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 2013-5-16

版权声明

内容简介

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

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

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

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

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

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

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

更新记录

2013-04-17 v0.2

2013-04-14 v0.1

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

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

目录

第 1 章 编程技巧

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

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

输入数据一般放在全局变量，且在运行过程中不要修改这些变量。

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

第 2 章
线性表

线性表 (Linear List) 包含以下几种:

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

第 3 章 栈和队列

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

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

3.1 栈

3.1.1 栈的 C 语言实现

C++ 可以直接使用 `std::stack`。

stack.c

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

typedef int stack_elem_t; // 元素的类型

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

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

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

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

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

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

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

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

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

stack.c

3.1.2 Hanoi 塔问题

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

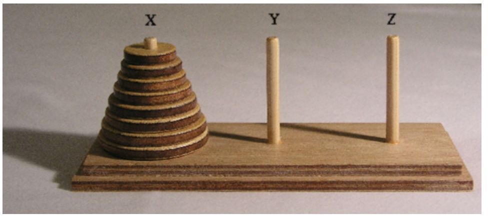


图 3-1 Hanoi 塔问题

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

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

递归代码如下：

hanoi.c

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

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

3.1.3 数制转换

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

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

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

```
        }
        return;
}

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

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

3.2 队列

3.2.1 队列的 C 语言实现

C++ 可以直接使用 `std::queue`。

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

#ifndef __cplusplus
```

```c
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

binary_tree.cpp

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

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

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

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

```
    }
}

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

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

    p = root;

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

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

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

    p = root;

    while(!s.empty() || p!=NULL) {
```

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

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

    p = root;

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

/**
 * @brief 层次遍历，也即 BFS.
 *
 * 跟先序遍历一模一样，唯一的不同是栈换成了队列
 */
void level_order(const binary_tree_node_t *root,
```

```
                int (*visit)(void*)) {
    const binary_tree_node_t *p;
    std::queue<const binary_tree_node_t *> q;

    p = root;

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

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

binary_tree.cpp

4.2 重建二叉树

binary_tree.cpp

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

// 测试
// BCAD CBAD, 输出 CDAB
// DBACEGF ABCDEFG, 输出 ACBFGED
int build_test() {
    const int MAX = 64;
    char pre[MAX] = {0};
    char in[MAX] = {0};
    char post[MAX] = {0};
```

```
    scanf("%s%s", pre, in);

    const int n = strlen(pre);
    build(n, pre, in, post);
    printf("%s\n", post);
}
```
binary_tree.cpp

4.3 堆

4.3.1 堆的 C 语言实现

C++ 可以直接使用 `std::priority_queue`。

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

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

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;

/**
 * @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 true，否则返回 false
 */
bool heap_empty(const heap_t *h) {
    if(h != NULL) {
        return h->size == 0;
    } else {
        return false;
    }
}

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

/*
 * @brief 小根堆的自上向下筛选算法.
 * @param[in] h 堆对象的指针
 * @param[in] start 开始结点
 * @return 无
```

```
 */
void heap_sift_down(const heap_t *h, const int start) {
    int i = start;
    int j;
    const heap_elem_t tmp = h->elems[start];

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

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

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

/**
 * @brief 添加一个元素.
 * @param[in] h 堆对象的指针
 * @param[in] x 要添加的元素
 * @return 无
 */
```

```c
void heap_push(heap_t *h, const heap_elem_t x) {
    if(h->size == h->capacity) { /* 已满，重新分配内存 */
        heap_elem_t* tmp =
            (heap_elem_t*)realloc(h->elems, h->capacity * 2 * sizeof(heap_elem_t));
        h->elems = tmp;
        h->capacity *= 2;
    }

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

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

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

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

heap.c

第 5 章
图

在 ACM 竞赛中，图一般使用邻接矩阵表示，代码框架如下；

graph.c
```c
#define MAXN 100  // 顶点最大个数

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

5.1 深度优先搜索

图的深度优先搜索的代码框架如下：

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

/**
 * @brief 图的深度优先搜索代码框架，搜索顶点.
```

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

graph.c

5.1.1 黑白图像

描述

输入一个 n*n 的黑白图像（1 表示黑丝，0 表示白色），任务是统计其中八连块的个数。如果两个黑格子有公共边或者公共定点，就说它们属于同一个八连块。如图 5-1所示的黑白图像中有 3 个八连块。

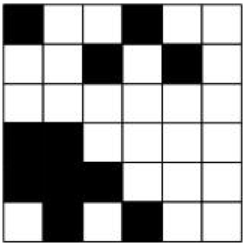


图 5-1 拥有 3 个八连块的黑白图

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

类似的题目

与本题相同的题目:

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

与本题相似的题目:

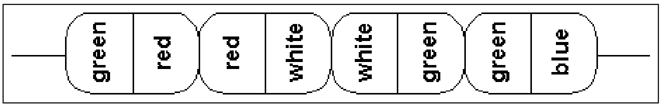
- TODO

5.1.2 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 $N (5 \le N \le 1000)$ 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 $1 \le i \le N_1$, 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

```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

类似的题目

与本题相同的题目:

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

与本题相似的题目:

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

5.2 广度优先搜索

我们通常说的 BFS，默认指的是单向 BFS，此外还有双向 BFS。

5.2.1 走迷宫

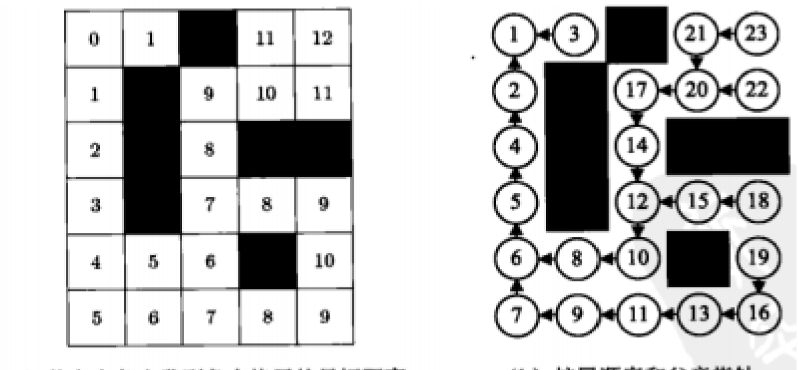
描述

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

分析

既然求的是“最短”，很自然的思路是用 BFS。举个例子，在如下图所示的迷宫中，假设入口是左上角 $(0, 0)$，我们就从入口开始用 BFS 遍历迷宫，就可以算出从入口到所有点的最短路径（如图 5-2(a) 所示），以及这些路径上每个节点的前驱（如图 5-2(b) 所示）。

[1] 刘汝佳, 算法竞赛入门经典，清华大学出版社，2009



（a）从左上角出发到各个格子的最短距离　（b）扩展顺序和父亲指针

图 5-2　用 BFS 求迷宫中最短路径

代码

maze.c

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

#define MAXN 100

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

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

// 队列
int q[MAXN * MAXN];

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

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

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

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

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

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

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

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

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

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

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

类似的题目

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

5.2.2 八数码问题

描述

编号为 1~8 的 8 个正方形滑块摆成 3 行 3 列，有一个格子空着，如图 5-3所示。

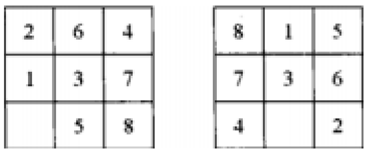


图 5-3 用 BFS 求迷宫中最短路径

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

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

给定初始局面，计算出最短的移动路径。

输入

用一行表示一个局面，例如下面的这个局面:

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

可以表示为 1 2 3 x 4 6 7 5 8。

输出

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

①刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

样例输入

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

样例输出

```
ullddrurdllurdruldr
```

分析

计算“最短”，很自然的想到 BFS。

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

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

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

此题更优的解法还有双向 BFS（见 §5.3），A* 算法（见 §5.8）。

代码

eight_digits_bfs.c

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

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

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

typedef int state_t[DIGITS]; // 单个状态

state_t q[MAX]; // 队列，也是哈希表
```

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

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

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

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

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

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

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

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

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

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

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

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

        front++;
    }

    return 0;//失败
}


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

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

int main() {
    int ans;

    input();

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

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

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

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

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

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

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

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

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

    return 1;
}

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

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

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

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

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

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

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

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

eight_digits_bfs.c

类似的题目

与本题相同的题目:

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

与本题相似的题目:

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

5.3 双向 BFS

5.3.1 八数码问题

题目见 §5.2.2。

代码

eight_digits_bibfs.c

eight_digits_bibfs.c

[①] 刘汝佳, 算法竞赛入门经典, 清华大学出版社, 2009

5.4 最小生成树

5.4.1 Prim 算法

5.4.2 Kruskal 算法

5.5 最短路径

5.5.1 单源最短路径 (Dijkstra 算法)

5.5.2 每点最短路径 (Floyd 算法)

5.6 拓扑排序

5.7 关键路径

5.8 A* 算法

5.8.1 八数码问题

题目见 §5.2.2。

代码

eight_digits_astar.c

```
/**
 简单解释几个要点，便于理解代码.
 1. 怎么判断是否有解？只要计算出的逆序个数总和为奇数，该数据必然无解
 2. 如何判断某一状态是否到过？本题存在一种完美哈希方案，即用康托展开。
    详见 http://128kj.iteye.com/blog/1699795
    2.1. 将一个状态视为数字 0-8 的一个排列，将此排列转化为序数，作为此状
 态的 HASH 值。0 表示空格. 转化算法此处不再赘述。

    2.2. 排列转化为序数，用序数作为 hash 值
    例，1 2 3 这三个数字的全排列，按字典序，依次为
 123 -- 0
 132 -- 1
 213 -- 2
 231 -- 3
 312 -- 4
 321 -- 5
 其中，左侧为排列，右侧为其序数。

 3. 使用数据结构 堆 加速挑选最优值。
```

```
 4. 函数 g 的计算，此状态在搜索树中已经走过的路径的节点数.

 5. 估价函数 h ，采用曼哈顿距离，见代码 calcH 函数。曼哈顿距离的定义是，
 假设有两个点 (x1,y1),(x2,y2)，则曼哈顿距离 L1=|x1-x2| + |y1-y2|
 */
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// 3x3 的棋盘，状态最多有 9! 种
// 8 位变进制数（空格）能表示 0 到 (9!-1) 内的所有自然数，恰好有 9! 个,
// 与状态一一对应，因此可以把状态一一映射到一个 8 位变进制数
#define     MAX         362880

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

#define     MOD         10      // 按十取模

typedef struct {
    int state; // 状态
    int parent;     // 父状态
    int flag;   // -1 表示已经展开过了 closed，0 表示死节点，1 表示还未展开，open
    int g, h, f; // 三个评估函数
    char choice;  // 左右上下四个方向移动，见全局常量 DI DJ DC
} state_t;

state_t states[MAX];  // 全局的一条状态变化路径

int  startIndex, goalIndex; // 开始状态，目标状态对应的 hash 值

// 目标状态
const int goal = 123456780;
// 每个数字在棋盘中的位置，例如 0，在 (1,1)=4 这个位置上
const int goal_pos[DIGITS] = {8,0,1,2,3,4,5,6,7};
const int space_number = 0; // 空格对应着数字 0

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

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

/**
 * @brief 计算状态的 hash 值，这里用康托展开，是完美哈希.
 *
 * @param[in] s 当前状态
 * @return 序数，作为 hash 值
 */
int hash(int s) {
```

```
    int i, j;
    int d[DIGITS];
    int key = 0;

    for(i = DIGITS - 1; i >=0; i--) {
        d[i] = s % MOD;
        s /= MOD;
    }

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

    return key;
}


/**
 * 估价函数 h。
 * @param s 状态
 * @return 预估代价
 */
int calcH(int s) {
    int i;
    int h = 0;

    for (i = DIGITS - 1; i >= 0; --i) {
        const int p = s % 10;
        s /= 10;
        // 曼哈顿距离
        h += abs(i / MATRIX_EDGE - goal_pos[p] / MATRIX_EDGE) +
            abs(i % MATRIX_EDGE - goal_pos[p] % MATRIX_EDGE);
    }
    return h;
}

/**
 * @brief 输入.
 * @return  成功返回数字，失败返回 0
 * @remark 《算法竞赛入门经典》第 131 页 7.5.3 节，是用 0 表示空格，
 * POJ 1077 是用'x' 表示空格，前者简化了一点，POJ 1077 还需要把'x' 映射成 0
 */
int input() {
    int ch, i;
    int start = 0;
    for (i = 0; i < DIGITS; ++i) {
        do {
```

```c
            ch = getchar();
        } while ((ch != EOF) && ((ch < '1') || (ch > '8')) && (ch != 'x'));
        if (ch == EOF) return 0;
        if (ch == 'x') start = start * MOD + space_number; // x 映射成数字 0
        else           start = start * MOD + ch - '0';
    }
    return start;
}

/**
 * 计算一个排列的逆序数，0 除外.
 */
int inversion_count(int permutation) {
    int i, j;
    int d[DIGITS];
    int c = 0; // 逆序数

    for(i = DIGITS - 1; i >=0; i--) {
        d[i] = permutation % MOD;
        permutation /= MOD;
    }

    for (i = 1; i < DIGITS; i++)  if (d[i] != space_number) {
        for (j = 0; j < i; j++) {
            if(d[j] != space_number) {
                if (d[j] > d[i]) {
                    c++;
                }
            }
        }
    }
    return c;
}

/**
 * 判断是否无解.
 *
 * 求出除 0 之外所有数字的逆序数之和，也就是每个数字后面比它小的数字的个数的和，
 * 称为这个状态的逆序。若两个状态的逆序奇偶性相同，则可相互到达，否则不可相互到达。
 * 由于原始状态的逆序数为 0（偶数），因此逆序数为偶数的目标状态有解。
 *
 * @param s 目标状态
 * @return 1 表示无解，0 表示有解
 */
int not_solvable(const int s) {
    return inversion_count(s) % 2;
}

// 存放 next() 的输出结果
char choice[4]; // 四个防线
int nextIndex[4]; // 接下来的四个状态

/*
```

```
 * @brief 向四个方向扩展
 * @param[in] s 状态
 * @return 无
 */
void next(int s) {
    int i, j, k;
    int p[MATRIX_EDGE][MATRIX_EDGE]; // 一个状态对应的矩阵
    int i0, j0;  // 空格位置

    for (i = MATRIX_EDGE - 1; i >= 0; i--) {
        for (j = MATRIX_EDGE - 1; j >= 0; j--) {
            p[i][j] = s % MOD;
            s /= MOD;
            if (p[i][j] == space_number) {
                i0 = i;
                j0 = j;
            }
        }
    }
    // 向四个方向探索
    for (k = 0; k < 4; ++k) {
        const int sx = i0 + DI[k]; // 空格的新位置 (sx, sy)
        const int sy = j0 + DJ[k];
        if ((sx >= 0) && (sx < 3) && (sy >= 0) && (sy < 3)) {
            int key;
            p[i0][j0] = p[sx][sy];
            p[sx][sy] = space_number;
            // 移动空格后，计算新的状态
            s = 0;
            for (i = 0; i < MATRIX_EDGE; i++)
                for (j = 0; j < MATRIX_EDGE; j++)
                    s = s * MOD + p[i][j];
            p[sx][sy] = p[i0][j0]; // 将矩阵还原，(i0, j0) 可以不管

            key = nextIndex[k] = hash(s);
            choice[k] = DC[k];
            if (states[key].state == 0) { // 该状态还没有出现过
                states[key].state = s;
                states[key].h = calcH(s);
            }
        } else {// 越界了
            nextIndex[k] = -1;
        }
    }
}

int cmpInt(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;
    }
}

#include "heap.c" // 相当于复制粘贴
heap_t heap;
int heapIndex[MAX + 4]; // 状态 x 在 heap 中的下标

/**
 * @brief A* 搜索
 * @param[in] start 初始状态
 * @return 如果无解，返回 0，如果有解返回 1
 */
int astar(const int start) {
    int i, j, k, ng, nf;
    if (not_solvable(start)) return 0;

    startIndex = hash(start);
    goalIndex  = hash(goal);
    if (start == goal) return 1;

    memset(states, 0, sizeof(states));
    states[startIndex].state = start;
    states[startIndex].flag    = 1;
    states[startIndex].g       = 0;
    states[startIndex].h       = states[startIndex].f = calcH(start);

    heap_push(&heap, startIndex);
    while(!heap_empty(&heap)) {
        i = heap_top(&heap); heap_pop(&heap);
        if (i == goalIndex) return 1; // 找到目标，返回

        states[i].flag = -1;
        ng = states[i].g + 1;
        next(states[i].state);
        for (k = 0; k < 4; ++k) {
            j = nextIndex[k];
            if (j < 0) continue;
            nf = ng + states[j].h;
            if ((states[j].flag == 0) || ((states[j].flag == 1) &&
                (nf < states[j].f))) {
                states[j].parent = i;
                states[j].choice = choice[k];
                states[j].g    = ng;
                states[j].f    = nf;
                if (states[j].flag > 0) {
                    heap_sift_up(&heap, heapIndex[j]);
                    heap_sift_down(&heap, heapIndex[j]);
                } else {
                    heap_push(&heap, j);
                    states[j].flag = 1;
                }
            }
```

```c
        }
    }
    return 0;
}

#include "stack.c" // 相当于复制粘贴
/**
 * @brief 打印移动序列.
 * @return 无
 */
void output() {
    int i;
    stack_t stack;
    stack_init(&stack, MAX);

    for (i = goalIndex; i != startIndex; i = states[i].parent) {
        stack_push(&stack, states[i].choice);
    }
    while(!stack_empty(&stack)) {
        printf("%c", stack_top(&stack));
        stack_pop(&stack);
    }

    printf("\n");
    stack_uninit(&stack);
}

/**
 * @brief 打印棋盘的每次变化.
 * @return 无
 */
void output1() {
    int i;
    int d[DIGITS];
    stack_t stack;
    stack_init(&stack, MAX);

    for (i = goalIndex; i != startIndex; i = states[i].parent) {
        stack_push(&stack, states[i].state);
    }
    stack_push(&stack, states[startIndex].state);

    while(!stack_empty(&stack)) {
        stack_elem_t tmp = stack_top(&stack);
        stack_pop(&stack);
        for(i = DIGITS - 1; i >=0; i--) {
            d[i] = tmp % MOD;
            tmp /= MOD;
        }
        for(i = 0; i < DIGITS; i++) {
            if((i + 1) % MATRIX_EDGE == 0) {
                printf("%d\n", d[i]);
            } else {
```

```
                printf("%d ", d[i]);
            }
        }
        printf("\n");
    }
    stack_uninit(&stack);
}

int main() {
    const int start = input();
    heap_init(&heap, MAX + 4, cmpInt);
    if (start > 0) {
        if (astar(start)) {
            output();
        } else {
            printf("no solution\n");
        }
    }
    heap_uninit(&heap);
    return 0;
}
```

eight_digits_astar.c

第 6 章 查找

6.1 折半查找

binary_search.c

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

binary_search.c

第 7 章 排序

7.1 快速排序

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

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
  * @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

第 8 章 暴力枚举法

8.1 算法思想

生成 –测试法。

8.2 简单枚举

8.2.1 分数拆分

输入正整数 k，找到所有的正整数 $x \geq y$，使得 $\frac{1}{k} = \frac{1}{x} + \frac{1}{y}$。

样例输入

2

12

样例输出

2

1/2=1/6+1/3

1/2=1/4+1/4

8 1/12=1/156+1/13

1/12=1/84+1/14

1/12=1/60+1/15

1/12=1/48+1/16

1/12=1/36+1/18

1/12=1/30+1/20

1/12=1/28+1/21

1/12=1/24+1/24

分析

既然说找出所有的 x, y，枚举对象自然就是他们了。可问题在于：枚举范围如何？从 $\frac{1}{12} = \frac{1}{156} + \frac{1}{13}$，可以看出，$x$ 可以比 y 大很多。难道要无休止地枚举下去？当然不是。由于 $x \geq y$，有 $\frac{1}{x} \leq \frac{1}{y}$，因此 $\frac{1}{k} - \frac{1}{y} \leq \frac{1}{y}$，即 $y \leq 2k$。这样，只需要在 $2k$ 范围之类枚举 y，然后根据 y 算出 x 即可。

8.3 枚举排列

8.3.1 生成 1 n 的排列

8.3.2 生成可重复集合的排列

8.3.3 下一个排列

8.4 子集生成

8.4.1 增量构造法

8.4.2 位向量法

8.4.3 二进制法

第 9 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)。

9.1 棋盘覆盖

9.2 循环赛日程表

第 10 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。

10.1 哈弗曼编码

10.1.1 POJ 1521 Entropy

描述

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

分析

题目很长，不过就是哈弗曼编码。

代码

poj1521_entropy.cpp

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

const int LINE_MAX = 256; // 一行最大字符数
const int MAX_ASCII = 128; // ASCII 码最大值
```

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

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

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

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

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

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

poj1521_entropy.cpp

类似的题目

与本题相同的题目:

- TODO

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

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

第 11 章
动态规划

如果一个问题具有以下两个要素：

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

则可以用动态规划求最优解。

动态规划分为 4 个步骤：

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

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

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

动规用表格将各个子问题的最优解存起来，避免重复计算，是一种空间换时间。

动规与贪心的相同点：最优子结构。

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

分治和贪心的相同点：disjoint subproblems。

11.1 最长公共子序列

描述

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

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

输入

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

输出

对每组测试数据，输出最大公共子序列的长度，每组一行。

样例输入

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

样例输出

```
4
2
0
```

分析

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

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

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

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

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

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

代码

lcs.c

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

#define MAX 201  /* 字符串最大长度为 200 */

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

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

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

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

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

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

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

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

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

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

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

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

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

lcs.c

类似的题目

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

11.2 DAG 上的动态规划

11.2.1 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 11-1所示。

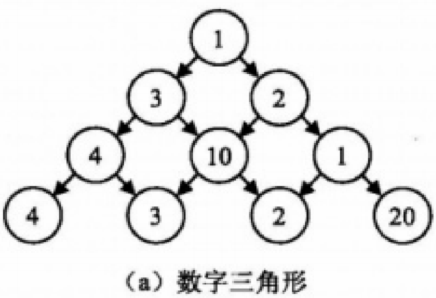


图 11-1 数字三角形问题

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

[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

输入

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

输出

输出最大的和。

样例输入

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

样例输出

```
30
```

分析

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

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

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

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

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

代码

版本 1，自顶向下。

numbers_triangle1.c

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

#define MAXN 100

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

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

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

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

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

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

numbers_triangle1.c

版本 2，自底向上。

numbers_triangle2.c

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

#define MAXN 100

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

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

/**
```

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

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

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

    dp();

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

numbers_triangle2.c

类似的题目

与本题相同的题目：

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

与本题相似的题目：

- TODO

11.2.2 嵌套矩形

描述

有 n 个矩形，每个矩形可以用 a, b 来描述，表示长和宽。矩形 $X(a, b)$ 可以嵌套在矩形 $Y(c, d)$ 中当且仅当 $a < c, b < d$ 或者 $b < c, a < d$（相当于旋转 X90 度）。例如（1,5）可

[①]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

以嵌套在（6,2）内，但不能嵌套在（3,4）中。你的任务是选出尽可能多的矩形排成一行，使得除最后一个外，每一个矩形都可以嵌套在下一个矩形内。

输入

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

输出

每组测试数据都输出一个数，表示最多符合条件的矩形数目，每组输出占一行

样例输入

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

样例输出

```
5
```

分析

本题实质上是求 DAG 中不固定起点的最长路径。

设 d[i] 表示从结点 i 出发的最长长度，状态转移方程如下:

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

其中，E 为边的集合。最终答案是 d[i] 中的最大值。

代码

embedded_rectangles.c

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

#define MAXN 1000 // 矩形最大个数

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

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

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

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

int main() {
    int N, i, j;
    int max, maxi;
    int a[MAXN],b[MAXN];

    scanf("%d", &N);
```

```
    while(N--) {
        memset(G, 0, sizeof(G));
        memset(d, 0, sizeof(d));

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

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

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

embedded_rectangles.c

类似的题目

与本题相同的题目：

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

与本题相似的题目：

- TODO

11.2.3 硬币问题

描述

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

输入

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

[①]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

输出

第 1 行为最小值，第 2 行为最大值。

样例输入

```
3
6
1
2
3
```

样例输出

```
2
6
```

分析

本题实质上是求 DAG 中固定终点的最长路径和最短路径。

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

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

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

本题还可以看作是完全背包问题（见 §11.3.2节）：背包容量为 S，背包必须要装满，物品即硬币，每个硬币的费用为面值 v_i，价值均为 1。求背包中物品的最小价值和最大价值。

代码

版本 1，自顶向下。

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

#define MAXN 100
#define MAXV 10000

/** 硬币面值的种类. */
int n;
/** 要找零的数目. */
int S;
/** 硬币的各种面值. */
int v[MAXN];
```

```
/** min[i] 表示面值之和为 i 的最短路径的长度, max 则是最长. */
int min[MAXV + 1], max[MAXV + 1];

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

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

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

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

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

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

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

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

    return 0;
}

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

coin_change.c

版本 2，自底向上。

coin_change2.c

```
#include<stdio.h>

#define MAXN 100
#define MAXV 10000

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

void dp() {
    int i, j;

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

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

```c
                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

```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

类似的题目

与本题相同的题目：

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

与本题相似的题目：

- TODO

11.2.4 最长上升子序列

描述

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

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

输入

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

输出

最长上升子序列的长度。

样例输入

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

[1]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

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；

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

代码

lis.c

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

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

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

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

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

    dp();

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

lis.c

类似的题目

与本题相同的题目：

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

与本题相似的题目:

- TODO

11.3 背包问题

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

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

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

背包问题是一种“多阶段决策问题”。

11.3.1 0-1 背包问题

描述

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

输入

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

输出

每行一个整数，表示价值总和的最大值。

[①] 刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

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

样例输入

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

样例输出

```
14
```

分析

由于每种物品仅有一个，可以选择装或者不装。

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

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

这个方程理解如下，把前 i 个物品装进容量为 j 的背包时，有两种情况:

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

动规过程的伪代码如下:

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

内循环从右向左也可以:

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

内循环从右向左时，可以把二维数组优化成一维数组。伪代码如下:

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

为什么呢？举个例子，测试数据如下:

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

f 是从上到下、从右到左计算的，如图 11-2所示。

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

图 11-2 0-1 背包问题的计算顺序

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

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

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

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

有了这个函数以后，0-1 背包问题的伪代码就可以这样写：

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

代码

版本 1，自底向上。

01knapsack.c

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

#define MAXN 1000
#define MAXW 1000

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

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

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

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

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

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

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

01knapsack.c

版本 2，自底向上，滚动数组。

01knapsack2.c

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

#define MAXN 1000
#define MAXW 1000

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

int d[MAXW + 1]; /* 滚动数组 */

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

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

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

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

        dp();
        printf("%d\n", d[W]);
    }
    return 0;
}
```
01knapsack2.c

类似的题目

与本题相同的题目:

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

与本题相似的题目:

- TODO

11.3.2 完全背包问题

描述

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

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

输入

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

[①]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009

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

输出

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

样例输入

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

样例输出

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

分析

每种物品有无限个可用，这是完全背包问题。

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

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

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

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

伪代码如下:

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

也可以写成:

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

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

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

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

还存在更优化的算法，复杂度为 $O(NW)$，伪代码如下:

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

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

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

根据上面的伪代码，状态转移方程也可以写成这种形式:

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

抽象出处理单个物品的函数:

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

代码

piggy_bank.c

```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

类似的题目

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

11.3.3 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：

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

伪代码如下：

```
for i = 1..N
    for j = W..w
        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[], w, v)
```

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

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

```
def BoundedKnapsack(d[], i)
    if c[i]*w[i] >= W
        unbounded_knapsack(d[], w[i], v[i]);
        return;

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

    zero_one_knapsack(d[], c[i] * w[i], c[i] * v[i]);
```

代码

bkp.c

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

#define MAXN 100
```

```
#define MAXW 100

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

int d[MAXW + 1]; /* 滚动数组 */

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

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

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

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

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

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

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

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

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

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

类似的题目

与本题相同的题目:

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

与本题相似的题目:

- TODO

第 12 章
回溯法

12.1　算法思想

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

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

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

12.2　八皇后问题

描述

在 8×8 的棋盘上，放置 8 个皇后，使得她们互不攻击，每个皇后的攻击范围是同行、同列和同对角线，要求找出所有解。如图 12-1所示。

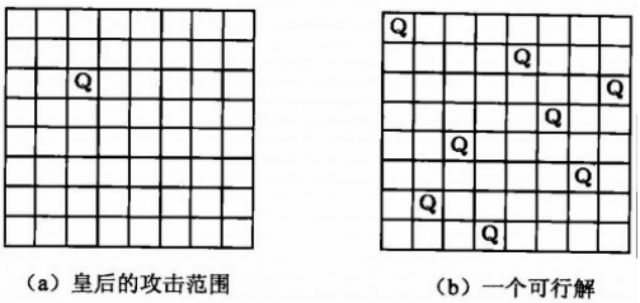


图 12-1　八皇后问题

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，$\cdots$，第 8 行有 1 中选择，总共有 $8! = 40320$ 个方案。如果用 C[x] 表示第 x 行皇后的列编号，则问题变成了一个全排列生成问题，枚举量不会超过 8!。

代码

eight_queen.c

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

#define QUEENS 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[QUEENS];          // 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 < QUEENS; ++j) {
        for (i = 0; i < QUEENS; ++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 < QUEENS; ++i) {
        for (j = 0; j < QUEENS; ++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), (j, C[j])
        // 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，该再哪一列上放一个皇后
 * @return 无
 */
void search(const int row) {
    if(row == QUEENS) {  // 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 < QUEENS; ++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);
```

```
            }
        }
    }
}

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

/**
 * @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 + QUEENS] == 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, 该再哪一列上放一个皇后
 * @return 无
 */
void search2(const int row) {
    if(row == QUEENS) {  // 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 < QUEENS; ++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 + QUEENS] = 1;
                search2(row + 1);
                // 恢复环境
                columns[j] = principal_diagonals[row + j] =
                    counter_diagonals[row - j + QUEENS] = 0;
            }
        }
    }
}

int main() {
    // search(0);
```

```
    search2(0);
    return 0;
}
```

eight_queen.c

类似的题目

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/

与本题相似的题目:

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

[1]刘汝佳, 算法竞赛入门经典，清华大学出版社，2009