // 列

    while (action_cur < ACTION_END) {
        const int newx = x + dx[action_cur];
        const int newy = y + dy[action_cur];
        const int newz = newx * MATRIX_EDGE + newy;

        next->count = s->count + 1;
        if (newx >= 0 && newx < MATRIX_EDGE && newy >= 0 &&
            newy < MATRIX_EDGE) { // 没有越界
            *next = *s;
            next->data[newz] = SPACE_NUMBER;
            next->data[z] = s->data[newz];

            if (!history_contains(next)) { /* 判重 */
                next->action = action_cur;
                next->index = path_index++;
                next->father = s->index;
                /* 记录路径 */
                path[next->index] = *next;
                action_cur++; /* return 前别忘了增 1 */
                return true;
            }
        }
        action_cur++;
    }
    return false;
}

// 目标状态
const state_t END = {{1, 2, 3, 4, 5, 6, 7, 8, 0}, -1, -1};

```

```
// for wikioi 1225
const state_t END1 = {{1, 2, 3, 8, 0, 4, 7, 6, 5}, -1, -1};

bool state_is_target(const state_t *s) {
    return memcmp(s->data, END.data, DIGITS * sizeof(int8_t)) == 0;
}
```

eight_digits_bfs2.c

方案 3，不知道完美哈希方案，但能够预估状态个数的上限制，用树双亲表示法存储路径，用标准库的哈希表判重。

```
//前面的代码与方案 2 一摸一样
//...

/***** 方案 3 不知道完美哈希方案，使用标准库的哈希表 *****/

// 重载 state_t 的 == 操作符
typedef struct state_t {
    int8_t data[DIGITS]; /** 状态的数据. */
    int action; /** 由父状态移动到本状态的动作 */
    int index; /** 本状态在 path[] 中的下标 */
    int father; /** 父状态在 path[] 中的下标 */
    int count; /** 所花费的步骤数（也即路径长度-1） */

    bool operator==(const state_t& other) const {
        return memcmp(data, other.data, DIGITS * sizeof(int8_t)) == 0;
    }
} state_t;

#include <unordered_set>

// 定制一个哈希函数
namespace std {
template<> struct hash<state_t> {
    size_t operator()(const state_t & x) const {
        int i;
        int ret = 0;
        for (i = 0; i < DIGITS; i++)
            ret = ret * 10 + x.data[i];
        return ret;
    }
};
}

std::unordered_set<state_t> visited;

void history_init() {
    visited.clear();
}

bool history_contains(const state_t *s) {
    return visited.count(*s) > 0;
}
```

eight_digits_bfs3.cpp

```
}  
  
void history_insert(const state_t *s) {  
    visited.insert(*s);  
}
```

```
//...  
//后面的代码也与方案 2 一摸一样
```

eight_digits_bfs3.cpp

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>
- wikioi 1225 八数码难题, <http://www.wikioi.com/problem/1225/>

与本题相似的题目：

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

11.4 四子连棋

描述

在一个 4*4 的棋盘上摆放了 14 颗棋子，其中有 7 颗白色棋子，7 颗黑色棋子，有两个空白地带，任何一颗黑白棋子都可以向上下左右四个方向移动到相邻的空格，这叫行棋一步，黑白双方交替走棋，任意一方可以先走，如果某个时刻使得任意一种颜色的棋子形成四个一线（包括斜线），这样的状态为目标棋局。

输入

一个 4*4 的初始棋局，黑棋子用 B 表示，白棋子用 W 表示，空格地带用 O 表示。

输出

移动到目标棋局的最少步数。

^①刘汝佳, 算法竞赛入门经典, 清华大学出版社, 2009

样例输入

```
BWBO
WBWB
BWBW
WBWO
```

样例输出

```
5
```

分析

求最少步数，很自然的想到广搜。

如何表示一个状态？用一个二维数组 `int board[4][4]` 表示，还需要记录该状态是由白子还是黑子移动而导致的，走到该状态已经花费的步数。

如何扩展节点？每一步，从队列弹出一个状态，两个空格都可以向四个方向扩展，把得到的状态入队列。

如何判重？棋盘用二维矩阵存储，用 0 表示空格，1 表示黑色，2 表示白色，所以最后可以看成是一个 16 位的三进制数。用这个数作为棋盘的编码，就可以用来判重了。注意，本题要黑白交替走，所以我们要区分状态是由白子还是黑子移动而导致的。

可以用 C++ 的 `std::map` 来判重，

```
/* history[0] 记录白子的历史，history[1] 记录黑子的历史. */
std::map<int, bool> history[2];
```

也可以开一个大数组当做哈希表，

```
#define HASH_MOD 43036875 /* hash 表大小 */
/* history[0] 记录白子的历史，history[1] 记录黑子的历史. */
bool history[2][HASH_MOD];
```

代码

```
/** wikioi 1004 四子连棋 , http://www.wikioi.com/problem/1004 */
#include <stdio.h>
#include <string.h>

#define LEN 4 /* 边长 */

/* 右, 左, 上, 下 (左下角为坐标原点) */
const int dx[] = { 1, -1, 0, 0 };
const int dy[] = { 0, 0, 1, -1 };

#ifdef __cplusplus
typedef char bool;
```

four_adjacent.c

```

#define false 0
#define true 1
#endif

/**
 * @strut 状态
 */
typedef struct state_t {
    // 状态的数据
    int board[LEN][LEN]; /* 棋局, 1 表示黑子, 2 表示白子, 0 表示空白 */
    int color; /* 本状态是由白子还是黑子移动而导致的 */
    int count; /* 所花费的步骤数 (也即路径长度-1), 求路径长度时需要 */
} state_t;

/**
 * @brief 计算状态的哈希值。
 * 棋盘用二维矩阵存储, 用 0 表示空格, 1 表示黑色, 2 表示白色, 所以最后可以看成
 * 一个 16 位的三进制数。最大为
 * 2222
 * 2221
 * 1111
 * 1100
 * 值为 43036875。
 * @return 棋盘所表示的三进制数转化为十进制数
 */
//TODO: 共 C16 7×C9 2 个状态, 用类似康托的方法储存
int state_hash(const state_t *s);

void history_init();

bool history_contains(const state_t *s);

void history_insert(const state_t *s);

void state_extend_init(const state_t *s);

bool state_extend(const state_t *s, state_t *next);

bool state_is_target(const state_t *s);

typedef state_t queue_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 "queue.c" /* 见“栈和队列->队列”这节, 如果是 C++, 则使用 std::queue */
// 队列
queue_t q;

void bfs(state_t *start) {
    queue_init(&q, 16);
    history_init();

    start->count = 0;
    start->color = 1;

```

```

    history_insert(start);
    queue_push(&q, *start);

    start->color = 2;

    history_insert(start); // 千万别忘了标记此处的访问记录
    if (state_is_target(start)) /* 如果起点就是终点, 返回 */
        return;
    queue_push(&q, *start);

    while (!queue_empty(&q)) {
        const state_t s = queue_front(&q);
        state_t next;
        queue_pop(&q);

        state_extend_init(&s);
        while (state_extend(&s, &next)) {
            if (state_is_target(&next)) {
                printf("%d\n", next.count);
                queue_uninit(&q);
                return;
            }
            queue_push(&q, next);
            history_insert(&next);
        }
    }
    queue_uninit(&q);
}

int main() {
    int i, j;
    char s[LEN + 1];
    state_t start;

    for (i = 0; i < LEN; i++) {
        scanf("%s", s);
        for (j = 0; j < LEN; j++) {
            if (s[j] == 'B') start.board[i][j] = 1;
            else if (s[j] == 'W') start.board[i][j] = 2;
            else start.board[i][j] = 0;
        }
    }

    bfs(&start);
    queue_uninit(&q);
    return 0;
}

/***** functions implement *****/

/* 哈希表容量, 要大于状态总数, 若存在完美哈希方案, 则等于状态总数 */
#define HASH_CAPACITY 43036875

```

```

/** 哈希表, 标记状态是否已访问过。
 * visited[0] 记录白子的历史, visited[1] 记录黑子的历史。
 */
bool visited[2][HASH_CAPACITY];

#define RADIX 3 /* 三进制 */

int state_hash(const state_t *s) {
    int i, j;
    int ret = 0;

    for (i = 0; i < LEN; i++) {
        for (j = 0; j < LEN; j++) {
            ret = ret * RADIX + s->board[i][j];
        }
    }
    return ret;
}

void history_init() {
    memset(visited[0], 0, sizeof(sizeof(bool) * HASH_CAPACITY));
    memset(visited[1], 0, sizeof(sizeof(bool) * HASH_CAPACITY));
}

bool history_contains(const state_t *s) {
    return visited[s->color - 1][state_hash(s)] == true;
}

void history_insert(const state_t *s) {
    visited[s->color - 1][state_hash(s)] = true;
}

/* 扩展的时候, 先定空格, 再定方向 */
/* 记录当前方向, 例如 action_cur[0] 记录了第一个空格, 当前在扩展哪个方向
 */
int action_cur[2];
#define ACTION_BEGIN 0
#define ACTION_END 4

typedef struct point_t {
    int x, y;
} point_t;

/* 记录当前在扩展哪一个空格, 值为 0 或 1 */
int space_cur;
/* 两个空格的位置 */
point_t extend_pos[2];

void state_extend_init(const state_t *s) {
    int i, j, k;
    action_cur[0] = ACTION_BEGIN;
    action_cur[1] = ACTION_BEGIN;
    space_cur = 0;
}

```



```

k = 0;
// 寻找两个空白的格子的位置
for (i = 0; i < LEN; i++) {
    for (j = 0; j < LEN; j++) {
        if (s->board[i][j] == 0) {
            extend_pos[k].x = i;
            extend_pos[k].y = j;
            k++;
        }
    }
}

bool state_extend(const state_t *s, state_t *next) {
    int i;

    for (i = 0; i < 2; i++) { /* 先第一个空格, 再第二个空格 */
        while (action_cur[i] < ACTION_END) {
            const int x = extend_pos[i].x;
            const int y = extend_pos[i].y;
            int nextx = x + dx[action_cur[i]];
            int nexty = y + dy[action_cur[i]];
            *next = *s;
            next->count = s->count + 1;
            next->color = 3 - s->color;

            if (nextx >= 0 && nextx < LEN && nexty >= 0 && nexty < LEN
                /* 必须黑白交替走 */
                && next->color == s->board[nextx][nexty]) {
                /* swap */
                {
                    int temp = next->board[x][y];
                    next->board[x][y] = next->board[nextx][nexty];
                    next->board[nextx][nexty] = temp;
                }

                if (!history_contains(next)) { /* 判重 */
                    action_cur[i]++; /* return 前别忘了增 1 */
                    return true;
                }
            }
            action_cur[i]++;
        }
    }
    return false;
}

bool state_is_target(const state_t *s) {
    int i, j;
    for (i = 0; i < LEN; i++) { /* 逐行检查 */
        int flag = 1; /* 某一行全是同一颜色 */
        for (j = 1; j < LEN; j++)

```

```
        if (s->board[i][j - 1] != s->board[i][j])
            flag = 0;
    if (flag)
        return 1;
}
for (j = 0; j < LEN; j++) { //逐列检查
    int flag = 1; /* 某一行全是同一颜色 */
    for (i = 1; i < LEN; i++)
        if (s->board[i][j] != s->board[i - 1][j]) flag = 0;
    if (flag) return 1;
}
/* 斜线 */
if (s->board[0][0] == s->board[1][1] && s->board[1][1] == s->board[2][2]
    && s->board[2][2] == s->board[3][3])
    return 1;
if (s->board[0][3] == s->board[1][2] && s->board[1][2] == s->board[2][1]
    && s->board[2][1] == s->board[3][0])
    return 1;
return 0;
}
```

four_adjacent.c

相关的题目

与本题相同的题目：

- wikioi 1004 四子连棋, <http://www.wikioi.com/problem/1004/>

与本题相似的题目：

- None

11.5 双向 BFS

11.5.1 八数码问题

题目见 §11.3。

代码

eight_digits_bibfs.c

eight_digits_bibfs.c

11.6 A* 算法

A* 算法 = 宽搜 + 优先队列

将广搜模板（见第 §11.1 节）中的队列改为优先队列，设计好当前状态距离目标状态的预估距离 $h()$ ，就变成了 A* 算法！

11.6.1 八数码问题

题目见 §11.3。

代码

eight_digits_astar.c

```

/** POJ 1077 Eight, http://poj.org/problem?id=1077
  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
  3. 使用优先队列，即堆，加速挑选最优值。
  4. 函数 g= 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>

#define DIGITS 9 // 棋盘上数字的个数，也是变进制数需要的位数
#define MATRIX_EDGE 3 // 棋盘边长
#define RADIX 10

/***** 一些常量 *****/
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 name[] = { 'u', 'd', 'l', 'r' };

// 目标状态
const int GOAL = 123456780;
// 每个数字在棋盘中的位置，例如 0，在 (2,2)=8 这个位置上
int GOAL_POS[DIGITS];

#ifdef __cplusplus
typedef char bool;
#define false 0
#define true 1
#endif

/**
 * @strut 状态
 */
typedef struct state_t {
    int board; /** 状态的数据，即棋局。*/

```

```

    int action; /* 由父状态移动到本状态的动作 */
    int father; /* 父状态在 path[] 中的下标, 也即父状态的哈希值 */
    int count; /** 所花费的步骤数 (也即路径长度-1), 作为 g */
    int h; /** 距离目标状态的估算距离 */
} state_t;

// 3x3 的棋盘, 状态最多有 9! 种
#define STATE_MAX 362880 /* 状态总数 */

state_t path[STATE_MAX+1];

/**
 * @brief 打印动作序列.
 * @param[in] end 终点状态的哈希值
 * @return 父状态
 */
void print_action(const int end) {
    if (path[end].father == -1) return;

    print_action(path[end].father);
    putchar(name[path[end].action]);
}

int state_hash(const state_t *s);

void history_init();

bool history_contains(const state_t *s);

void history_insert(const state_t *s);

void state_extend_init(const state_t *s);

bool state_extend(const state_t *s, state_t *next);

bool state_is_target(const state_t *s);

typedef state_t heap_elem_t; // 元素的类型
/** 状态的比较函数 */
int cmp_state(const state_t *x, const state_t *y) {
    return (x->count + x->h) - (y->count + y->h);
}

/* 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" /* 见“树->堆”这节 */
// 优先队列
heap_t q;

/**
 * 距离目标状态的估算距离 h。
 * @param s 状态
 * @return h
 */

```

```

static int state_get_h(const state_t *state) {
    int i;
    int h = 0;
    int s = state->board;

    for (i = DIGITS - 1; i >= 0; --i) {
        const int p = s % RADIX;
        s /= RADIX;
        /* 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;
}

/* 计算 GOAL_POS */
static void calc_goal_pos() {
    int cur = GOAL;
    int i;
    for (i = DIGITS-1; i >= 0 ; i--) {
        int digit = cur % RADIX;
        GOAL_POS[digit] = i;
        cur /= RADIX;
    }
}

int bfs(state_t *start) {
    calc_goal_pos();
    heap_init(&q, 16, cmp_state);
    history_init();

    start->action = -1;
    start->father = -1;
    start->count = 0;
    start->h = state_get_h(start);

    path[state_hash(start)] = *start;
    history_insert(start);
    if (state_is_target(start))
        return state_hash(start);
    heap_push(&q, *start);

    while (!heap_empty(&q)) {
        const state_t s = heap_top(&q);
        state_t next;
        heap_pop(&q);

        state_extend_init(&s);
        while (state_extend(&s, &next)) {
            if (state_is_target(&next)) {
                // printf("%d\n", next.count);
                heap_uninit(&q);
                return state_hash(&next);
            }
        }
    }
}

```

```

        }
        heap_push(&q, next);
        history_insert(&next);
    }
}
heap_uninit(&q);
return -1;
}

static void int_to_board(int n, int board[DIGITS]) {
    int i;
    for (i = DIGITS - 1; i >= 0; i--) {
        board[i] = n % RADIX;
        n /= RADIX;
    }
}

static int board_to_int(const int board[DIGITS]) {
    int i, s = 0;
    for (i = 0; i < DIGITS; i++)
        s = s * RADIX + board[i];
    return s;
}

/**
 * @brief 输入.
 * @return 无
 */
void input(state_t *start) {
    int ch, i;
    int board[DIGITS];
    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') board[i] = 0; // x 映射成数字 0
        else board[i] = ch - '0';
    }
    start->board = board_to_int(board);
}

/** for wikioi 1225 */
void input1(state_t *start) {
    scanf("%d", &start->board);
}

/**
 * 计算一个排列的逆序数, 0 除外.
 */
static 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 % RADIX;
        permutation /= RADIX;
    }

    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;
}

/**
 * 判断是否有解.
 *
 * 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。
 *
 * 若起始状态和目标状态的逆序数奇偶性相同, 则可相互到达, 否则不可相互到达。
 *
 * @param s 目标状态
 * @return 1 表示有解, 0 表示无解
 */
static int solvable(const int s) {
    return (inversion_count(s) + inversion_count(GOAL)) % 2 == 0;
}

int main(void) {
    state_t start;
    int end; /* 目标状态在 path[] 中的下标 */
    input(&start);

    if (!solvable(start.board)) return 0;

    end = bfs(&start);

    print_action(end);
    printf("\n");
    return 0;
}

/***** functions implement *****/

/***** 方案 1, 完美哈希, 使用康托展开 *****/

// 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};
/* 哈希表容量，要大于状态总数，若存在完美哈希方案，则等于状态总数 */
#define HASH_CAPACITY STATE_MAX

bool visited[HASH_CAPACITY];

/**
 * @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 state_hash(const state_t *state) {
    int i, j;
    int board[DIGITS];
    int key = 0;

    int_to_board(state->board, board);

    for (i = 0; i < DIGITS; i++) {
        int c = 0; // 逆序数
        for (j = i + 1; j < DIGITS; j++) {
            if (board[j] < board[i]) {
                c++;
            }
        }
        key += c * fac[i];
    }

    return key;
}

void history_init() {
    memset(visited, 0, sizeof(visited));
}

bool history_contains(const state_t *s) {
    return visited[state_hash(s)] == true;
}

void history_insert(const state_t *s) {
    visited[state_hash(s)] = true;
}

int action_cur;
#define ACTION_BEGIN 0
#define ACTION_END 4

/* 扩展点，即 0 的位置 */
int z;
int board[DIGITS]; /* 棋盘，暂存数据 */

```



```

void state_extend_init(const state_t *s) {
    action_cur = ACTION_BEGIN;

    int_to_board(s->board, board);
    for (z = 0; z < DIGITS; z++) {
        if (board[z] == SPACE_NUMBER) {
            break; // 找 0 的位置
        }
    }
}

bool state_extend(const state_t *s, state_t *next) {
    const int x = z / MATRIX_EDGE; // 行
    const int y = z % MATRIX_EDGE; // 列

    while (action_cur < ACTION_END) {
        const int newx = x + dx[action_cur];
        const int newy = y + dy[action_cur];
        const int newz = newx * MATRIX_EDGE + newy;

        if (newx >= 0 && newx < MATRIX_EDGE && newy >= 0 &&
            newy < MATRIX_EDGE) { // 没有越界
            board[z] = board[newz];
            board[newz] = SPACE_NUMBER;
            *next = *s;
            next->board = board_to_int(board);
            board[newz] = board[z]; /* 恢复 s 的棋盘 */
            board[z] = SPACE_NUMBER;

            next->count = s->count + 1;
            next->h = state_get_h(next);
            if (!history_contains(next)) { /* 判重 */
                next->action = action_cur;
                next->father = state_hash(s);
                /* 记录路径 */
                path[state_hash(next)] = *next;
                action_cur++; /* return 前别忘了增 1 */
                return true;
            }
        }
        action_cur++;
    }
    return false;
}

// 目标状态
const state_t END = {123456780, -1, -1};
// for wikioi 1225
const state_t END1 = {123804765, -1, -1};

bool state_is_target(const state_t *s) {
    return s->board == END.board;
}

```

}

eight_digits_astar.c

第 12 章

深度优先搜索

12.1 深搜与回溯法的区别

深搜 (Depth-first search, DFS) 的定义见 http://en.wikipedia.org/wiki/Depth_first_search, 回溯法 (backtracking) 的定义见 <http://en.wikipedia.org/wiki/Backtracking>

回溯法 = 深搜 + 剪枝。一般大家用深搜时, 或多或少会剪枝, 因此深搜与回溯法没有什么不同, 可以在它们之间画上一个等号。本书同时使用深搜和回溯法两个术语, 但读者可以认为二者等价。

深搜一般用递归 (recursion) 来实现, 这样比较简洁。

深搜能够在候选答案生成到一半时, 就进行判断, 抛弃不满足要求的答案, 所以深搜比暴力搜索法要快。

12.2 深搜与递归的区别

深搜经常用递归 (recursion) 来实现, 二者常常同时出现, 导致很多人误以为他俩是一个东西。

深搜, 是逻辑意义上的算法, 递归, 是一种物理意义上的实现。深搜, 可以用递归来实现, 也可以用栈来实现; 而递归, 一般总是用来实现深搜。可以说, **递归一定是深搜, 深搜不一定用递归**。

递归有两种加速策略, 一种是**剪枝 (prunning)**, 对中间结果进行判断, 提前返回; 一种是**加缓存** (即 memoization 优化技术), 缓存中间结果, 防止重复计算, 用空间换时间, 和动规中自顶向下的记忆化搜索类似。

剪枝的手段五花八门, 要具体问题具体分析, 要充分观察, 充分利用各种信息来剪枝, 在中间节点提前返回。

加缓存, 可以用数组或 HashMap。维度简单的, 用数组; 维度复杂的, 用 HashMap, C++ 有 `std::map`, C++ 11 以后有 `std::unordered_map`, 比 `std::map` 快。

其实, 递归 + 缓存, 就是一种 memorization。所谓 **memorization** (翻译为记忆化搜

索, 见第 §10.1 节), 就是“top-down with cache” (自顶向下 + 缓存), 它是 Donald Michie 在 1968 年创造的术语, 表示一种优化技术, 在 top-down 形式的程序中, 使用缓存来避免重复计算, 从而达到加速的目的。

memorization 不一定用递归, 就像深搜不一定用递归一样, 可以在迭代 (iterative) 中使用 memorization。**递归也不一定用 memorization**, 可以用 memorization 来加速, 但不是必须的。只有当递归使用了缓存, 它才是 memorization。

12.3 四色问题

描述

给定 $N(N \leq 8)$ 个点的地图, 以及地图上各点的相邻关系, 请输出用 4 种颜色将地图涂色的所有方案数 (要求相邻两点不能涂成相同的颜色)。

数据中 0 代表不相邻, 1 代表相邻。

输入

第一行一个整数 N , 代表地图上有 N 个点。

接下来 N 行, 每行 N 个整数, 每个整数是 0 或者 1。第 i 行第 j 列的值代表了第 i 个点和第 j 个点之间是相邻的还是不相邻, 相邻就是 1, 不相邻就是 0。我们保证 $a[i][j] = a[j][i]$ 。

输出

染色的方案数

样例输入

```
8
0 0 0 1 0 0 1 0
0 0 0 0 0 1 0 1
0 0 0 0 0 0 1 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
```

样例输出

```
15552
```

分析

这是一道经典的题目。深搜。

代码

```
/* wikioi 1116 四色问题    , http://www.wikioi.com/problem/1116/ */
#include <stdio.h>
#include <string.h>

#define MAXN 8

int N;
int g[MAXN][MAXN];

/* 记录每个点的颜色, 四种颜色用 1234 表示, 0 表示未染色. */
int history[MAXN];
int count; /* 方案个数 */

/**
 * 深搜, 给第 i 个节点涂色.
 * @param i 第 i 个地点
 * @return 无
 */
void dfs(int i) {
    int j, c;
    if (i == N) {
        count++;
        return;
    }

    for (c = 1; c < 5; c++) {
        int ok = 1;
        for (j = 0; j < i; j++) {
            if (g[i][j] && c == history[j])
                ok = 0; /* 相邻且同色 */
        }
        if (ok) {
            history[i] = c;
            dfs(i + 1);
        }
    }
}

int main() {
    int i, j;

    scanf("%d", &N);
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            scanf("%d", &g[i][j]);
        }
    }
}
```

four_colors.c

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

four_colors.c

相关的题目

与本题相同的题目：

- wikioi 1116 四色问题, <http://www.wikioi.com/problem/1116/>

与本题相似的题目：

- None

12.4 全排列

描述

给出一个正整数 n , 请输出 n 的所有全排列

输入

一个整数 $n(1 \leq n \leq 10)$

输出

一共 $n!$ 行, 每行 n 个用空格隔开的数, 表示 n 的一个全排列。并且按全排列的字典序输出。

样例输入

3

样例输出

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

分析

这也是一道短小精悍的经典题目。深搜。从代码上看，与上一题的思路几乎一模一样。

代码

```
/* wikioi 1294 全排列 , http://www.wikioi.com/problem/1294/ */
#include <stdio.h>
#include <string.h>

#define MAXN 10

int N;

int history[MAXN];
int count;

void dfs(int i) {
    int j, k;
    if (i == N) {
        count++;
        for (j = 0; j < N; j++) {
            printf("%d ", history[j]);
        }
        printf("\n");
        return;
    }

    for (k = 1; k <= N; k++) {
        int ok = 1;
        for (j = 0; j < i; j++) {
            if (history[j] == k)
                ok = 0;
        }
        if (ok) {
            history[i] = k;
            dfs(i + 1);
        }
    }
}

int main() {
    scanf("%d", &N);
    dfs(0);
    return 0;
}
```

all_permutations.c

all_permutations.c

相关的题目

与本题相同的题目：

- wikioi 1294 全排列, <http://www.wikioi.com/problem/1294/>

与本题相似的题目：

- None

12.5 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 12-1 所示。

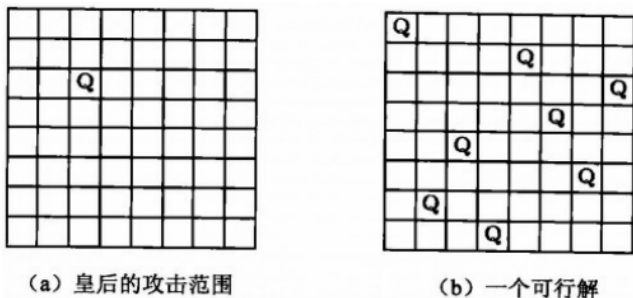


图 12-1 八皇后问题

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， \dots ，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

```
#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/>

^①刘汝佳, 算法竞赛入门经典, 清华大学出版社, 2009

- 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>

12.6 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);
        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_addresses.cpp

12.7 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, \dots, 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

```
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

12.8 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

```
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 13 章

图

在 ACM 竞赛中，图一般使用邻接矩阵表示，代码框架如下；

```
#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

13.1 深度优先搜索

图的深度优先搜索的代码框架如下：

```
/**
 * @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 图的邻接矩阵
 * @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.
```

graph.c


```

* @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 在使用的時候，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

13.1.1 黑白图像

描述

输入一个 $n \times n$ 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 13-1 所示的黑白图像中有 3 个八连块。

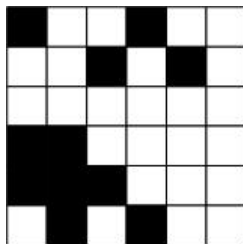


图 13-1 拥有 3 个八连块的黑白图

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
- None

与本题相似的题目：

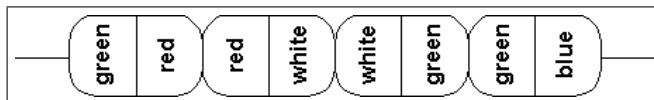
- None

13.1.2 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
- None

与本题相似的题目：

- UVa 10129 Play on Words, <http://t.cn/zTlnBDX>

13.2 广度优先搜索

13.3 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。

13.3.1 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

^①刘汝佳, 算法竞赛入门经典, 清华大学出版社, 2009

法需附设一个数组 `closededge`，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 `closededge[i-1]`，它包括两个域，其中 `lowcost` 存储该边上的权。显然， $\text{closededge}[i].\text{lowcost} = \min\{\text{cost}(u, v_i), u \in U\}$ 。 `adjvex` 域存储该边依附的在 U 中的顶点。

图 13-2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13-1所示。

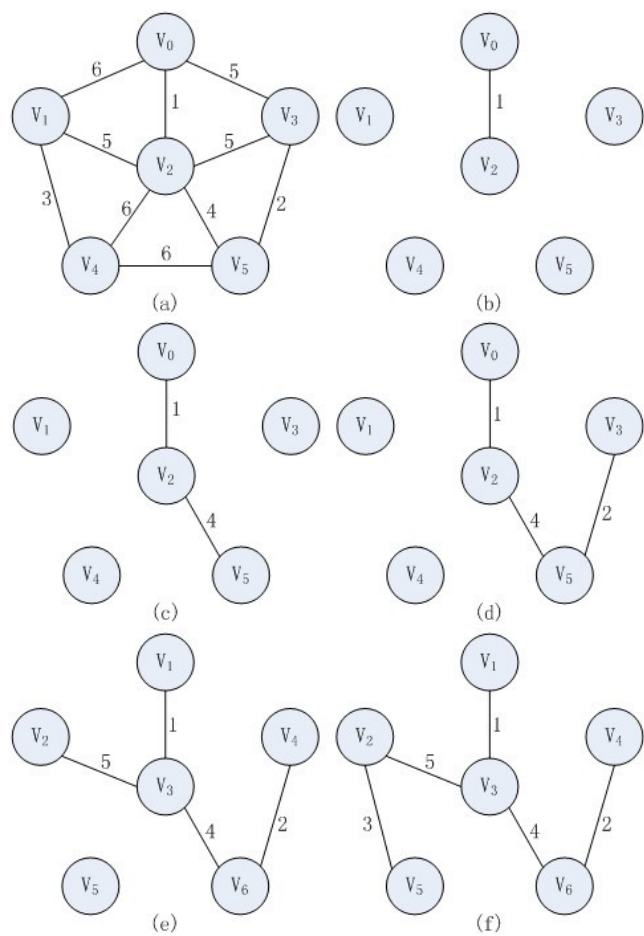


图 13-2 Prim 算法构造最小生成树的过程

表 13-1 构造最小生成树过程中辅助数组的变化

<div>closedge \ i</div>	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;
#define GRAPH_INF INT_MAX

/**
 * @struct
 * @brief 邻接矩阵.
 */
typedef struct mgraph_t {
    int nv; /* 顶点数 */
    int ne; /* 边数 */
    /* 邻接矩阵, 存放边的信息, 如权重等 */
    graph_weight_t matrix[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
} mgraph_t;

mgraph_t g;

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

```

```

} 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 = GRAPH_INF;
    int min_loc = -1;
    for (i = 0; i < n; i++)
        if (closedge[i].lowcost > -GRAPH_INF) {
            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->nv;
    /* 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 = -GRAPH_INF; /* 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 : %d\n", 'A' + closedge[k].adjvex, 'A' + k,
            g->matrix[closedge[k].adjvex][k]);
        sum += g->matrix[closedge[k].adjvex][k];
        // sum += closedge[k].lowcost; // 等价
        closedge[k].lowcost = -GRAPH_INF; /* 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 < closededge[j].lowcost) {
            closededge[j].adjvex = k;
            closededge[j].lowcost = w;
        }
    }
    free(closededge);
    return sum;
}

```

```

/** 读取输入, 构建图. */
void read_graph() {
    int i, j, k, m, n;

    /* 读取节点和边的数目 */
    scanf("%d%d", &m, &n);
    getchar(); // 消耗回车键
    g.nv = m;
    g.ne = n;

    /* 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] = GRAPH_INF;
        }
    }

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

```

```

/* 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() {
    read_graph();
    /* 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率为 n , 第二个循环语句的频率为 $n-1$ 。其中有两个内循环: 其一是在 `closedge[v].lowcost` 中求最小值, 其频率为 $n-1$; 其二是重新选择具有最小代价的边, 其频率为 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

```
#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_INF INT_MAX
```

mgraph_prim2.c

```

/**
 * @struct
 * @brief 邻接矩阵.
 */
typedef struct mgraph_t {
    int nv; /* 顶点数 */
    int ne; /* 边数 */
    /* 邻接矩阵, 存放边的信息, 如权重等 */
    graph_weight_t matrix[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
} mgraph_t;

mgraph_t g;

/**
 * @struct 边
 */
typedef struct edge_t {
    int tail; /* 弧尾, from */
    int head; /* 弧头, to */
    graph_weight_t w; /* 权值 */
} edge_t;

static int edge_cmp(const edge_t *e1, const edge_t *e2) {
    return e1->w - e2->w;
}

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" /* 见 “树->堆” 这节 */

/**
 * @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 e;
    heap_t h;
    const int n = g->nv;
    /* 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->ne, 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 */
    e.tail = u;
    e.head = v;
    /* tail 在树内, head 不在树内 */
    e.w = g->matrix[u][v];
    heap_push(&h, e);
}
while(!heap_empty(&h) && count < n) {
    /* 从堆中退出最小权值边, 存入 ed */
    e = heap_top(&h); heap_pop(&h);
    if(!U[e.head]) {
        /* 输出生成树 TE 的边, 即此边加入 TE */
        printf("%c - %c: %d\n", 'A' + e.tail, 'A' + e.head,
            g->matrix[e.tail][e.head]);
        sum += g->matrix[e.tail][e.head];
        u = e.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;
}

// ...

```

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 $O(e \log_2 e)$ 。

13.3.2 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13-3所示为 Kruskal 算法构造一棵最小生成树的过程。

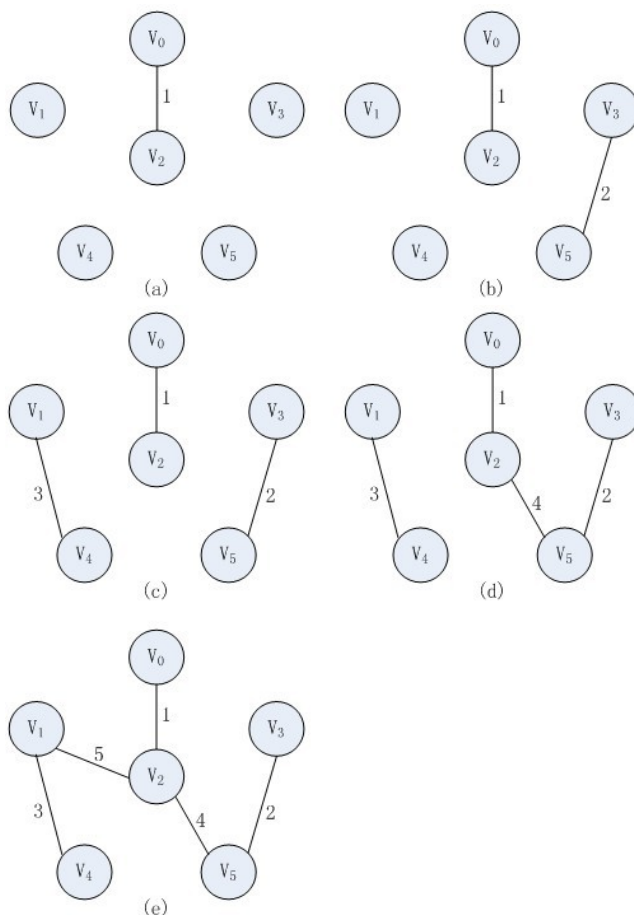


图 13-3 Kruskal 算法构造最小生成树的过程

下面是 Kruskal 算法的 C 语言实现。

代码

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

/* 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 "ufs.c" /* 见“树->并查集”这节 */

#define MAX_VERTICES_NO 11 /* 顶点编号最大值 */
#define MAX_EDGES 100 /* 最大边数 */
/** 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. */
```

kruskal.c

```

typedef int graph_weight_t;

/**
 * @struct 无向图的边.
 */
typedef struct edge_t{
    int u; /** 顶点编号 */
    int v; /** 顶点编号 */
    graph_weight_t w; /** 权值 */
} edge_t;

edge_t edges[MAX_EDGES];
int s[MAX_VERTICES_NO]; /* 并查集, 0 位置未用 */

static int edge_cmp(const edge_t *e1, const edge_t *e2) {
    return e1->w - e2->w;
}

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" /* 见“树->堆”这节 */

/**
 * @brief Kruskal 算法, 求图的最小生成树.
 * @param[in] edges 边的数组
 * @param[in] n 边数, 一定要大于或等于 (顶点数-1)
 * @param[in] m 顶点数
 * @return MST 的边的权值之和
 */
graph_weight_t kruskal(const edge_t edges[], int n, int m) {
    int i;
    graph_weight_t sum = 0;
    heap_t h;
    if (n < m - 1) return -1;

    ufs_init(s, MAX_VERTICES_NO);
    heap_init(&h, n, edge_cmp);

    /* 把所有边插入堆中 */
    for (i = 0; i < n; i++) {
        heap_push(&h, edges[i]);
    }

    for (i = 0; i < n; i++) {
        /* 从堆中退出最小权值边 */
        const edge_t e = heap_top(&h);
        int u, v;
        heap_pop(&h);
        /* 取两顶点所在集合的根 */
        u = ufs_find(s, e.u);
        v = ufs_find(s, e.v);
    }
}

```



```

        if (u != v) { /* 不是同一集合, 说明不连通 */
            ufs_union(s, 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", e.u, e.v);
            sum += e.w;
        }
    }

    heap_uninit(&h);
    return sum;
}

static int edge_cmp1(const void *e1, const void *e2) {
    const edge_t* const ee1 = (const edge_t *)e1;
    const edge_t* const ee2 = (const edge_t *)e2;
    return ee1->w - ee2->w;
}

/** Kruskal 算法, 快排 + 并查集. */
graph_weight_t kruskal1(edge_t edges[], int n, int m) {
    int i;
    graph_weight_t sum = 0;
    if (n < m - 1) return -1;

    ufs_init(s, MAX_VERTICES_NO);
    qsort(edges, n, sizeof(edge_t), edge_cmp1);

    for (i = 0; i < n; i++) {
        /* 从堆中退出最小权值边, 存入 ed */
        const edge_t e = edges[i];
        /* 取两顶点所在集合的根 */
        const int u = ufs_find(s, e.u);
        const int v = ufs_find(s, e.v);
        if (u != v) { /* 不是同一集合, 说明不连通 */
            ufs_union(s, 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", e.u, e.v);
            sum += e.w;
        }
    }

    return sum;
}

/* test
输入数据:
7 11
0 1 7
0 3 5
1 2 8
1 3 9
1 4 7
2 4 5

```

```

3 4 15
3 5 6
4 5 8
4 6 9
5 6 11

```

输出:

```

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

```

Total:39

*/

```

int main() {
    int i, m, n;

    /* 读取顶点数, 边数目 */
    scanf("%d%d", &m, &n);

    /* 读取边信息 */
    for (i = 0; i < n; i++) {
        scanf("%d %d %d", &edges[i].u, &edges[i].v, &edges[i].w);
    }

    /* 求解最小生成树 */
    printf("Total:%d\n", kruskal(edges, n, m));
    return 0;
}

```

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 $O(n + e \log_2 e)$ 。

13.3.3 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`，也就不需要 `double` 了。

代码

```
/* POJ 1751 Highways, http://poj.org/problem?id=1751 */ poj_1751_highways_prim.c

/* 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 "mgraph_prim1.c" /* 见“图->最小生成树->Prim 算法”这节 */

// 1. 修改范围
#define MAX_VERTICES_NUM 750
// 2. 重写 read_graph()
// 3. 重写 main()
// 4. 修改 mgraph_prim() 里的 printf，权值大于 0 才打印出来
if (g->matrix[closededge[k].adjvex][k] > 0)
    printf("%d %d\n", closededge[k].adjvex+1, k+1);

/* 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]);
}

/** 读取输入，构建图. */
void read_graph() {
    int i, j;
    scanf("%d", &n);
    g.nv = n;
    g.ne = n * (n - 1) / 2;

    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;
    }
}

int main() {
    read_graph();
    mgraph_prim(&g);
    return 0;
}
```

poj_1751_highways_prim.c

相关的题目

与本题相同的题目：

- POJ 1751 Highways, <http://poj.org/problem?id=1751>

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>

13.3.4 最优布线问题

描述

学校需要将 n 台计算机连接起来, 不同的 2 台计算机之间的连接费用可能是不同的。为了节省费用, 我们考虑采用间接数据传输结束, 就是一台计算机可以间接地通过其他计算机实现和另外一台计算机连接。

为了使得任意两台计算机之间都是连通的 (不管是直接还是间接的), 需要在若干台计算机之间用网线直接连接, 现在想使得总的连接费用最省, 让你编程计算这个最小的费用。

输入

输入第一行为两个整数 $n, m (2 \leq n \leq 100000, 2 \leq m \leq 100000)$, 表示计算机总数, 和可以互相建立连接的连接个数。接下来 m 行, 每行三个整数 a, b, c 表示在机器 a 和机器 b 之间建立连接的话费是 c 。(题目保证一定存在可行的连通方案, 数据中可能存在权值不一样的重边, 但是保证没有自环)

输出

输出只有一行一个整数, 表示最省的总连接费用。

样例输入

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

样例输出

```
2
```

分析

本题是非常直白的 `kruskal` 算法，可以直接使用第 §13.3.2 节的样例代码。

代码

```

/* wikioi 1231 最优布线问题, http://www.wikioi.com/problem/1231/ */
// 1. 修改范围
#define MAX_VERTICES_NO 100001 /* 顶点编号最大值 */
#define MAX_EDGES 100000 /* 最大边数 */
// 2. 注释掉 kruskal() 里的 printf()
// 3. sum 类型改为 long long, kruskal() 返回值改为 long long
// 4. main() 里的 printf 改为 %lld

/* 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 "kruskal.c" /* 见 “图-> 最小生成树->Kruskal 算法” 这节 */

```

wiring.c

wiring.c

相关的题目

与本题相同的题目：

- wikioi 1231 最优布线问题, <http://www.wikioi.com/problem/1231/>

与本题相似的题目：

- None

13.4 最短路径

13.4.1 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

$$\text{dist}[j] = \min \{ \text{dist}[j], \text{weight}(v, v_j) | v_j \in V - S \}$$

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13-4所示的有向图及其邻接矩阵运行运行 Dijkstra 算法,

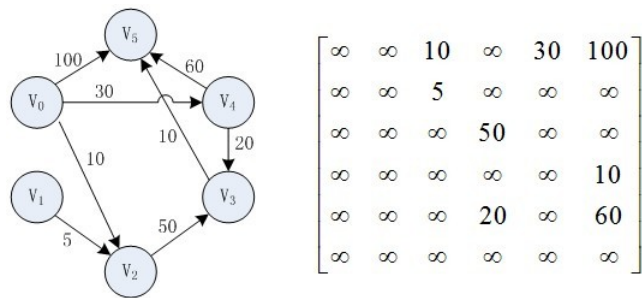


图 13-4 有向图及其邻接矩阵

运算过程中 v_0 到其余个顶点的最短路径, $\text{dist}[]$ 向量的变化情况如表13-2所示 (从一行到下一行只需要更新新加入点的邻接点)。

表 13-2 Dijkstra 算法过程中 $\text{dist}[]$ 向量的变化情况

终点	i=1	i=2	i=3	i=4	i=5
v_1	∞	∞	∞	∞	∞
v_2	10 (v_0, v_2)				
v_3	∞	60 (v_0, v_2, v_3)	50 (v_0, v_4, v_5)		
v_4	30 (v_0, v_4)	30 (v_0, v_4)			
v_5	100 (v_0, v_5)	100 (v_0, v_5)	90 (v_0, v_4, v_5)	60 (v_0, v_4, v_3, 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_INF INT_MAX

/**
 * @struct
 * @brief 邻接矩阵.
 */
typedef struct mgraph_t {
    int nv; /* 顶点数 */
    int ne; /* 边数 */
    /* 邻接矩阵, 存放边的信息, 如权重等 */
    graph_weight_t matrix[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
} 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 */
graph_weight_t 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, graph_weight_t dist[], int path[]) {
    int i, j;
    const int n = g->nv;

    /* 初始化 S 集合 */
    int *S = (int*)calloc(n, sizeof(int));
    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_INF) {
            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 = -1;
        graph_weight_t min = GRAPH_INF;
        for(j = 0; j < n; j++) {
            if(!S[j] && dist[j] < min) {
                u = j;
                min = dist[j];
            }
        }
        if (u < 0) break; /* 结束 */
        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_INF &&
                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", 'A' + v);
    } else {
        print_path_r(path[v], path);
        printf("->%c", 'A' + 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");
    }
}

/**
 * @brief 读取输入, 构建图.
 */
void read_graph() {
    int i, j, k;

    /* 读取节点和边的数目 */
    scanf("%d%d", &g.nv, &g.ne);

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

    /* 读取边信息 */
    getchar(); // 消耗回车键
    for (k = 0; k < g.ne; k++) {
        char chx, chy;
        graph_weight_t w;
        scanf("%c %c %d", &chx, &chy, &w);
    }
}

```

```

        getchar();
        i = chx - 'A';
        j = chy - 'A';
        g.matrix[i][j] = w;
    }
}

```

```
/* 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() {
    read_graph();

    /* 求 v0 到其他所有顶点的最短路径 */
    mgraph_dijkstra(&g, 0, dist, path);
    print_path(path, g.nv);
    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 $O(n^2)$ 。

13.4.2 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 $k(k = 0, 1, \dots, n-1)$ 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, \dots, v_1) 和 (v_1, \dots, v_j) 分别是当前找到的中间顶点的序号不大于 0 的最短路径，那么 $(v_i, \dots, v_1, \dots, v_j)$ 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, \dots, v_k) 和 (v_k, \dots, v_j) 分别是 v_i 到 v_k 和从 v_k 到 v_j 的中间顶点的序号不大于 $k-1$ 的最短路径，则将 $(v_i, \dots, v_k, \dots, v_j)$ 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，

$$D^{(-1)}, D^{(0)}, D^{(1)}, \dots, D^{(k)}, \dots, D^{(n-1)}$$

其中，

$$D^{(-1)}[i][j] = \text{g->matrix}[i][j],$$

$$D^{(k)}[i][j] = \min \left\{ D^{(k-1)}[i][j], D^{(k-1)}[i][k] + D^{(k-1)}[k][j] \right\}, 0 \leq k \leq n-1$$

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13-5所示的有向图及其邻接矩阵运行 Floyd 算法，

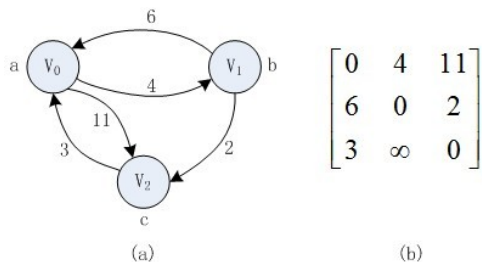


图 13-5 有向图及其邻接矩阵

运算过程中矩阵 D 的变化如表13-3所示。

表 13-3 Floyd 算法过程中方阵和最短路径的变化

D	D ⁽⁰⁾			D ⁽¹⁾			D ⁽²⁾			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	P ⁽⁰⁾			P ⁽¹⁾			P ⁽²⁾			P ⁽³⁾		
	0	1	2	0	1	2	0	1	2	0	1	2
0	A		A	AB		A	AB		AB	AB		AB
	B		C			C			C			C
1	B	B		B	B		B	BC		BC	BC	
	A	C		A	C		A			A		
2	C			C	CA		C	CA		CA	CA	
	A			A	B		A	B		B		

Floyd 算法的 C 语言实现如下。

代码

#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_INF (INT_MAX / 2) /* 确保加法不溢出 */

/**
 * @struct
 * @brief 邻接矩阵。
 */
typedef struct mgraph_t {
 int nv; /* 顶点数 */
 int ne; /* 边数 */
 /* 邻接矩阵，存放边的信息，如权重等 */
 graph_weight_t matrix[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
} mgraph_t;

mgraph_floyd.c

```

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,
                  graph_weight_t dist[][MAX_VERTICES_NUM],
                  int path[][MAX_VERTICES_NUM]) {
    int i, j, k;
    const int n = g->nv;

    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", 'A' + u);
    } else {
        print_path_r(u, path[u][v], path);
        printf("->%c", 'A' + 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->nv; i++) {
        for (j = 0; j < g->nv; j++) {
            if (i != j) {
                print_path_r(i, j, path);
                printf("\n");
            }
        }
        printf("\n");
    }
}

/**
 * @brief 读取输入, 构建图.
 */
void read_graph() {
    int i, j, k;

    /* 读取节点和边的数目 */
    scanf("%d%d", &(g->nv), &(g->ne));

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

    /* 读取边信息 */
    getchar(); // 消耗回车键
    for (k = 0; k < g->ne; k++) {
        char chx, chy;
        graph_weight_t w;

        scanf("%c %c %d", &chx, &chy, &w);
        getchar();
        i = chx - 'A';
        j = chy - 'A';
        g->matrix[i][j] = w;
    }
}

```



```

    }
}

/* 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() {
    read_graph();
    /* 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 $D^{(0)}, D^{(1)}, \dots, D^{(n-1)}$ 。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。

13.4.3 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 §13.4.1 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 最短路, http://acm.hdu.edu.cn/showproblem.php?pid=2544 */
hdu_2544.c

/* 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 "mgraph_dijkstra.c" /* 见“图->最短路径”这节 */

// 1. 修改范围
#define MAX_VERTICES_NUM 100
// 2. 重写 read_graph()
```

```
// 3. 重写 main()

/** 读取输入，构建图. */
void read_graph() {
    int i, j, k;
    /* 读取节点和边的数目 */
    scanf("%d%d", &g.nv, &g.ne);

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

    /* 读取边信息 */
    for (k = 0; k < g.ne; k++) {
        int x, y;
        graph_weight_t w;
        scanf("%d %d %d", &x, &y, &w);
        i = x - 1;
        j = y - 1;
        g.matrix[i][j] = w;
        g.matrix[j][i] = w; // 注意，街道是双向的
    }
}

int main() {
    while (1) {
        read_graph();
        if (g.nv <= 0 || g.ne <= 0) break;
        /* 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.nv - 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>

13.4.4 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

用 Floyd 算法求出每点之间的最短路径，输出距离最大的。

代码

```
poj_1125.c
/* poj 1125 Stockbroker Grapevine, http://poj.org/problem?id=1125 */

/* 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 "mgraph_floyd.c" /* 见“图->最短路径”这节 */

// 1. 修改范围
#define MAX_VERTICES_NUM 100
// 2. 添加 max_len 数组
/** 存放每个点的最短路径的最长者 */
graph_weight_t max_len[MAX_VERTICES_NUM];
// 3. 添加 max_element() 和 min_element
// 4. 重写 read_graph()
// 5. 重写 main()

static int max_element(const graph_weight_t w[], int begin, int end) {
    int i;
    int max_value = -GRAPH_INF;
    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;
}

/** 读取输入，构建图. */
void read_graph() {
    int i, j;
    scanf("%d", &(g.nv));
```

```
g.ne = 0;

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

/* 读取边信息 */
for (i = 0; i < g.nv; i++) {
    int m;
    graph_weight_t w;
    scanf("%d", &m);
    g.ne += m;
    while (m--) {
        scanf("%d %d", &j, &w);
        --j;
        g.matrix[i][j] = w;
    }
}

int main() {
    /* 读取顶点数 */
    while (1) {
        int i, min_pos;
        read_graph();
        if (g.nv == 0)
            break;

        /* 求两两之间的最短路径 */
        mgraph_floyd(&g, dist, path);

        /* 找最短路径的最长者 */
        for (i = 0; i < g.nv; i++) {
            max_len[i] = dist[i][max_element(dist[i], 0, g.nv)];
        }
        /* 找 max_len 的最小者 */
        min_pos = min_element(max_len, 0, g.nv);

        if (max_len[min_pos] == GRAPH_INF) {
            printf("disjoint\n");
        } else {
            printf("%d %d\n", min_pos + 1, max_len[min_pos]);
        }
    }
    return 0;
}
```

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>

13.5 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 $\langle V_i, V_j \rangle$ 是途中的弧，则在序列中顶点 V_i 必须排在顶点 V_j 之前。

如果有向图表示一个工程，顶点表示活动，用有向边 $\langle v_i, v_j \rangle$ 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 Vertex 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

```
#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。 */
mgraph_topo_sort.c
```



```

typedef int graph_weight_t;
#define GRAPH_INF INT_MAX

/**
 * @struct
 * @brief 邻接矩阵.
 */
typedef struct mgraph_t {
    int nv; /* 顶点数 */
    int ne; /* 边数 */
    /* 邻接矩阵, 存放边的信息, 如权重等 */
    graph_weight_t matrix[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
} 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" /* 见“栈和队列->栈”这节 */

/*
 * @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->nv;
    /* 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_INF)
                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_INF) {

```

```

        if(--in_degree[i] == 0) stack_push(&s, i);
    }
}

free(in_degree);
if(count != n) { /* 有环 */
    return 0;
} else { /* 无环 */
    return 1;
}
}

/** 读取输入, 构建图. */
void read_graph() {
    int i, j, k;
    char buf[10]; /* 和 gets() 配合消除回车键 */
    /* 读取节点和边的数目 */
    scanf("%d %d", &g.nv, &g.ne);
    gets(buf); // 消耗回车键

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

    /* 读取边信息 */
    for (k = 0; k < g.ne; k++) {
        char chx, chy;
        graph_weight_t w;
        scanf("%c %c %d", &chx, &chy, &w);
        gets(buf); // 消耗回车键
        i = chx - 'A';
        j = chy - 'A';
        g.matrix[i][j] = w;
    }
}

```

/* 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
*/
int main() {
    int i;
    read_graph();
    /* 拓扑排序 */
    mgraph_topo_sort(&g, topological);
    for (i = 0; i < g.nv; i++) {
        printf("%c ", 'A'+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。

13.5.1 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

```
#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_INF INT_MAX

/**
 * @struct
 * @brief 邻接矩阵.
 */
typedef struct mgraph_t {
    int nv; /* 顶点数 */
    int ne; /* 边数 */
    /* 邻接矩阵，存放边的信息，如权重等 */
    graph_weight_t matrix[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
} 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" /* 见“栈和队列->栈”这节 */

/**
 * @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->nv;
    /* 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_INF)
            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_INF) {
        --in_degree[i];
    }
    // 选择入度为 0 的顶点
    for (i = 0; i < n; i++) if (g->matrix[u][i] < GRAPH_INF) {
        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.nv), &m)
        && g.nv > 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

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

/* 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.nv; i++) {
            printf("%c", 'A' + 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>

13.6 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 $\langle j, k \rangle$ 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 $dut(\langle j, k \rangle)$, 则有如下关系

$$\begin{aligned} e(i) &= ve(j) \\ l(i) &= vl(k) - dut(\langle j, k \rangle) \end{aligned}$$

求 $ve(j)$ 和 $vl(k)$ 需分两步进行:

1. 从 $ve(0) = 0$ 开始向前递推

$$ve(j) = \max \{ve(i) + dut(\langle i, j \rangle)\}, \langle i, j \rangle \in T$$

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

$$vl(j) = \min \{vl(k) - dut(\langle j, k \rangle)\}, \langle j, k \rangle \in S$$

其中 S 是所有以顶点 j 为弧尾的边的集合。

例如, 对图13-6(a) 所示 AOE 网络的计算过程如表13-4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13-6(b) 所示。

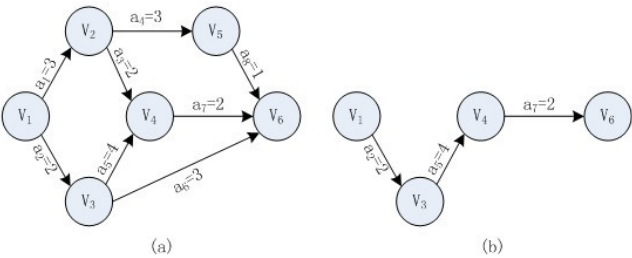


图 13-6 有向图及其邻接矩阵

表 13-4 图13-6(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

```
#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_INF INT_MAX

/**
 * @struct
 * @brief 邻接矩阵.
 */
typedef struct mgraph_t {
    int nv; /* 顶点数 */
    int ne; /* 边数 */
    /* 邻接矩阵，存放边的信息，如权重等 */
```

```

    graph_weight_t matrix[MAX_VERTICES_NUM][MAX_VERTICES_NUM];
} 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" /* 见“栈和队列->栈”这节 */

/*
 * @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->nv;
    /* 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_INF)
                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_INF) {
            --in_degree[i];
        }
    }
}

```

```

    }
    // 更新邻接点的 ve
    for (i = 0; i < n; i++) if (g->matrix[u][i] < GRAPH_INF) {
        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_INF) {
        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->nv * sizeof(graph_weight_t));
    graph_weight_t *vl = (graph_weight_t*) malloc(
        g->nv * 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->nv; i++) vl[i] = ve[g->nv-1];

    // 逆序计算 vl
    for (i = g->nv-1; i >= 0; i--) {
        int k = topological[i];
        for (j = 0; j < g->nv; j++) {
            if (g->matrix[j][k] < GRAPH_INF) {
                if (vl[j] > vl[k] - g->matrix[j][k])
                    vl[j] = vl[k] - g->matrix[j][k];
            }
        }
    }
    for (i = 0; i < g->nv; i++) {
        for (j = 0; j < g->nv; 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;
}

```

```

/** 读取输入, 构建图. */
void read_graph() {
    int i, j, k;

    /* 读取节点和边的数目 */
    scanf("%d%d", &g.nv, &g.ne);

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

    /* 读取边信息 */
    getchar(); // 消耗回车键
    for (k = 0; k < g.ne; k++) {
        char chx, chy;
        graph_weight_t w;
        scanf("%c %c %d", &chx, &chy, &w);
        getchar();
        i = chx - 'A';
        j = chy - 'A';
        g.matrix[i][j] = w;
    }
}

```

```

/* 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, count;
    read_graph();
    /* 拓扑排序 */
    count = mgraph_critical_path(&g, path);
    for (i = 0; i < count; i++) {
        printf("%c ", 'A' + 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 $O(n^2)$ 。

第 14 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。

14.1 数论

14.1.1 欧几里德算法

求最大公约数 (greatest 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。

```
/**
 * @brief 求最大公约数，欧几里德算法，也即辗转相除法
 *
 * @param[in] a a
 * @param[in] b b
 * @return a 和 b 的最大公约数
 */
unsigned int gcd(unsigned int a, unsigned 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
 * @param[in] b b
 * @return a 和 b 的最大公约数
 */
unsigned int gcd1(unsigned int a, unsigned int b) {
    while (b != 0) {
        unsigned int tmp = b;
        b = a % b;
        a = tmp;
    }
}
```

^①http://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
 * @param[in] b b
 * @return a 和 b 的最大公约数
 */
unsigned int gcd2(unsigned int a, unsigned 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), $\text{lcm}(a, b) = a \times b / \text{gcd}(a, b)$ 。

例题

- wikioi 1212 最大公约数, <http://www.wikioi.com/problem/1212/>

14.1.2 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 $\text{gcd}(a, b) = ax + by$ 。

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。

```

/**
 * @brief 扩展欧几里德算法
 * @param[in] a a
 * @param[in] b b
 * @param[out] x
 * @param[out] y
 * @return gcd(a,b)
 */
unsigned int ex_gcd(unsigned int a, unsigned int b, int *x, int *y) {
    if (b == 0) {
        *x = 1; *y = 0; return a;
    }

```

ex_gcd.c

^①http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm

```

    } else {
        const unsigned int tmp = ex_gcd(b, a % b, y, x);
        *y -= (*x)*(a/b);
        return tmp;
    }
}

```

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；

求解不定方程

求不定方程 $ax + by = c$ 的整数解 x, y ，其中 $a > b > c \geq 0$ 。

设 $g = \gcd(a, b)$ ，方程 $ax + by = g$ 的一组解是 (x_0, y_0) ，则当 c 是 g 的倍数时 $ax + by = c$ 的一组解是 (x_0, y_0) 。证明略。

```

/**
 * @brief 求解不定方程 ax+by=c
 * @param[in] a a
 * @param[in] b b
 * @param[in] c c
 * @param[out] x x
 * @param[out] y y
 * @return 是否有解, 1 表示有解, 0 表示无解
 */
int linear_equation(unsigned int a, unsigned int b, unsigned int c,
    int *x, int *y) {
    unsigned int k;
    const unsigned int g = ex_gcd(a, b, x, y);
    if (c % g) return 0;

    k = c / g;
    (*x) *= k; (*y) *= k;
    return 1;
}

```

求解模线性方程

解方程 $ax \equiv b \pmod{n}$ ，其中 a, b, n 是正整数。

$ax \equiv b \pmod{n}$ 的含义是“ ax 和 b 除以 n 的余数相同”，设这个倍数为 y ，则 $ax - b = ny$ ，这恰好就是前面介绍过的不定方程。接下来的步骤就不用说了吧。


```

/**
 * @brief 求解模线性方程  $ax = b \pmod n$ 
 * @param[in] a
 * @param[in] b
 * @param[in] n  $n > 0$ 
 */
int modular_linear_equation(unsigned int a, unsigned int b, unsigned int n) {
    int x, y, x0, i;
    unsigned int k;
    unsigned int g = ex_gcd(a, n, &x, &y);
    if(b % g) return 0;

    k = b / g;
    x0 = (k * x) % n; //特解
    for (i = 0; i < g; i++)
        printf("%d\n", (x0 + i * (n/g)) % n);
    return 1;
}

```

求解模的逆元

有一个特殊情况要引起读者重视，当 $b = 1$ 时， $ax \equiv 1 \pmod n$ 的解称为 a 关于模 n 的逆 (inverse)，它类似于“倒数”的概念。什么时候 a 的逆存在呢？根据上面的讨论，方程 $ax - ny = 1$ 要有解，这样，1 必须是 $\gcd(a, n)$ 的倍数，因此 a 和 n 必须互素。

14.1.3 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(unsigned int n) {
    int i;
    if (n < 2) return 0;
    for (i = 2; i < n; i++) {
        if (n % i == 0) return 0;
    }
    return 1;
}

```

^①http://en.wikipedia.org/wiki/Primality_test

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(unsigned int n) {
    int i;
    if (n < 2) return 0;
    const int upper = sqrt(n);

    for (i = 2; i <= upper; i++) {
        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(unsigned 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, \dots, 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.....。

```
#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);
```

^①http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

```

    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(unsigned int n) {
    return prime_table[n];
}

```

暴力枚举法优化版

下面这段代码是从 GNU GMP(<http://gmplib.org/>) 的源代码中抠出来的。

```

int is_prime(unsigned int n) {
    unsigned int q, r, d;    /* 商, 余数, 除数 */

    /* 把这个常数展开成二进制就容易理解了 */
    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;    /* 保证 d 不超过 sqrt(n) */
        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/>

14.1.4 大整数取模

描述

求 $a \bmod b, 1 \leq a \leq 10^{1000}, 1 \leq b \leq 100000$ 。

输入

每行 2 个整数 a 和 b

输出

对每一行输出结果 $a \bmod b$

样例输入

```
2 3
12 7
152455856554521 3250
```

样例输出

```
2
5
1521
```

代码

```
#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 (1000/MOD_LEN+1) /* 整数的最大位数 */

char    a[MAX_LEN * MOD_LEN];
int     x[MAX_LEN], y;

/**
 * @brief 将输入的字符串转化为大整数，用数组表示，低位在低地址.
 * @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;
```

bigint_mod.c

```

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 x mod y
 * @param[in] x 大整数
 * @param[in] y 模
 * @return x mod y
 */
int bigint_mod(const int x[MAX_LEN], int y) {
    int ret = 0;
    int i;
    for (i = MAX_LEN-1; i >= 0; i--) {
        ret = (ret * BIGINT_MOD + x[i]) % y;
    }
    return ret;
}

int main() {
    while(scanf("%s%d", a, &y) > 1) {
        bigint_input(a, x);
        printf("%d\n", bigint_mod(x, y));
    }
    return 0;
}

```

bigint_mod.c

相关的题目

与本题相同的题目：

- HDU 1212 Big Number, <http://acm.hdu.edu.cn/showproblem.php?pid=1212>

与本题相似的题目：

- None

14.2 组合数学

第 15 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。

15.1 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

```
55555555555555555555
```

代码

bigint_add.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_RADIX 10000
#define RADIX_LEN 4
#define MAX_LEN (200/RADIX_LEN+1) /* 整数的最大位数 */

char    a[MAX_LEN * RADIX_LEN], b[MAX_LEN * RADIX_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，用数组表示，低位在低地址
 * @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 -= RADIX_LEN) { /* [i-RADIX_LEN, i) */
        int temp = 0;
```

```

        int k;
        const int low = i-RADIX_LEN > 0 ? i-RADIX_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_RADIX) { /* 看是否要进位 */
            z[i] -= BIGINT_RADIX;
            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

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_RADIX 10000
#define RADIX_LEN 4
#define MAX_LEN (100/RADIX_LEN+1) /* 整数的最大位数 */

char    a[MAX_LEN * RADIX_LEN], b[MAX_LEN * RADIX_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，用数组表示，低位在低地址
 * @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 -= RADIX_LEN) { /* [i-RADIX_LEN, i) */
        int temp = 0;
        int k;
        const int low = i-RADIX_LEN > 0 ? i-RADIX_LEN : 0;
        for (k = low; k < i; k++) {
            temp = temp * 10 + s[k] - '0';

```

```
    }

    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_RADIX;
            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：

• None

15.3 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_RADIX 10000
#define RADIX_LEN 4
#define MAX_LEN (200/RADIX_LEN+1) /* 整数的最大位数 */

char    a[MAX_LEN * RADIX_LEN], b[MAX_LEN * RADIX_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; /* 用于跳过后导 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, 用数组表示, 低位在低地址
 * @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 -= RADIX_LEN) { /* [i-RADIX_LEN, i) */
        int temp = 0;
        int k;
        const int low = i-RADIX_LEN > 0 ? i-RADIX_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_RADIX) { /* 看是否要进位 */
                z[i + j + 1] += z[i + j] / BIGINT_RADIX; /* 进位 */
                z[i + j] %= BIGINT_RADIX;
            }
        }
    }
}

```

```
}

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：

- None

15.4 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


```

    * @return 无
    */
void bigint_print(const int x[], const int n) {
    int i;
    int start_output = 0; /* 用于跳过后导 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, 用数组表示, 低位在低地址
 * @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 -= RADIX_LEN) { /* [i-RADIX_LEN, i) */
        int temp = 0;
        int k;
        const int low = i-RADIX_LEN > 0 ? i-RADIX_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_RADIX;
            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_RADIX, 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_RADIX) { /* 看是否要进位 */
        z[i+1] += z[i] / BIGINT_RADIX; /* 进位 */
        z[i] %= BIGINT_RADIX;
    }
}
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/>
- wikioi 3118 高精度练习之除法, <http://poj.grids.cn/practice/3118/>

与本题相似的题目：

- None

15.5 大数阶乘

15.5.1 大数阶乘的位数

描述

求 $n!$ 的位数, $0 \leq n \leq 10^7$ 。

输入

第一行是一个正整数 T , 表示测试用例的个数。接下来的 T 行, 每行一个正整数 n 。

输出

对每个 n , 每行输出 $n!$ 的位数

样例输入

```
2  
10  
20
```

^①李文新, 程序设计导引及在线实践, 清华大学出版社, 2007

样例输出

```
7
19
```

分析

最简单的办法，是老老实实计算出 $n!$ ，然后就知道它的位数了。但这个方法很慢，会超时 (TLE)。

组合数学里有个 Stirling 公式 (Stirling's formula^①):

$$\lim_{n \rightarrow \infty} \frac{n!}{\sqrt{2\pi n} \left(\frac{n}{e}\right)^n} = 1$$

可以用这个公式来计算 $n!$ 的位数，它等于

$$n \log \frac{n}{e} + \frac{1}{2} \log 2\pi n + 1$$

代码

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

/**
 * @brief 计算 n 的阶乘的位数，用 Stirling 公式
 * @param[in] n n>=0
 * @return n 的阶乘的位数
 */
int factorial_digits(unsigned int n) {
    const double PI = 3.14159265358979323846;
    const double E = 2.7182818284590452354;
    if (n == 0) return 1;
    return (int)(n * log10(n/E) + 0.5 * log10(2*PI*n)) + 1;
}

int main() {
    int i, T, n;

    scanf("%d", &T);
    for (i = 0; i < T; i++) {
        scanf("%d", &n);
        printf("%d\n", factorial_digits(n));
    }
    return 0;
}
```

bigint_factorial_digits.c

bigint_factorial_digits.c

^①http://en.wikipedia.org/wiki/Stirling's_approximation

相关的题目

与本题相同的题目：

- POJ 1423 Big Number, <http://poj.org/problem?id=1423>

与本题相似的题目：

- None

15.5.2 大数阶乘

描述

计算 $n!$, $0 \leq n \leq 10000$ 。

输入

每行一个整数 n

输出

对每个 n ，每行输出 $n!$

样例输入

```
1
2
3
```

样例输出

```
1
2
6
```

代码

```
#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_RADIX 10000
#define RADIX_LEN 4
/* 10000! 有 35660 位 */
#define MAX_LEN (35660/RADIX_LEN+1) /* 整数的最大位数 */
```

bigint_factorial.c

```

int      x[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 大整数乘法, x = x*y.
 * @param[inout] x x
 * @param[in] y y
 * @return 无
 */
void bigint_mul(int x[], const int y) {
    int i;
    int c = 0; /* 保存进位 */

    for (i = 0; i < MAX_LEN; i++) { /* 用 y, 去乘以 x 的各位 */
        const int tmp = x[i] * y + c;
        x[i] = tmp % BIGINT_RADIX;
        c = tmp / BIGINT_RADIX;
    }
}

/**
 * @brief 计算 n 的阶乘
 * @param[in] n
 * @param[out] x 存放结果
 * @return 无
 */
void bigint_factorial(int n, int x[]) {
    int i;
    memset(x, 0, sizeof(int) * (MAX_LEN + 1));
    x[0] = 1;

    for (i = 2; i <= n; i++) {

```

```
        bigint_mul(x, i);
    }
}

int main() {
    int n;
    while (scanf("%d", &n) != EOF) {
        bigint_factorial(n, x);
        bigint_print(x, MAX_LEN + 1);
        printf("\n");
    }
    return 0;
}
```

bigint_factorial.c

相关的题目

与本题相同的题目：

- HDU 1042 N!, <http://acm.hdu.edu.cn/showproblem.php?pid=1042>

与本题相似的题目：

- None

第 16 章

字符串

16.1 字符串排序

16.2 单词查找树

16.3 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。

16.3.1 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。

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

kmp.c

^①《算法》, Robert Sedgewick, 人民邮电出版社, <http://book.douban.com/subject/10432347/>

```

#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;
}

```

```

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

16.3.2 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。

```

/**
 * 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;
}

```

boyer_moore.c

^①《算法》, Robert Sedgewick, 人民邮电出版社, <http://book.douban.com/subject/10432347/>

^②BOYER 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

16.3.3 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。

```

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

```

rabin_karp.c

^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 = 0xffff;

/*
 * @brief 哈希函数.
 *
 * @param[in] key 待计算的字符串
 * @param[in] 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

16.3.4 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

16.4 正则表达式

第 17 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。

17.1 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 17-1 所示（来自 <http://fisherlei.blogspot.com/2012/12/leetcode-next-permutation.html>）。



1. From right to left, find the first digit (PartitionNumber) which violate the increase trend, in this example, 6 will be selected since 8,7,4,3,2 already in a increase trend.
2. From right to left, find the first digit which large than PartitionNumber, call it changeNumber. Here the 7 will be selected.
3. Swap the PartitionNumber and ChangeNumber.
4. Reverse all the digit on the right of partition index.

图 17-1 下一个排列算法流程

代码

```

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) {

```

next_permutation.c

```

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

17.2 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：

```

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

```
    }
}
```

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：

```
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

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：

```
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;
```

ror.c

```
    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：

```
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;
```

ror.c

```
    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;
}
```