Online Judge Notes

Sun Yufei *

2019 年 10 月 12 日

目　录

*GitHub: https://github.com/SunYufei

1 语言技巧

1.1 C++ 头文件引入技巧

使用 C++ 编写程序时只需引入如下头文件，包含 STL、C/C++ I/O、math.h、string.h 等。

```
#include <bits/stdc++.h>
using namespace std;
```

1.2 C++ 格式化输入输出

对于 C 语言中的输入输出函数

```
scanf("格式化字符串", <参量表>);
printf("格式化字符串", <参量表>);
```

格式化字符串中的符号规则如下表所示：

符号	说 明
%	格式说明起始符号，不可缺少
-	有：左对齐输出；无：右对齐输出
0	有：指定空位为 0；无：指定空位为空格
m.n	*m*：输出项在输出设备上所占字符数 *n*：精度，实数的小数位数，默认 $n=6$
l	int→long；float→double
h	int→short
%d	十进制 int
%ld	十进制 long
%lld	十进制 long long
%u	十进制 unsigned int
%o	八进制 unsigned int
%x	十六进制 unsigned int (%X 大写)
%f	float
%lf	double
%e	指数
%g	保证 *n*（默认 6）位有效数字的情况下使用小数方式，否则使用科学计数法
%c	一个字符 char
%s	字符串 char *

cin 与 cout 是 C++ 中的输入输出对象，性能相比 scanf 与 printf 非常差，推荐在输入输出时使用 scanf 与 printf，只有在必要时才使用 cin 和 cout。

例如，STL string 的读写只能用 cin 和 cout：

```
string str;
cin >> str;
cout << str;
getline(cin, str); // 读一整行使用 getline 函数
```

1.3 Python 语言输入输出技巧

Python 语言的输入技巧：

```
# 一个整数
n = int(input())

# 一个字符串
s = input()

# 两个整数，用空格分开
n, m = map(int, input().split())

# 一行整数，空格分开
l = list(map(int, input().split()))
l = [int(v) for v in input().split()]
s = set(map(int, input().split()))
s = {int(v) for v in input().split()}

# 一些字符串，空格分开
l = input().split()
```

Python 的格式化输出语法与 C 语言中的 printf 语法相似。

```
print("格式化字符串" % (<参量表>))
print("%d%02d" % (hour, minute))

# 在不方便指定参量类型时，推荐使用 format
print("格式化字符串".format(<参量表>))
print("{0:d}{1:02d}".format(hour, minute))
```

Python 输出时需要注意的一些问题：

```
# 按序输出 list 中的元素，空格分开结尾无空格
l = [1, 2, 3, 4, 5]
print(' '.join(map(str, l))) # 注意此处的 str
print(' '.join([str(v) for v in l]))

# 整数输出时注意末尾的.0
a = 5 / 5, b = 6 // 5
print(a, int(a), b) # 1.0 1 1
```

1.4 算术运算符

C++ 与 Python 的 +、−、∗、/ 及 % 运算符相同；对于 int 类型的除法，C++ 返回结果整数部分，Python 得到含小数部分的结果，若想与 C++ 一样只得到整数部分，需要使用“//”运算符。Python 中没有自增自减运算符“++”和“−−”。

1.5 常用数学函数

	C++ #include<cmath>	Python import math
$\lvert x\rvert$	abs(int x) fabs(double x)	abs(x) math.fabs(x)
$\lceil x\rceil$	ceil(double x)	math.ceil(x)
$\lfloor x\rfloor$	floor(double x)	math.floor(x)
r^p	pow(double r, p)	math.pow(r, p)
e^x	exp(double x)	math.exp(x)
$\sqrt{x}$	sqrt(double x)	math.sqrt(x)
$x!$	/	math.factorial(x)
$\log_e x$	log(double x)	math.log(x)
$\log_{10} x$	log10(double x)	math.log10(x)
$\log_a x$	/	math.log(x, a)
$\sin x$	sin(double x)	math.sin(x)
$\cos x$	cos(double x)	math.cos(x)
$\tan x$	tan(double x)	math.tan(x)
$\arcsin x$	asin(double x)	math.asin(x)
$\arccos x$	acos(double x)	math.acos(x)
$\arctan x$	atan(double x)	math.atan(x)
rad → deg	/	math.degrees(x)
deg → rad	/	math.radians(x)
round	round(double x)	round(x, n)
$\gcd(a,b)$	见??	math.gcd(a, b)
π	M_PI	math.pi
e	M_E	math.e
∞	#include<limits> INT_MAX UINT_MAX 等	math.inf

注意：三角函数中的 x 为弧度制。

1.6 Python 表达式求值

Python 的 eval 函数可以将 str 当成有效的表达式并求值返回结果，可以与 math 结合在表达式求值题目中发挥巨大的作用。例如：

```
expession = '2 + 3 * 5 - 6 // 4'
print(eval(expression)) # 16
```

也可以进行其他表达式计算，如 CSP 201709-3 JSON 查询。

```
t = eval('obj' + ''.join(['[' + repr(x) + ']'
    for x in c]))
```

repr 函数将对象转化为供解释器读取的形式。

1.7 浮点数比较

浮点数在计算机中的存储不总是精确地，在经过大量计算后，浮点数的存储会产生一定的误差，在使用“==”运算符进行比较时会出现差错。此时需要引入一个极小数 eps 来对误差进行修正。

```
const double eps = 1e-8;
fabs(a - b) < eps; // 相等
a - b > eps; // a > b
a - b < -eps; // a < b
```

2 数据结构

2.1 数组

2.1.1 普通数组

```
l = ['a', 'b', 3]
# 长 1000，初始值全为 0
l = [0] * 1000
# 100 * 100 大小初始值为 0 的二维数组
l = [[0] * 100 for _ in range(100)]
```

```
// 全部赋值 0
int a[10] = {0};
// 部分指定初始值，其余为默认值
int a[10] = {5, 3, 2, 4, 1};
// 二维数组初始化
int a[5][6] = {{3, 1, 2}, {8, 4}, {}};
// 字符数组可以使用字符串初始化，仅限初始化
char str[15] = "Hello, World!";
```

2.1.2 赋值函数 memset 与 fill

使用 memset 与 fill 函数可以对数组赋相同的值。memset 对每个字节赋同样的值，**只建议赋值 0 和 −1**，fill 函数可以给数组赋任意值，memset 执行速度快。

```
int a[10];
// memset(数组名, 值, sizeof(数组名));
memset(a, 0, sizeof(a));
// 将 [start, end) 之间的元素赋值为 value
// fill(iterator start, it end, const T& val);
// 将从 start 开始的 n 个元素赋值为 val
// fill_n(iterator start, Size n, const T& val);
fill(a, a + 10, 2);
fill(a, 10, 3);
```

2.1.3 变长数组

变长数组常用于解决普通数组超内存的情况，也可用于以邻接表方式储存图。

Python 提供了变长的连续存储数据结构 list，可以进行索引、切片、加、乘、检查成员等操作：

```
a = ['a', 'b', 3] # 初始化
# 下标访问
a[0], a[1], a[-1] # 'a', 'b', 3
# 切片，注意下标范围 [start, end)
a[1 : 2] # ['b']
a.append(x) # 将 x 插入到 a 尾部
t = a.pop(0) # 剔除 0 号元素并返回 0 号元素值
a.reverse() # 列表倒置
# 合并（不去重）
a = [1, 2, 3], b = [3, 4, 5]
a + b # [1, 2, 3, 3, 4, 5]
```

STL 中提供了变长数组 vector：

```
vector<TYPE> v; // 一维数组
vector<vector<TYPE> > vv; // 二维数组
vector<TYPE> v[size]; // 一维定长，另一维边长数组
v.push_back(x); // 将 x 插到 vector 尾部，O(1)
v.pop_back(); // 将 vector 尾部元素删除，O(1)
size_type t = v.size(); // vector 长度，O(1)
v.clear(); // 清空 vector，O(n)
v.insert(it, x); // 在迭代器 it 处插入 x，O(n)
v.erase(it); // 删除迭代器 it 处元素，O(1)
v.erase(begin, end); // 删除 [begin, end) 内元素
```

2.2 栈

栈是一种**后进先出**的数据结构，常用于模拟实现递归，防止程序对栈内存的限制导致程序出错。一般的递归层数达到几千或几万层就会崩溃，使用栈模拟递归算法可以避免此类问题出现。

Python LifoQueue 是一个后进先出的队列，可以当做栈来使用：

```
from queue import LifoQueue

# maxsize 设置最大长度，0 为无限长
q = LifoQueue(maxsize=0)
q.put(x) # 将 x 入队列
item = q.get() # 获取队首元素
is_empty = q.empty() # 判断队列是否为空
is_full = q.full() # 判断队列是否已满
s = q.qsize() # 返回队列大小（多线程不可靠）
```

STL stack 初始化及常用函数：

```
stack<TYPE> s;
s.push(const TYPE &val);  // 将 val 入栈，O(1)
s.pop(); // 弹出栈顶元素，O(1)
s.clear(); // 清除所有元素，O(n)
TYPE t = s.top(); // 获得栈顶元素，O(1)
bool e = s.empty(); // 检测栈是否为空，O(1)
size_type t = s.size(); // 栈内元素个数，O(1)
```

2.3 队列

队列是一种**先进先出**的数据结构。当需要实现广度优先搜索时，可以使用 queue 代替手动实现的队列，提高程序准确性。

Python Queue 是一个先进先出的队列，其常用函数与2.2 LifoQueue 相同。

Python deque 是双端队列，使用时注意插入删除操作的位置及相关函数。

```
from collections import deque

# deque 可从 iterable 指定的数据创建
# maxlen 指定最大长度
d = deque([iterable[, maxlen]]])
d.append(x)  # 添加 x 到右端
d.appendleft(x) # 添加 x 到左端
d.clear() # 移除所有元素
c = d.count(x) # 计算 deque 中 x 元素个数
d.extend(iterable) # 扩展 deque 的右侧
```

```
d.extendleft(iterable) # 扩展 deque 的左侧
item = d.pop() # 移除并返回最右侧元素
item = d.popleft() # 移除并返回最左侧元素
d.remove(x) # 找到第一个 x 并移除，未找到抛出异常
d.rotate(n = 1) # 向右循环 n 步，若 n 为负向左循环
```

STL queue 初始化及常用函数：

```
queue<TYPE> q;
q.push(const TYPE &val); // 将 val 入队列，O(1)
q.pop(); // 将队首元素出队，O(1)
bool e = q.empty(); // 检测队列是否为空，O(1)
size_type t = q.size(); // 队列元素个数，O(1)
TYPE &back(); // 返回队尾元素的引用，O(1)
TYPE &front(); // 返回队首元素的引用，O(1)
```

注意：在使用 front 和 pop 函数前，必须用 empty 判断队列是否为空。

2.4 优先队列

优先队列底层用堆来实现。在优先队列中，**队首元素**是队列中**优先级最高**的一个。优先队列可以解决一些贪心问题，也可用于对 Dijkstra 算法进行优化。

Python PriorityQueue 初始化及常用函数：

```
from queue import PriorityQueue

q = PriorityQueue(maxsize=0) # maxsize 最大长度
q.put((priority_number, data)) # 插入元素
data = q.get() # 获取队首元素
is_empty = q.empty() # 队列是否为空
# 当队列的元素是自定义时，需要在元素类中定义比较规则
```

STL priority_queue 初始化及常用函数：

```
priority_queue<TYPE> q;
// 只能通过 top 函数访问队首元素，使用前判断是否为空
TYPE t = q.top();
q.push(x); // 将 x 入队列
q.pop(); // 将队首元素出队列
bool t = q.empty(); // 判断队列是否为空
size_type s = q.size(); // 队列元素个数
```

定义优先队列内元素的优先级时，基本数据类型可以使用 less 或 greater 来指定优先级，其中 less 表示大的优先，greater 表示小的优先；结构体元素需要重载小于号运算符。

```
// 第二个参数指定存储容器
priority_queue<int, vector<int>, less<int>> q;
```

```
struct fruit {
    string name;
    int price;
    friend bool operator< (fruit f1, fruit f2) {
        return f1.price < f2.price;
    }
}
priority_queue<fruit> q; // 自定义元素类型
```

2.5 字符串

2.5.1 C string.h

string.h 中包含许多用于字符数组的函数：

```
int strlen(char* s); // 字符串长度
// 字符串比较，原则是字典序
// 相等 0；s1 > s2，正数；s1 < s2，负数
int strcmp(char* s1, char* s2);
void strcpy(char* s1, char* s2); // 将s2复制给s1
void strcat(char* s1, char* s2); // 将s2加到s1后
```

2.5.2 STL string

STL string 对字符串常用需求进行了封装：

```
string str = "abcd";
str += "ef"; // 将 "ef" 添加到 str 尾部
// == != < <= > >= 字符串比较，规则是字典序
int len = str.length(); // 字符串长度，O(1)
/* insert 函数，O(n) */
str.insert(pos, s); // 在 pos 位置插入字符串 s
// 在 it 插入串 [first, last)
str.insert(it, first, last);
/* erase 函数，O(n) */
str.erase(it); // 删除 it 指向元素
str.erase(first, last); // 删除区间[first, last)
str.erase(pos, len); // 从pos开始删除 len 个字符
str.clear(); // 清空字符串，O(1)
str.substr(pos, len); // 从 pos 开始长 len 的子串
string::npos // 常数，用于 find 函数匹配失败返回值
/* find 函数 */
int p = str.find(s); // s 在 str 的位置
int p = str.find(s, pos); // 从str的pos位匹配s
/* replace 函数 */
// 把 str 从 pos 开始，长 len 的子串替换为 s
string s2 = str.replace(pos, len, s);
// 把 str 的 [first, last) 范围的子串替换为 s
string s2 = str.replace(first, last, s);
char* s = str.c_str(); // 转换为字符数组
```

2.6 集合

Python 的集合 set 是一个**无序的、不含重复元素**的容器，不记录元素插入点或位置，不可通过下标访问，查询性能相比 list 高很多。

```
s = set()
s.add(x) # 将 x 插入集合中，若 x 存在，不进行操作
s.update(x) # 更新 s，加上 x 中的元素
s.remove(x) # 将 x 从集合中移除，若 x 不存在报错
s.discard(x) # 将 x 从集合中移除，允许 x 不存在
item = s.pop() # 随机删除一个元素
s.clear() # 清空集合
s.issubset(x) # s 是否为 x 的子集
x & y # 交
x | y # 并
x - y # 差
x ^ y # 对称差，即 x 和 y 的交集减并集
```

STL 集合 set 是一个**内部自动有序**且**不含重复元素**的容器，最重要的作用是自动去重并按升序排序。

```
set<TYPE> s;
set<TYPE>::iterator it; // 迭代器
// 遍历
for(auto it = s.begin(); it != s.end(); ++it)
do_something(*it);
s.insert(x); // 将 x 插入集合中，O(log n)
it = s.find(x); // 返回 x 的迭代器，O(log n)
// 删除迭代器 it 指向元素，O(1)，可与 find 结合
s.erase(it);
s.erase(x); // 删除值为 x 的元素，O(log n)
s.erase(begin, end); // 删除 [begin, end) 内元素
size_type t = s.size(); // set 内元素个数，O(1)
s.clear(); // 清空所有元素，O(n)
```

C++ 11 中增加了 unordered_set，以散列代替 set 内部红黑树，去重但不排序，速度相比 set 快很多。

2.7 映射

通过名字来引用值的数据结构称为映射，名字成为键 key，与值 value 成对。常用于：建立字符（串）与整数之间映射；判断大整数或者其他类型数据是否存在；建立字符串与字符串之间的映射。

Python dict 字典是键值对的无序可变集合，具有极快的查找速度。

```
d = dict() # 空字典或 d = {}
d = {key1: val1, key2: val2} # 指定初值
d = dict(key1=val1, key2=val2)
len(d) # 返回字典长度
val = d[key] # 下标访问
d[key] = value # 将 d[key] 设置为 value
val = d.get(key) # get 方法访问
val = d.pop(key) # 删除 key，返回对应 value
(key, val) = d.popitem() # 随机删除并返回一对 k-v
d.update((key, val)) # 用指定的(key, val)更新字典
d.items() # 返回可遍历的 (key, val) 元组数组
d.keys() # 返回字典所有键
key in d # 判断 key 是否字典中
d.values() # 返回字典所有值
del d[key] # 删除 key 对应的键值对，不存在抛出异常
d.clear() # 清空字典
```

STL map 可以将任何类型或容器映射到任何类型或容器。map 中**键值唯一，插入时按键值排序**。map 中存放的是 pair 结构体，有 first 和 second 两个不同类型的成员。

```
map<key_type, value_type> mp;
mp['a'] = 1; // 下标访问
// 迭代器访问
for (auto i = mp.begin(); i != mp.end(); ++i)
do_something(i->first, i->second);
it = mp.find(key); // 返回键为 key 的迭代器
mp.erase(it); // 删除 it 指向元素
mp.erase(key); // 删除键值为 key 的元素
mp.erase(first, last); // 删除[first, last)内元素
size_type s = mp.size(); // 映射个数
mp.clear(); // 清空所有元素
```

C++ 11 提供 unordered_map，以散列代替红黑树实现 map，只处理映射不排序，速度比 map 快得多。

3 基础算法

3.1 进制转换

对于一个 P 进制数 x，将其转换为 Q 进制数 z，需要分为两步：

(1) 将 x 转换为十进制数 y，若 P 进制数 $x = a_1a_2...a_n$，则 $y = a_1 * P^{n-1} + a_2 * P^{n-2} + ... + a_{n-1} * P + a_n$；

```
def p_to_ten(x: int, p: int) -> int:
    y, tmp = 0, 1
    while x != 0:
        y = y + (x % 10) * tmp
        x = x // 10
        tmp = tmp * p
    return y
```

```
int p_to_ten(int x, int p) {
    int y = 0, tmp = 1;
    while (x != 0) {
        y = y + (x % 10) * tmp;
        x /= 10;
        tmp *= p;
    }
    return y;
}
```

(2) 将 y 转换为 Q 进制数 z，采用“除基取余法”，每次将待转换数除以 Q，然后将得到的余数作为低位存储，而商则继续除以 Q 并进行上面的操作，最后当商为 0 时，将所有位从高到低输出就可以得到 z。

```
def ten_to_q(y: int, q: int):
    z = []
    while y != 0:
        z.append(y % q)
        y = y // q
    print(''.join(map(str, reversed(z)))) if len(
        z) else print(0)
```

```
void ten_to_q(int y, int q) {
    int z[40], num = 0;
    do {
        z[num++] = y % q;
        y /= q;
    } while (y != 0);
    for (int i = num - 1; i >= 0; --i)
        printf("%d", z[i]);
}
```

3.2 递归

3.2.1 分治

分治法将原问题划分成若干个**规模较小**而**结构与原问题相同**的子问题，然后分别解决，最后**合并子问题的解**，即可得到原问题的解。

3.2.2 递归

递归适合用来实现分治思想。递归函数中包含两个重要的概念：**递归边界**和**递归调用**。

Fibonacci 数列 可以使用递归式 $F(n) = F(n-1) + F(n-2), F(0) = F(1) = 1$ 编写程序，但会造成重复计算的问题；推荐写递推程序，不会产生爆栈问题。

```
long long fib(int n) {
    if (n <= 1)
        return 1;
    long long a = 1, b = 1;
    for (int i = 2; i <= n; ++i) {
        b = a + b;
        a = b - a;
    }
    return b;
}
```

全排列 对于全排列，可以使用深度优先搜索实现，也可使用 STL next_permutation 函数，效率更高。

```
// 函数原型
bool next_permutation(begin, end[, cmp]);
// 示例程序
int perm[6] = {1, 2, 3, 4, 5, 6}, count = 0;
do {
    ++count;
} while(next_permutation(perm, perm + 6));
```

也可使用递归程序实现

```
int p[maxn];
bool hashTable[maxn] = {false};
void generateP(int index, int n) {
    if (index == n + 1) {
        // 递归边界
        return;
    }
    for(int x = 1; x <= n; x++) {
        if (hashTable[x] == false) {
```

```
            p[index] = x;
            hashTable[x] = true;
            generateP(index + 1, n);
            hashTable[x] = false;
        }
    }
}
```

3.2.3 回溯

在到达递归边界的某一层，由于一些事实导致已经不需要任何一个子问题的递归，就可以直接返回上一层。这种做法称为**回溯法**。

n 皇后问题 在 $n*n$ 的国际象棋棋盘上放置 n 个皇后，两两不在同一行、同一列、同一条对角线上，求合法方案数。

可以使用全排列来实现。当放置部分皇后时，可能剩余的皇后无论怎样放置都不可能合法，此时就没必要递归了，直接返回上一层即可，这样可以减少很多计算量。

```
int n, count = 0, p[maxn];
bool hashTable[maxn] = {false};
void generateP(int index) {
    if (index == n + 1) {
        count++;
        return;
    }
    for (int x = 1; x <= n; x++) {
        if (hashTable[x] == false) {
            bool flag = true;
            for (int pre=1; pre <index; pre++) {
                if(abs(index-pre)==abs(x-p[pre])){
                    flag = false;
                    break; // 与之前皇后同一对角线
                }
            }
            if (flag) {
                p[index] = x;
                hashTable[x] = true;
                generateP(index + 1);
                hashTable[x] = false;
            }
        }
    }
}
```

3.3 排序

3.3.1 STL sort

STL 中包含一些进行排序操作的函数，函数的操作区间为 [begin, end)，使用 cmpfunc 指定二元比较函数，默认的比较函数是 operator<。这里的 begin、middle、end 都是迭代器 iterator。

```
// 使用随机化快速排序对区间内所有元素排序，不稳定
void sort(begin, end[, cmpfunc]);

// 若 m 是排序后的第 middle-begin 个元素
// 则将 m 放置在 middle 位置
// 排在 m 前后的两个区间都是无序的
void nth_element(begin, middle, end[, cmp]);

// 判断一个区间是否有序
bool is_sorted(begin, end[, cmp]);

// 符合一元判断函数test条件的移到前面,不符的移到后面
// 返回指向第一个不符合条件元素的迭代器 mid
// 符合的在[begin,end)之间,不符合的在[mid,end)之间
iterator partition(begin, end, func test);
// 稳定的partition
iterator stable_partition(begin, end,func test);
```

3.3.2 快速排序

快速排序是冒泡排序的一种改进，使用分治思想，时间复杂度为 $O(n\log n) \sim O(n^2)$，空间复杂度为 $O(\log n)$。

将第一个元素设为枢轴，从左边找一个大于枢轴的元素，从右边找一个小于枢轴的元素，进行交换；最终小于枢轴的元素在枢轴的左边，大于枢轴的元素在枢轴的右边；对分开的两部分再次使用快速排序进行排序。

示例程序：排序范围为 [start, end]

```
void quicksort(int a[], int start, int end) {
    if (start < end) { // 递归终止条件
        int i = start, j = end + 1;
        int x = a[start];
        while (true) {
            // 从左侧找到一个大于枢轴的元素
            while (a[++i] < x && i < end);
            // 从右侧找到一个小于枢轴的元素
            while (a[--j] > x && j > start);
            // 迭代器相遇退出循环
```

```
            if (i >= j)
                break;
            // 交换两个元素
            swap(a[i], a[j]);
        }
        // 将枢轴归位
        a[start] = a[j];
        a[j] = x;
        // 注：也可使用STL partition函数，注意范围
        // 对左半边排序
        quicksort(a, start, j - 1);
        // 对右半边排序
        quicksort(a, j + 1, end);
    }
}
```

3.4 散列

3.4.1 散列的定义与整数散列

散列（hash）即**将元素通过一个函数转换为整数，使得该整数可以尽量唯一地代表这个元素**。这个转换函数称为**散列函数** H。对于**整数** k，常用的方法有把 k 作为数组下标；也可使用**除留余数法**，即把 k 除以一个数 m 得到的余数作为 hash 值的方法。这样就可以把很大的数转换为不超过 m 的整数。

一般来说，可以使用 STL map/unordered_map 来直接使用 hash 的功能，因此除非必须模拟这些方法或是对算法效率要求高，一般不需要自己实现。

3.4.2 字符串散列

若 $P(x,y),0\le x,y\le m$ 表示二维平面上的一个点，将其映射为一个整数的散列函数可以写为 $H(P)=x*m+y$。

对于字符串，将其映射为一个整数，使该整数可以唯一地代表字符串。对于只有大写字母的字符串，可以将其当做 26 进制的数，然后将其转换为十进制，$H[i]=H[i-1]*26+\text{index}(\text{str}[i]))$。当字符串较长时，产生的整数会非常大。为了应对这种情况，可以将产生的结果对一个整数 m 取模，即 $H[i]=(H[i-1]*26+\text{index}(\text{str}[i]))\%m$，此方法可能会导致冲突。

在 int 数据范围内，如果把进制数设置为一个 10^7 级别的素数 p（如 10^7+19），同时把 m 设置为一个 10^9 级别的素数（如 10^9+7），产生冲突的概率非常小 $H[i]=(H[i-1]*p+\text{index}(\text{str}[i]))\%m$。

例如：给出 n 个只有小写字母的字符串，求其中不同的字符串个数。

对于这个问题，如果只用字符串 hash 来做，只需要将 n 个字符串的 hash 求出来，去重排序即可，当然也可以用 set 或者 map 直接做，速度比字符串 hash 慢一些。

```
const int MOD = 1e9 + 7;
const int P = 1e7 + 19;
vector<int> ans;

long long hash(string str) {
    long long h = 0;
    for (int i = 0; i < str.length(); ++i)
        h = (h * P + str[i] - 'a') % MOD;
    return h;
}

int main() {
    long long id;
    string str;
    while(getline(cin, str), str != '#') {
        id = hash(str);
        ans.push_back(id);
    }
    sort(ans.begin(), ans.end()); // 排序
    int count = 0;
    for (int i = 0; i < ans.size(); ++i)
        if (i == 0 || ans[i] != ans[i - 1])
            count++; // 统计不同的数的个数
    printf("%d", count);
    return 0;
}
```

3.5 贪心

3.5.1 简单贪心

贪心算法：当前最好选择。一般来说，**如果在想到某个似乎可行的策略，并且自己无法举出反例，那么就勇敢去实现它**。

例题：给定数字 0-9 若干个，可以任意排列这些数字，全部使用。使得到的数字最小，0 不能做首位。

```
int count[10] = {0};
string str;
getline(cin, str);
```

```
for (auto i = str.begin(); i < str.end(); ++i)
    count[(*i) - '0']++; // 统计数字个数
for (int i = 1; i < 10; ++i) // 输出第一个数字
    if (count[i] > 0) {
        printf("%d", i);
        count[i]--;
        break;
    }
for (int i = 0; i < 10; ++i) // 依次输出剩余数字
    for (int j = 0; j < count[i]; ++j)
        printf("%d", i);
```

3.5.2 区间贪心

给出一个**区间不相交问题：给出 n 个开区间 (x,y)，从中选择尽可能多的开区间，使得这些开区间两两没有交集**。例如对于开区间 $(1,3),(2,4),(3,5),(6,7)$ 来说，可以选出最多三个区间 $(1,3),(3,5),(6,7)$，它们互相没有交集。

首先考虑最简单的情况，若开区间 I_1 被开区间 I_2 包含，显然选择 I_1 是最好，如果选择 I_1，则有更大的空间去容纳其他的开区间。

接下来把所有区间按左端点 x 从大到小排序，如果去掉区间包含的情况，那么一定有 $y_1 > y_2 > ... > y_n$ 成立。此时最右边有一段是一定不会和其他区间重叠的，此时应该**总是先选择左端点最大的区间**。

```
struct Interval {
    int x, y;
} I[1000];

bool cmp(Interval a, Interval b) {
    if (a.x != b.x) // 先按左端点从大到小排序
        return a.x > b.x;
    else // 左端点相同的按右端点从小到大排序
        return a.y < b.y;
}

int n, i;
scanf("%d", &n);
for (i = 0; i < n; ++i)
    scanf("%d%d", &I[i].x, &I[i].y);
sort(I, I + n, cmp); // 把区间排序
// ans 记录不相交个数，tx记录上一个被选中区间左端点
int ans = 1, tx = I[0].x;
for (i = 1; i < n; ++i)
    if (I[i].y <= tx) { // 若该区间右端点在 tx 左边
        tx = I[i].x; // 以 I[i] 作为新选中区间
        ans++; // 不相交区间个数 +1
    }
printf("%d", ans); // 输出结果
```

3.6 二分法

3.6.1 二分查找

对于**有序容器**，可以使用 STL 中的 binary_search 函数进行二分法查找。

```
// 函数原型，查找范围 [first, last)
bool binary_search(it first, it last, T &val);
// 例如
int a[] = {1, 2, 3, 4, 5, 6};
bool have5 = binary_search(a, a + 6, 5);
```

3.6.2 快速幂

给定三个整数 $a,b,m(a < 10^9, b < 10^6, 1 < m < 10^9)$，求 $a^b\%m$。在快速幂的算法中，若 b 是奇数，$a^b = a * a^{b-1}$；若 b 是偶数，$a^b = a^{b/2} * a^{b/2}$。

针对不同的题目，需要注意的是：(1) 若初始时 $a \geq m$，需要在进入函数前让 a 对 m 取模；(2) 若 $m = 1$，可以直接在函数外部判为 0。

快速幂的递归写法不做讨论。在迭代写法中，如果把 b 写成二进制，则 b 就可以写成若干二次幂的和，即 $a^b = a^{2^k+...+4+2+1} = a^{2^k} * ... * a^4 * a^2 * a$，若 b 的二进制的 i 号位为 1，那么项 a^{2^i} 就被选中。

```
typedef long long ll;
ll bin_pow(ll a, ll b, ll m) {
    if (m == 1)
        return 0;
    if (a >= m)
        a %= m;
    ll ans = 1;
    while (b > 0) {
        if (b % 2 == 1)  // 若 b 的二进制末尾为 1
            ans = ans * a % m; // 令 ans 累积上 a
        a = a * a % m; // 令 a 平方
        b = b / 2; // 将 b 的二进制右移一位
    }
    return ans;
}
```

3.6.3 lower_bound, upper_bound

在**有序容器**中，lower_bound(first, last, val) 函数用来寻找在容器 [first, last) 范围内**第一个值大于或等于 val** 的元素位置；upper_bound(first, last, val) 函数用来寻找在容器 [first, last) 范围内**第一个值大于 val** 的元素位置。两个函数的复杂度均为 $O(\log(\text{last} - \text{first}))$。

4 数学问题

4.1 基本数学运算

	C++ STL
求和	accumulate(start, end, init)
最大值	max(x, y) max_element(first, last[, cmp])
最小值	min(x, y) min_element(first, last[, cmp])

4.2 最大公约数与最小公倍数

欧几里得算法（辗转相除法） 求解最大公约数常用欧几里得算法，即若 a,b 均为整数 $\gcd(a,b) = \gcd(b,a\%b)$。

```
int gcd(int a, int b) {
    if (b == 0)
        return a;
    else
        return gcd(b, a % b);
}
```

最小公倍数 $\text{lcm}(a,b) = \dfrac{ab}{\gcd(a,b)}$，在实际计算中 ab 有可能溢出，因此更恰当的写法是 $\text{lcm}(a,b) = \dfrac{a}{\gcd(a,b)} * b$。

4.3 素数

4.3.1 素数的判断

判断整数 n 是否为素数只需要判定 n 能否被 $2,3,...,\lfloor\sqrt{n}\rfloor$ 中的一个整除即可，复杂度为 $O(\sqrt{n})$。

```
bool isPrime(int n) {
    if (n <= 1)
        return false;
    int sqr = (int)sqrt(n);
    for (int i = 2; i <= sqr; ++i)
        if (n % i == 0)
            return false;
    return true;
}
```

4.3.2 素数表的获取

使用 Eratosthenes 筛选法获取范围为 $[1,n]$ 的素数表，首先需要枚举 $[2,n]$ 的所有数，对每一个素数，筛去它的所有倍数，剩下的都是素数。

```
vector<long> prime_array(long n) {
   int i, j;
   vector<bool> p(n + 1, false);
   vector<long> ret;
   for (i = 2; i <= n; ++i)
      if (p[i] == false) {
         ret.push_back(i);
         for (j = i + i; j < n; j += i)
            p[j] = true;
      }
   return ret;
}
```

4.4 质因子分解

4.4.1 质因子分解

质因子分解是指将正整数 n 写成一个或多个质数的乘积的形式，如 $180 = 2^2 \times 3^2 \times 5$。显然，质因子分解最后归结到若干不同质数的乘积，可以先把素数表打印出来。

注意：如果题目要求对 1 进行处理，视题目条件进行判定处理。

```
map<int, int> ans; // 质因子映射
vector<long> prime = prime_array(n); // 素数表
if (n == 1) do_something(); // n = 1 情况
else {
   int sqr = (int)sqrt(n), i;
   for (i = 0; i < n && prime[i] <= sqr; ++i) {
      // 若 prime[i] 是 n 的因子
      if (n % prime[i] == 0) {
         ans[prime[i]] = 0; // 计算质因子个数
         while (n % prime[i] == 0) {
            ans[prime[i]]++;
            n /= prime[i];
         }
      }
      if (n == 1) // 及时退出
         break;
   }
   if (n != 1) // 若无法被根号 n 以内的质因子除尽
      ans[n] = 1;
}
```

4.4.2 $n!$ 中质因子 p 个数

例如：$6! = 1 \times 2 \times 3 \times 4 \times 5 \times 6$，于是 6! 中有 4 个质因子 2，2 个质因子 3。$n!$ 中有 $\left(\left\lfloor \frac{n}{p} \right\rfloor + \left\lfloor \frac{n}{p^2} \right\rfloor + ...\right)$ 个质因子 p，时间复杂度为 $O(\log n)$。

```
int calc(int n, int p) {
   int ans = 0;
   while(n) {
      ans += n / p;
      n /= p;
   }
   return ans;
}
```

利用这个算法可以计算出 $n!$ **的末尾有多少个 0**。由于末尾 0 的个数等于 $n!$ 中因子 10 的个数，也等于 $n!$ 中质因子 5 的个数，因此只需计算 calc(n, 5) 就可得到结果。

将此例推广到一般情况可得，$n!$ 中质因子 p 的个数等于 $[1,n]$ 中 p 的倍数的个数 $\frac{n}{p}$ 加上 $\frac{n}{p}!$ 中质因子 p 的个数，因此可到递归版本的 calc 函数。

```
int calc(int n, int p) {
   if (n < p)
      return 0;
   return n / p + calc(n / p, p);
}
```

4.5 大整数运算

4.5.1 高精度计算

浮点数高精度计算时使用 double 类型存储待计算数据进行计算，整数运算需要使用大整数。

4.5.2 大整数的存储、比较与输出

使用整数数组 int d[1000] 存储，**整数的高位存储在数组的高位，低位存储在数组的低位**。

```
struct bign {
   int d[1000];
   int len;
   bign() {
      memset(d, 0, sizeof(d));
      len = 0;
   }
};
```

把整数按字符串读入时，需要在读入之后存储到 d[] 时反转一下。

```
bign change(char str[]) {
   bign a;
   a.len = strlen(str);
   for (int i = 0; i < a.len; ++i)
      a.d[i] = str[a.len - i - 1] - '0';
   return a;
}
```

比较两个大整数大小的规则简单：先判断两者 len 的大小，若不相等，则以长的为大；如果相等，则从高位到低位进行比较，直到出现某一位不等，就可以判断两个数的大小。

```
// a = b, 0; a > b, 1; a < b, -1
int compare(bign a, bign b) {
   if (a.len > b.len)  return 1;
   else if (a.len < b.len)  return -1;
   else {
      for (int i = a.len - 1; i >= 0; --i) {
         if (a.d[i] > b.d[i]) return 1;
         else if (a.d[i] < b.d[i]) return -1;
      }
      return 0;
   }
}
```

大整数输出时从最高位到最低位依次输出即可。

```
void print(bign a) {
   for (int i = len - 1; i >= 0; --i)
      printf("%d", a.d[i]);
}
```

4.5.3 大整数的四则运算

高精度加法 对于其中的某一位，将该位上的两个数字和进位相加，得到的结果取个位数作为该位结果，取十位数作为新的进位。

```
bign add(bign a, bign b) {
   bign c;
   int carry = 0, i, t; // carry 为进位
   // 以较长的为边界
   for (i = 0; i < a.len || i < b.len; ++i) {
      t = a.d[i] + b.d[i] + carry;
      c.d[c.len++] = t % 10; // 该位结果
      carry = t / 10; // 新的进位
   }
   if (carry != 0) // 最后进位不为 0
      c.d[c.len++] = carry;
   return c;
}
```

高精度减法 对于其中的某一位，比较被减位和减位，如果不够减，向高位借 1，如果够减，直接减。注意减法后**高位多余的 0 要忽视，但保证结果至少一位数**。

```
bign sub(bign a, bign b) {
   bign c; int i;
   // 以较长的为边界
   for (i = 0; i < a.len || i < b.len; ++i) {
      if (a.d[i] < b.d[i]) { // 如果不够减
         a.d[i + 1]--;
         a.d[i] += 10;
      }
      c.d[c.len++] = a.d[i] - b.d[i];
   }
   // 去除高位的 0，同时至少保留一位最低位
   while (c.len-1 >= 1 && c.d[c.len - 1] == 0)
      --c.len;
   return c;
}
```

高精度与低精度乘法 低精度指的是基本数据类型，如 int 等。对于某一位来说，取 bign 的某一位与 int 型整体相乘，再与进位相加。

```
bign multiply(bign a, int b) {
   bign c;
   int carry = 0, i, t; // carry 进位
   for (i = 0; i < a.len; ++i) {
      t = a.d[i] * b + carry;
      c.d[c.len++] = t % 10;
      carry = t / 10;
   }
   // 乘法的进位可能不止一位
   while (carry != 0) {
      c.d[c.len++] = carry % 10;
      carry /= 10;
   }
   return c;
}
```

高精度与低精度除法 除法运算与小学所学相同。对于某一步操作：上一步的余数乘以 10 加上该步的位，

得到该步临时的被除数，将其与除数比较。若不够除，则该位的商为 0；若够除，则商为对应的商，余数为对应的余数。最后一步要减去高位多余的 0，并保证结果至少一位。

```
bign divide(bign a, int b, int &r) {
    bign c;
    // 被除数的每一位和商的每一位对应，先令长度相等
    c.len = a.len;
    for (int i = a.len - 1; i >= 0; --i) {
        // 和上一位遗留的余数组合
        r = r * 10 + a.d[i];
        if (r < b) // 不够除，该位为 0
            c.d[i] = 0;
        else { // 够除
            c.d[i] = r / b;
            r = r % b;
        }
    }
    // 去除高位的 0，同时至少保留一位最低位
    while (c.len-1 >= 1 && c.d[c.len - 1] == 0)
        --c.len;
    return c;
}
```

4.6 组合数计算

组合数计算主要讨论 C_n^m 和 $C_n^m \% p$ 两个问题。

对于 C_n^m，可以使用定义式 $C_n^m = \dfrac{n!}{m!(n-m)!}$ 直接计算，在 $n = 21$ 时超出 long long 范围；也可使用递推公式 $C_n^m = C_{n-1}^m + C_{n-1}^{m-1}, C_n^0 = C_n^n = 1$ 编写递归程序，注意重复计算问题，在 $n = 67, m = 33$ 时溢出；推荐使用定义式变形 $C_n^m = \prod_{i=1}^{m} \dfrac{n-m+i}{i}$ 来计算，复杂度为 $O(m)$，在 $n = 62, m = 31$ 时溢出：

```
long long C(int n, int m) {
    long long ans = 1;
    for (int i = 1; i <= m; ++i)
        // 注意一定要先乘后除
        ans = ans * (n - m + i) / i;
    return ans;
}
```

一般来说，常见的情况是求 $C_n^m \% p$，有四种方法，有各自的数据适合范围，一般第一种方法已经满足需求。

方法一，$m \le n \le 1000$，p 无限制，求解方法是求 C_n^m 的递推式变形

```
int res[1010][1010] = {0};
int C(int n, int m, int p) {
    if (m == 0 || m == n)
        return 1;
    if (res[n][m] != 0)
        return res[n][m];
    res[n][m] = (C(n-1, m) + C(n-1, m-1)) % p;
    return res[n][m];
}
```

方法二，Lucas 定理，$m \le n \le 10^{18}$，p 是素数

```
int p; // 全局 p
int Lucas(int n, int m) {
    if (m == 0)
        return 1;
    return C(n % p, m % p)*Lucas(n/p, m/p) % p;
}
```

5 搜索

5.1 深度优先搜索 DFS

深度优先搜索的基本模型：

```
void dfs(int step) {
    if (step == n + 1) // 判断边界
        return;
    // 尝试每一种可能
    for (int i = 1; i <= n; i++) {
        if (符合条件) {
            dfs(step + 1);
        }
    }
    return; // 返回
}
```

迷宫问题，迷宫由 n 行 m 列的单元格组成，每个单元格要么为空，要么有障碍，求最短路径长度。

```
int n, m, p, q, ans = 1e9;
int a[51][51], book[51][51];
// 右下左上
int next[4][2] = {{0,1}, {1,0}, {0,-1}, {-1,0}};

void dfs(int x, int y, int step) {
    int tx, ty, k;
    if (x == p && y == q) {
        // 到达终点，更新最小值
        if (step < ans) ans = step;
        return;
    }
    // 枚举四种走法
    for (k = 0; k < 4; k++) {
        tx = x + next[k][0]; // 计算下一点坐标
        ty = y + next[k][1];
        // 判断是否越界
        if (tx < 1 || tx > n || ty < 1 || ty > m)
            continue;
        // 判断是否为障碍物或是否已在路径中
        if (a[tx][ty] == 0 && book[tx][ty] == 0){
            book[tx][ty] = 1; // 标记此点已经过
            dfs(tx, ty, step + 1); // 尝试下一点
            book[tx][ty] = 0; // 尝试结束，取消标记
        }
    }
    return;
}
// 从 (1, 1) 开始
dfs(1, 1, 0);
```

5.2 广度优先搜索 BFS

BFS 以广度为第一关键词，当碰到岔道口时，总是先**依次访问**从该岔道口能**直接到达的所有结点**，然后再按这些结点被访问的顺序去依次访问，直到所有结点被访问为止。广度优先搜索的基本模型：

```
void bfs(int s) {
    queue<int> q;
    q.push(s);
    while (!q.empty()) {
        取出队首元素 top;
        访问 top;
        将队首元素出队;
        将 top 的下一层结点中未曾入队的结点入队，并设
            置已入队;
    }
}
```

再以迷宫问题为例：

```
int n, m, p, q, ans = 1e9;
int a[51][51], book[51][51];
int next[4][2] = {{0,1}, {1,0}, {0,-1}, {-1,0}};

struct node {
    int x, y, father, step;
};
// 判断坐标 (x, y) 是否需要访问
bool judge(int x, int y) {
    if (x > n || x < 1 || y > m || y < 1) // 越界
        return false;
    // 当前位置不可访问或 (x, y) 已入队
    if (a[x][y] == 0 || book[x][y])
        return false;
    return true;
}
// bfs 函数
int bfs(int x, int y) {
    node t;
    queue<node> q;
    t.x = x, t.y = y;
    q.push(t);
    book[x][y] = 1;
    while (!q.empty()) {
        node top = q.front();
        q.pop();
        if (top.x == p && top.y == q)
            return top.s; // 到达终点，返回步数
        for (int i = 0; i < 4; i++) {
            int tx = top.x + next[i][0];
```

```
            int ty = top.y + next[i][1];
            // 若新位置需要访问
            if (judge(tx, ty)) {
                t.x = tx, t.y = ty;
                // 新结点层数 +1
                t.step = top.step + 1;
                q.push(t); // 新结点加入队列
                book[tx][ty] = 1; // 令新位置已访问
            }
        }
    }
    return -1; // 无法到达终点
}
// 从 (1, 1) 开始
cout << bfs(1, 1);
```

6 树

6.1 二叉树存储与基本操作

6.1.1 存储

```
struct node {
    int data;
    node* lchild;
    node* rchild;
};

node* root = NULL;

node* newNode(int v) {
    node* n = new node;
    n->data = v;
    n->lchild = n->rchild = NULL;
    return n;
}
```

6.1.2 查找、修改

```
void search(node* root, int x, int newdata) {
    if (root == NULL)
        return ;
    if (root->data == x)
        root->data = newdata;
    search(root->lchild, x, newdata);
    search(root->rchild, x, newdata);
}
```

6.1.3 插入结点

```
void insert(node* &root, int x) {
    if (root == NULL) {
        root = newNode(x);
        return ;
    }
    if ("可以在左子树插入")
        insert(root->lchild, x);
    else
        insert(root->rchild, x);
}
```

6.1.4 由数组创建

```
node* Create(int data[], int n) {
   node* root = NULL;
   for (int i = 0; i < n; i++)
      insert(root, data[i]);
   return root;
}
```

6.1.5 完全二叉树存储

对于完全二叉树的任一结点（设编号 x），**其左子结点编号一定是 $2x$，而右子结点的编号一定是 $2x+1$，且 1 号位存放的必须是根结点。**

该数组中元素存放的顺序恰好为该完全二叉树的层序遍历序列。

判断某个结点是否为叶结点的标志为：**该结点（下标为 x）的左子结点的编号 $2x$ 大于结点总数 n。**

判断某个结点是否为空的标志为：**该结点下标 x 大于结点总数 n。**

6.2 二叉树的遍历

6.2.1 先序、中序、后序遍历

先序遍历性质：**序列的第一个一定是根结点。**

中序遍历性质：**只要知道根结点，就可以通过根结点在中序遍历序列中的位置区分出左子树和右子树。**

后序遍历性质：**后序遍历序列最后一个一定是根结点。**

无论是先序遍历还是后序遍历序列，都必须知道中序遍历序列才能唯一地确定一棵树。

6.2.2 层序遍历

对二叉树进行层序遍历就相当于是对二叉树从根结点开始的广度优先搜索。其基本思路如下：

(1) 将根结点 root 入队列 q；

(2) 取出队首结点，访问它；

(3) 如果该结点有左孩子，将左孩子入队列

(4) 如果该结点有右孩子，将右孩子入队列

(5) 返回 (2)，直到队列为空

```
// 若要记录结点层次，添加 layer 变量
struct node {
   int data;
   int layer;
   node* lchild, rchild;
};

void LayerOrder(node* root) {
   queue<node*> q;
   root->layer = 1; // 根结点层号为 1
   q.push(root);
   while (!q.empty()) {
      node* top = q.front();
      q.pop();
      do_something(top); // 对队首元素的操作
      if (top->lchild != NULL) {
         // 子结点层次号 + 1
         top->lchild->layer = top->layer + 1;
         q.push(top->lchild);
      }
      if (top->rchild != NULL) {
         top->rchild->layer = top->layer + 1;
         q.push(top->rchild);
      }
   }
}
```

6.2.3 根据遍历序列重建二叉树

给定一棵二叉树的先序序列和中序序列，重建这棵二叉树。假设已知先序序列 $pre[1..n]$，中序序列 $in[1..n]$。由先序序列性质可知，先序序列的第一个元素 pre_1 是当前二叉树的根结点；再由中序序列的性质可知，当前二叉树的根结点将中序序列划分为左子树和右子树。因此要在中序序列中找到一个结点 in_k，使得 $in_k = pre_1$，这样就在中序序列中确定根结点。易知左子树结点个数 $numLeft = k-1$，左子树序列区间为 $pre[2..k], in[1..k-1]$，右子树的序列区间为 $pre[k+1..n], in[k+1..n]$，接着只需要网左子树和右子树进行递归构建二叉树即可。

```
// 当前先序序列区间为 [preL, preR]
// 中序序列区间为 [inL, inR]，返回根结点地址
node* create(int preL, int preR, int inL, int
    inR) {
   if (preL > preR) // 先序序列长度 <= 0，返回
      return NULL;
   // 新建结点用于存放当前二叉树的根结点
   node* root = new node;
   root->data = pre[preL];
   int k;
   for (k = inL; k <= inR; k++)
```

```
        // 在中序序列中找到 in[k] == pre[L] 的结点
        if (in[k] == pre[preL])
            break;
    int numLeft = k - inL; // 左子树结点个数

    root->lchild = create(preL + 1, preL +
        numLeft, inL, k - 1);
    root->rchild = create(preL + numLeft + 1,
        preR, k + 1, inR);

    return root;
}
```

中序序列可以与先序、后序、层序序列中的任意一个来构建唯一的二叉树，后三者两两搭配或者三个一起都无法构建唯一二叉树。

6.3 并查集 UFS

6.3.1 定义

并查集是一种维护集合的数据结构，支持合并（合并两个集合）、查找（判断两个元素是否在一个集合）两种操作。

并查集使用数组 $father[N]$ 实现，其中 $father[i]$ 表示元素 i 的父亲节点，而父亲结点本身也是这个集合内的元素（$1 \le i \le N$）。例如 $father[1]=2$ 表示元素 1 的父亲结点是元素 2，以这种父亲关系来表示元素所属的集合。若 $father[i]=i$ 则说明元素 i 是该集合的根结点，但**对同一个集合来说只存在一个根结点，且将其作为所属集合的标识**。

6.3.2 基本操作

初始化 一开始，每个元素都是独立的一个集合，因此需要令所有的 $father[i]=i$，也可令所有 $father[i]=-1$。

```
for (int i = 1; i <= N; i++) father[i] = i;
```

查找 查找操作就是对给定的结点寻找其根结点的过程，即反复寻找父亲结点，直到找到根结点。

```
int find(int x) {
    while (x != father[x])
        x = father[x];
    return x;
}
```

合并 题目中一般给出两个元素，要求把这两个元素所在的集合合并。具体实现上一般是先判断两个元素是否属于同一个集合，只有当两个元素属于不同集合时才合并，合并的过程一般是把其中一个集合的根结点的父亲指向另一个集合的根结点。

```
void Union(int a, int b) {
    int fa = findFather(a);
    int fb = findFather(b);
    if (fa != fb)
        father[fa] = fb;
}
```

6.3.3 路径压缩

若题目给出的元素数量很多并且形成一条链，那么这个查找函数的效率非常低。压缩时**把当前查询结点的路径上的所有结点的父亲都指向根结点**，查找的时候就不需要一直回溯去找父亲了，复杂度可以降为 $O(1)$。

转换的过程可以概括为如下两个步骤：(1) 按原先的写法获得 x 的根结点 r；(2) 重新从 x 开始走一遍寻找根结点的过程，把路径上经过的所有结点的父亲全部改为结点 r。

```
int findFather(int x) {
    int t = x; // 保存原来的 x
    while (x != father[x]) // 寻找根结点
        x = father[x];
    // 将路径上所有结点 father 改为根结点
    while (t != father[t]) {
        int tt = a;
        a = father[a];
        father[tt] = x;
    }
    return x;
}
```

6.4 堆

堆是一棵**完全二叉树**，树中每个结点的值都不小于（或不大于）其左右孩子结点的值。若父亲结点的值大于或等于孩子结点的值，那么称这样的堆为**大顶堆**，这时**每个结点的值都是以它为根结点的子树的最大值**；反之，**小顶堆每个结点的值都是以它为根结点的子树的最小值**。

STL 中有在可以随机访问的容器上进行堆操作：

```
// 判断是否为堆
bool is_heap(begin, end[, fun cmp]);
// 在 [begin, end) 的区间上建立堆
void make_heap(begin, end[, fun cmp]);
// 插入元素到堆
// 前提 [begin, end) 是堆，将 end 指向元素插入堆中
// 使用前需要把元素插入到 end 位置，如 push_back
// 然后调用 push_heap
void push_heap(begin, end[, fun cmp]);
// 从堆中移除
// 前提 [begin, end) 是堆，将 begin 指向元素移除
// 被移除的元素实际上放置在 end - 1 位置，不会释放
void pop_heap(begin, end[, fun cmp]);
// 堆排序，前提：[begin, end) 是堆
void sort_heap(begin, end[, fun cmp]);
```

7 图

7.1 定义

图由顶点和边组成，每条边的两端都必须是图的两个顶点，记号 $G(V,E)$ 表示图 G 的顶点集为 V、边集为 E。

一般来说，图可分为**有向图**和**无向图**，有向图的所有边都有方向；而无向图的所有边都是双向的。

顶点的**度**是指和该顶点相连的边的条数，顶点的出边条数称为该点的**出度**，顶点的入边条数称为该顶点的**入度**。

顶点和边都可以有一定属性，而量化的属性称为**权值**。

7.2 图的存储

7.2.1 邻接矩阵

设图 $G(V,E)$ 的顶点标号为 $0,1,...,n-1$，那么可以令二维数组 $G[n][n]$ 的二维分别表示图的顶点标号，若 $G[i][j]=1$，则说明顶点 i 和顶点 j 之间有边；若 $G[i][j]=0$，则说明顶点 i 和顶点 j 之间不存在边，这个二维数组 G 被称为**邻接矩阵**。另外，如果存在边权，则可以令 $G[i][j]$ 存放边权，对不存在的边的边权设置为 0、-1 或是一个很大的数。

7.2.2 邻接表

把同一个顶点的所有出边放在一个列表中，那么 n 个顶点就有 n 个列表，没有出边对应空表，这 n 个列表被称为图 G 的**邻接表**，记为 $adj[n]$，其中 $adj[i]$ 存放顶点 i 的所有出边组成的列表。

7.3 图的遍历

7.3.1 采用 DFS 遍历

采用 DFS 遍历即沿着一条路径遍历直到无法继续前进，才退回到路径上离当前顶点最近的还未访问分值顶点的岔道口。

7.3.2 采用 BFS 遍历

7.4 最短路径

对任意给出的图 $G(V,E)$ 和起点 S、终点 T，求一条从 S 到 T 的路径，使得这条路径上经过的所有边的边权之和最小。

解决最短路径问题常用的算法有 Dijkstra 算法、Bellman-Ford 算法、SPFA 算法和 Floyd 算法。不同的算法应用于不同的题目。

7.4.1 Dijkstra 算法

Dijkstra 算法用于解决**单源最短路问题**，即给定图 G 和起点 s，通过算法得到 s 到其他每个顶点的最短距离。

基本思想是对图 $G(V,E)$ 设置集合 S，存放已被访问的顶点，然后每次从集合 $V-S$ 中选择与起点 s 的最短距离最小的顶点 u，访问并加入集合 S。之后，令顶点 u 为中介点，优化起点 s 与所有从 u 能到达的顶点 v 之间的最短距离。这样的操作执行 n 次，直到集合 S 已包含所有顶点。

集合 S 可以用一个 bool 数组 $vis[]$ 来实现，当 $vis[i]$ =true 时表示顶点 V_i 已被访问。令 int 型数组 $d[]$ 表示起点 s 到顶点 V_i 的最短距离，初始时除了起点 s 的 $d[s]=0$，其余顶点都赋值为一个很大的数（10^9 或者 0x3fffffff，但不要使用 0x7fffffff）来表示不可达。

```
const int MAXV = 1000; // 最大顶点数
const int INF = 1e9; // 最大数
int n, G[MAXV][MAXV]; // n 为顶点数
int d[MAXV]; // 起点到达各点的最短路径长度
// pre[v] 表示从起点到 v 的最短路径上 v 的前个顶点
int pre[MAXV];
bool vis[MAXV] = {false}; // 标记数组true为已访问

void Dijkstra(int s) { // s 为起点
   fill(d, d + MAXV, INF);
   d[s] = 0; // 数组 d 初始化
   for (int i = 0; i < n; i++) { // 循环 n 次
      // u 使 d[u] 最小，MIN 存放该最小的 d[u]
      int u = -1, MIN = INF;
      for (int j = 0; j < n; j++) {
         // 找到未访问的顶点中 d[] 最小的
         if (!vis[j] && d[j] < MIN) {
            u = j;
            MIN = d[j];
         }
      }
      // 找不到小于 INF 的 d[u]
      // 说明剩下的顶点和起点 s 不连通
      if (u == -1) return;
      vis[u] = true; // 标记 u 为已访问
      for (int v = 0; v < n; v++) {
         // 如果 v 未访问 && u 能到达 v
         // 以 u 为中介点可以使 d[v] 更优
         if (!vis[v] && G[u][v] != INF && d[u]
             + G[u][v] < d[v]) {
            d[v] = d[u] + G[u][v]; // 优化 d[v]
            pre[v] = u; // 记录 v 的前驱顶点
         }
      }
   }
}
```

Dijkstra 算法只能应对所有边权都是非负数的情况，若边权出现负数，那么 Dijkstra 算法可能会出错，这时最好使用 SPFA 算法。

若题目给出的是无向边，只需把无向边当成两条指向相反的有向边即可。

7.4.2 Bellman-Ford 算法和 SPFA 算法

Bellman-Ford 算法可以解决单源最短路径问题，也能处理有负权边的情况。

7.4.3 Floyd 算法

Floyd 算法用于解决**全源最短路问题**，即给定图 $G(V,E)$，求任意两点 u、v 之间的最短路径长度，复杂度为 $O(n^3)$，限制了顶点数在 200 以内，使用邻接矩阵来实现 Floyd 算法是非常合适方便的。

Floyd 算法流程：

```
for (k = 1; k <= n; k++)
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         if (dis[i][k] != INF &&
            dis[k][j] != INF &&
            dis[i][k] + dis[k][j] < dis[i][j])
            dis[i][j] = dis[i][k] + dis[k][j];
```

7.5 最小生成树

7.5.1 prim 算法

7.5.2 kruskal 算法

8 动态规划

8.1 动态规划的定义及写法

8.1.1 动态规划定义

动态规划（DP）是一种用来解决一类**最优化问题**的算法思想。简单来说，动态规划将一个复杂的问题分解成若干个子问题，通过**综合子问题的最优解来得到原问题的最优解**。动态规划会**将每个求解过的子问题解记录下来**，不会重复计算。

8.1.2 递归写法

以 Fibonacci 数列为例。根据斐波那契数列的定义 $F_n = F_{n-1} + F_{n-2}$ 可以写出递归程序。

```
int F(int n) {
   if (n == 0 || n == 1) return 1;
   else return F(n - 1) + F(n - 2);
}
```

这样会造成很多重复计算。为避免重复计算，可以开一个一维数组 dp 保存已经计算过的结果，其中 $\mathrm{dp}[n]$ 记录 F_n 的结果，并用 $\mathrm{dp}[n] = -1$ 表示 F_n 当前还没有被计算过。

```
vector<int> dp(MAXN, -1);
int F(int n) {
   if (n == 0 || n == 1) return 1;
   if (dp[n] != -1) return dp[n];
   else {
      dp[n] = F(n - 1) + F(n - 2);
      return dp[n];
   }
}
```

8.1.3 递推写法

数塔问题 将一些数字排成数塔形状，第一层一个数字，……，第 n 层有 n 个数字。现在要从第一层走到第 n 层，每次只能走向下一层链接的两个数字中的一个，问最后将路径上所有数字相加后得到的和的最大值是多少。

令 $f[i][j]$ 存放第 i 层的第 j 个数字，$\mathrm{dp}[i][j]$ 表示从第 i 层第 j 个数字出发的到达最低层的所有路径中能得到的最大和，$\mathrm{dp}[1][1]$ 就是最终想要的答案。

如果要求出从位置 $(1,1)$ 到达最底层的最大和 $\mathrm{dp}[1][1]$，那么一定先求出它的两个子问题 $\mathrm{dp}[2][1]$ 和 $\mathrm{dp}[2][2]$，它们两个的最大值加上 $(1,1)$ 位置的值即为最终结果。即 $\mathrm{dp}[1][1] = \max(\mathrm{dp}[2][1], \mathrm{dp}[2][2]) + f[1][1]$，一般化得：$\mathrm{dp}[i][j] = \max(\mathrm{dp}[i+1][j], \mathrm{dp}[i+1][j+1]) + f[i][j]$，边界 $\mathrm{dp}[n][1..n] = f[n][1..n]$。

将 $\mathrm{dp}[i][j]$ 称为问题的状态，把上面的式子成为**状态转移方程**。此时解题代码可以写为：

```
int f[MAXN][MAXN], dp[MAXN][MAXN], i, j, n;
for (i = 1; i <= n; ++i)
   for (j = 1; j <= n; ++j)
      scanf("%d", &f[i][j]); // 输入数据
for (j = 1; j <= n; ++j)
   dp[n][j] = f[n][j];  // 处理边界
// 从第 n - 1 层不断向上计算出 dp[i][j]
for (i = n - 1; i >= 1; --i)
   for (j = 1; j <= i; ++j)
      dp[i][j] = max(dp[i+1][j], dp[i+1][j+1])
            + f[i][j]; // 状态转移方程
// 输出结果
printf("%d", dp[1][1]);
```

显然，这个题目也可使用递归完成。两者的区别在于：使用**递推写法**的计算方式是**自底向上**，即从边界开始，不断向上解决问题，直到问题解决；使用**递归写法**的计算方式是**自顶向下**，即从目标问题开始，将它分解成子问题的组合，直到分解至边界位置。

如果一个问题的最优解可以由其子问题的最优解有效地构造出来，那么称这个问题拥有**最优子结构**。

一个问题必须拥有重叠子问题和最优子结构，才能使用动态规划去解决。

8.2 最大连续子序列和

给定一个数字序列 $A[1..n]$，求 $i, j (1 \le i \le j \le n)$，使得 $A_i + ... + A_j$ 最大。

令状态 $\mathrm{dp}[i]$ 表示以 $A[i]$ 作为末尾的连续序列的最大和，最终结果就是 $\mathrm{dp}[0..n-1]$ 中的最大值。

当最大和的连续序列只有一个元素 $A[i]$ 时，最大和就是 $A[i]$ 本身；当最大和的连续序列有多个元素，$A[p..i]$ 时，最大和是 $\mathrm{dp}[i-1] + A[i]$，即 $A[p] + ... + A[i-1] + A[i] = \mathrm{dp}[i-1] + A[i]$。于是可以得到状态转移方程：$\mathrm{dp}[i] = \max\{A[i], \mathrm{dp}[i-1] + A[i]$，边界 $\mathrm{dp}[0] = A[0]$。

```
int A[MAXN], dp[MAXN], i;
```

```
for (i = 0; i < n; i++) scanf("%d", &A[i]);
dp[0] = A[0]; // 边界
for (i = 1; i < n; i++)  // 状态转移方程
   dp[i] = max(A[i], dp[i - 1] + A[i]);
printf("%d", *max_element(dp + 1, dp + n));
```

8.3 最长不下降子序列 LIS

在一个数字序列中，找到一个最长的子序列（可以不连续），使得这个子序列是不下降（非递减）的。

例如，现有序列 $A=\{1,2,3,-1,-2,7,9\}$，它的最长不下降子序列是 $\{1,2,3,7,9\}$，长度为 5。

令 dp[i] 表示以 $A[i]$ 结尾的最长不下降子序列长度，这样对 $A[i]$ 来说就会有两种可能：

(1) 如果存在 $A[i]$ 之前的元素 $A[j](j<i)$，使得 $A[j]\le A[i]$ 且 $\text{dp}[j]+1>\text{dp}[i]$，即把 $A[i]$ 跟在以 $A[j]$ 结尾的 LIS 后面时能比当前以 $A[i]$ 结尾的 LIS 长度更长，那么就把 $A[i]$ 跟在 $A[j]$ 结尾的 LIS 后面，形成一条更长的 LIS，即 $\text{dp}[i]=\text{dp}[j]+1$。

(2) 如果 $A[i]$ 之前的元素都比 $A[i]$ 大，那么 $A[i]$ 就只好自己形成一条 LIS，长度为 1，即这个子序列里面只有一个 $A[i]$。

最后以 $A[i]$ 结尾的 LIS 长度就是 (1)(2) 中能形成的最大长度。

由此可以写出状态转移方程：$\text{dp}[i]=\max\{1,\text{dp}[j]+1\}(j\in[0,i-1],A[j]\le A[i])$。转移方程隐含了边界 $\text{dp}[i]=1(i\in[0,n])$。显然只要将 i 从小到大遍历即可求出整个 dp 数组，再从整个 dp 数组中找出最大的那个才是最终结果。

```
int lengthOfLIS(vector<int> a) {
   size_t len = a.size(), i, j;
   if (len <= 0) return 0;
   vector<int> dp(len, 1); // 初始边界条件
   int ans = -1;
   for (i = 0; i < len; i++) {
      for (j = 0; j < i; j++)
         if (a[j] < a[i] && dp[j] + 1 > dp[i])
            dp[i] = dp[j] + 1; // 状态转移方程
      ans = max(ans, dp[i]);
   }
   return ans;
}
```

8.4 最长公共子序列 LCS

问题描述：给定两个字符串（或数字序列）A 和 B，求一个字符串，使得这个字符串时 A 和 B 的最长公共部分（子序列可以不连续）。

例如：“sadstory”与“adminsorry”的最长公共子序列为“adsory”，长度为 6。

令 dp[i][j] 表示字符串 A 的 i 号位和字符串 B 的 j 号位之前的 LCS 长度（下标从 1 开始），如 dp[4][5] 表示“sads”与“admin”的 LCS 长度。那么可以根据 $A[i]$ 和 $B[j]$ 的情况，分为两种决策：

(1) 若 $A[i]==B[j]$，则字符串 A 与字符串 B 的 LCS 增加了 1 位，即有 $\text{dp}[i][j]=\text{dp}[i-1][j-1]+1$。例如，样例中 dp[4][6] 表示“sads”与“admins”的 LCS 长度，比较 $A[4]$ 与 $B[6]$，发现二者都是 s，因此 dp[4][6] 就等于 dp[3][5] 加 1，即为 3。

(2) 若 $A[i]\ne B[j]$，则字符串 A 的 i 号位和字符串 B 的 j 号位之前的 LCS 无法延长，因此 $\text{dp}[i][j]$ 将会继承 $\text{dp}[i-1][j]$ 与 $\text{dp}[i][j-1]$ 中的较大值，即有 $\text{dp}[i][j]=\max\{\text{dp}[i-1][j],\text{dp}[i][j-1]\}$。

因此可以得到状态转移方程：

$$\text{dp}[i][j]=\begin{cases}\text{dp}[i-1][j-1], & A[i]=B[j]\\ \max\{\text{dp}[i-1][j],\ \text{dp}[i][j-1]\}, & A[i]\ne B[j]\end{cases}$$

边界：$\text{dp}[i][0]=\text{dp}[0][j]=0(0\le i\le n,0\le j\le m)$，由边界出发就可以得到整个 dp 数组，最终 dp[n][m] 就是需要的答案，时间复杂度为 $O(nm)$。

```
char A[100], B[100];
int dp[100][100], i, j;
gets(A + 1); gets(B + 1); // 从下标为 1 开始读入
int lenA = strlen(A + 1), lenB = strlen(B + 1);
// 边界
for (i = 0; i <= lenA; i++)  dp[i][0] = 0;
for (j = 0; j <= lenB; j++)  dp[0][j] = 0;
// 状态转移方程
for (i = 1; i <= lenA; i++)
   for (j = 1; j <= lenB; j++) {
      if (A[i] == B[j])
         dp[i][j] = dp[i - 1][j - 1] + 1;
      else
         dp[i][j] = max(dp[i-1][j],dp[i][j-1]);
   }
printf("%d", dp[lenA][lenB]); // 结果
```

8.5 最长回文子串

给出一个字符串 S，求 S 的最长回文子串。例如 PATZJUJZTACCBCC 的最长回文子串是 ATZJUJZTA，长度为 9。

令 $\mathrm{dp}[i][j]$ 表示 $S[i..j]$ 所表示的子串是否是回文子串，是为 1，不是为 0。

若 $S[i]=S[j]$，那么只要 $S[i+1..j-1]$ 是回文子串，$S[i..j]$ 就是回文子串；如果 $S[i+1..j-1]$ 不是回文子串，则 $S[i..j]$ 也不是回文子串。

若 $S[i]\neq S[j]$，那么 $S[i..j]$ 一定不是回文子串。

由此可以写出状态转移方程：

$$\mathrm{dp}[i][j]=\begin{cases}\mathrm{dp}[i+1][j-1], & S[i]=S[j]\\ 0, & S[i]\neq S[j]\end{cases}$$

边界为 $\mathrm{dp}[i][i]=1, \mathrm{dp}[i][i+1]=(S[i]=S[i+1])?1:0$。

9 字符串