



C++编程



目录

- 函数
- 递归
- 不定长输入
- 无穷大
- 排序和去重
- 埃氏筛
- 线性筛
- 二分查找
- 在线和离线
- 二维数组
- 字符数组
- 字符串
- 高精度运算
- 结构体
- 文件操作
- 快速读入
- 位运算

C++ 和 ++C

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a = 1, b;
    b = a ++;
    cout << a << " " << b;
    return 0;
}
```

```
2 1
-----
Process exited after 0.0988 seconds w
ith return value 0
请按任意键继续. . .
```

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a = 1, b;
    b = ++ a;
    cout << a << " " << b;
    return 0;
}
```

```
2 2
-----
Process exited after 0.08708 seconds
with return value 0
请按任意键继续. . .
```

数据类型与范围

- 一般我们近似认为：

- 整型 **int** 10^9

（精确值： $2^{31}-1=2147483647$ ）

- 长整型 **long long** 10^{18}

（精确值： $2^{63}-1=9223372036854775807$ ）

```
int a, b;  
long long n, m;
```

交换函数 swap

- 右边的代码可以改成左边

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a, b;
    cin >> a >> b;
    swap(a, b);
    cout << a << " " << b;
    return 0;
}
```

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a, b, c;
    cin >> a >> b;
    c = a;
    a = b;
    b = c;
    cout << a << " " << b;
    return 0;
}
```

min 函数和 max 函数

- 输出a、b中的最小值

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a = 2, b = 5;
    cout << min(a, b);
    return 0;
}
```

```
2
-----
Process exited after 0.0913 seconds w
ith return value 0
请按任意键继续. . .
```

- 输出a、b中的最大值

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a = 2, b = 5;
    cout << max(a, b);
    return 0;
}
```

```
5
-----
Process exited after 0.0932 seconds w
ith return value 0
请按任意键继续. . .
```

floor 函数 和 ceil 函数

- 对浮点数向下取整

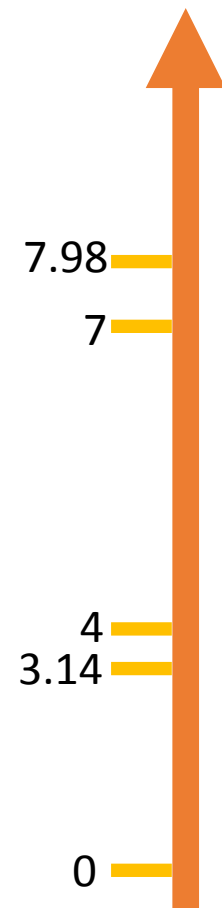
```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    float a = 7.98;
    cout << floor(a);
    return 0;
}
```

```
7
-----
Process exited after 0.09883 seconds
with return value 0
请按任意键继续. . .
```

- 对浮点数向上取整

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    float a = 3.14;
    cout << ceil(a);
    return 0;
}
```

```
4
-----
Process exited after 0.09262 seconds
with return value 0
请按任意键继续. . .
```



round 函数

- 对浮点数四舍五入

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    float a = 7.98, b = 3.14;
    cout << round(a) << " " << round(b);
    return 0;
}
```

8 3

Process exited after 0.2209 seconds w
ith return value 0
请按任意键继续. . .

自定义函数

什么是函数？

- 一种对应关系，对每个自变量，得到一个返回值

main()

swap(a,b)

min(a,b) / max(a,b)

floor(x) / ceil(x) / round(x)

pow(a,b)

自定义abs函数

自定义函数中的自变量名，可以和主函数中的自变量同名，但是建议不同名



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

```
int myabs(int x)
{
    if (x < 0) return -x;
    else return x;
}
```

```
int main()
{
    int x;
    cin >> x;
    cout << myabs(x);
    return 0;
}
```

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int x;
    cin >> x;
    cout << abs(x);
    return 0;
}
```

自定义min函数（max函数同理）

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

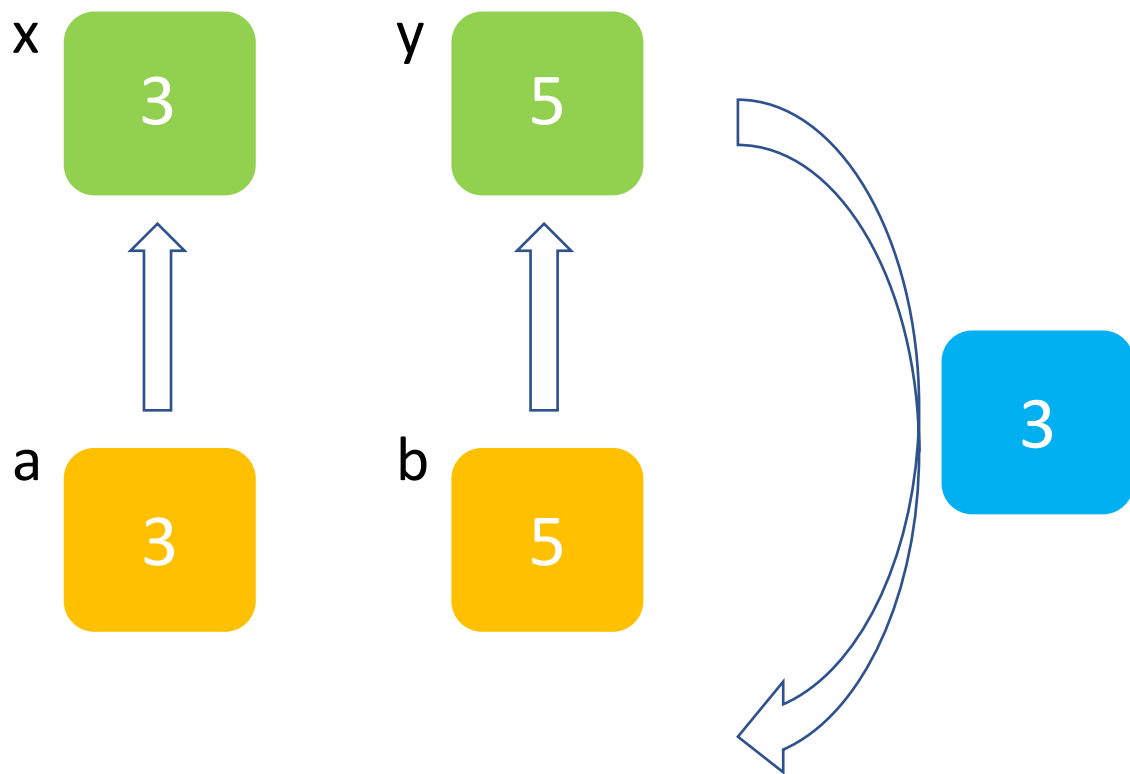
int mymin(int x, int y)
{
    if (x < y) return x;
    else return y;
}

int main()
{
    int a, b;
    cin >> a >> b;
    cout << mymin(a, b);
    return 0;
}
```

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a, b;
    cin >> a >> b;
    cout << min(a, b);
    return 0;
}
```

自定义min函数

- 这里的过程被称为“传值”



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

int mymin(int x, int y)
{
    if (x < y) return x;
    else return y;
}

int main()
{
    int a, b;
    cin >> a >> b;
    cout << mymin(a, b);
    return 0;
}
```

三元运算符

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

int mymin(int x, int y)
{
    return x < y ? x : y;
}

int main()
{
    int a, b;
    cin >> a >> b;
    cout << mymin(a, b);
    return 0;
}
```

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

int mymin(int x, int y)
{
    if (x < y) return x;
    else return y;
}

int main()
{
    int a, b;
    cin >> a >> b;
    cout << mymin(a, b);
    return 0;
}
```

自定义pow函数

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

long long mypow(int a, int b)
{
    long long ans = 1;
    while (b --) ans = ans * a;
    return ans;
}

int main()
{
    int x, y;
    cin >> x >> y;
    cout << mypow(x, y);
    return 0;
}
```

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int x, y;
    cin >> x >> y;
    cout << pow(x, y);
    return 0;
}
```

自定义swap函数

等等，这里
有问题！



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

void myswap(int x, int y)
//函数不需要返回值，定义为 void类型
{
    int temp = x;
    x = y;
    y = temp;
}
```

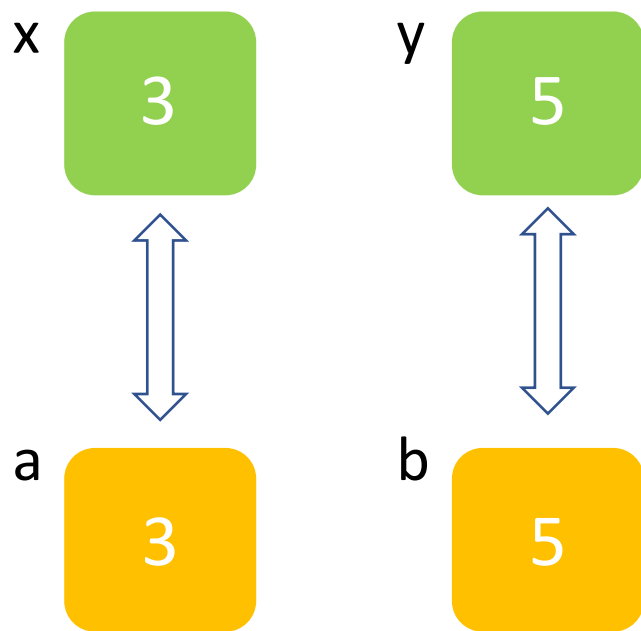
```
int main()
{
    int a, b;
    cin >> a >> b;
    myswap(a, b);
    cout << a << " " << b;
    return 0;
}
```



```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a, b;
    cin >> a >> b;
    swap(a, b);
    cout << a << " " << b;
    return 0;
}
```

自定义swap函数

- 地址传值



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

void myswap(int &x, int &y)
//这里必须使用地址传值
{
    int temp = x;
    x = y;
    y = temp;
}

int main()
{
    int a, b;
    cin >> a >> b;
    myswap(a, b);
    cout << a << " " << b;
    return 0;
}
```


挑战一下：用函数形式改写“质数判定”

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int i, n, flag = 1;
    cin >> n;
    for (i = 2; i * i <= n; i++)
        if (n % i == 0) { flag = 0; break; }

    if (flag == 1 && n != 1) cout << n << "是质数";
    else cout << n << "不是质数";
    return 0;
}
```

这里的break,
用于提前结束循环



应战

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

bool prime(int x)
{
    if (x == 1) return false; // 数字 1 要特判
    for (int i = 2; i * i <= x; i++)
        if (x % i == 0) return false; // return 自带 break 功能
    return true; // 循环结束后还未返回 false 的必定是质数
}

int main()
{
    int n;
    cin >> n;
    if (prime(n) == true) cout << n << "是质数";
    else cout << n << "不是质数";
    return 0;
}
```

这里可以简化写:
if (prime(n))



递归调用

- 简单来说：就是一个自定义程序，它里面又调用了它自己

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

long long fac(int n)
{
    if (n == 0) return 1;
    return n * fac(n - 1); //这里调用了 fac() 自己
}

int main()
{
    int n;
    cin >> n;
    cout << fac(n);
    return 0;
}
```

注意在一个函数
内部不允许再定义
函数，只可以再调
用函数



递归调用

- 递归调用有两个要点:
 1. 有明确的递归边界
 2. 有明确的递归方向

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

long long fac(int n)
{
    if (n == 0) return 1;
    return n * fac(n - 1);
}

int main()
{
    int n;
    cin >> n;
    cout << fac(n);
    return 0;
}
```

递归调用的流程

- 目测左右两段程序的输出结果

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

void test(int x)
{
    int m = x / 2;
    cout << m << endl;
    if (m > 1) test(m);
}

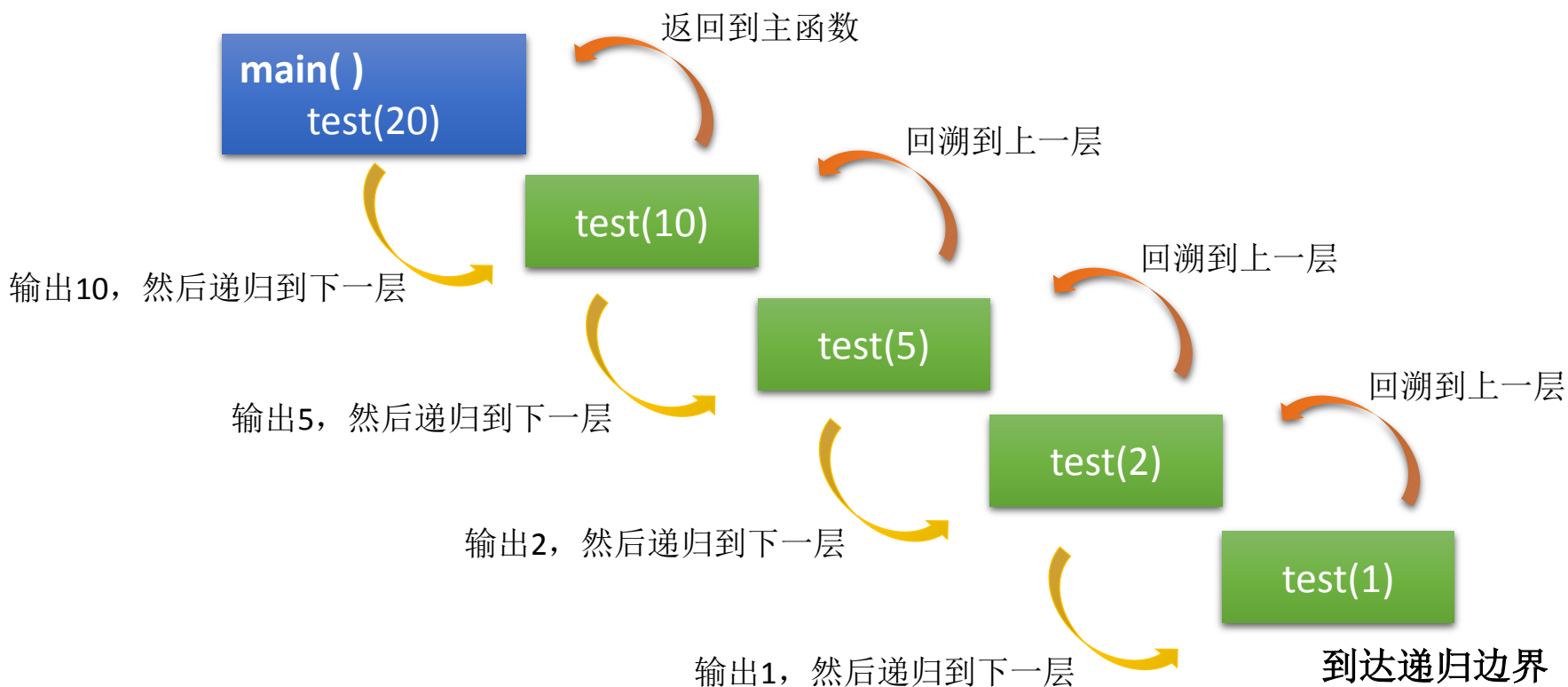
int main()
{
    test(20);
    return 0;
}
```

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

void test(int x)
{
    int m = x / 2;
    if (m > 1) test(m);
    cout << m << endl;
}

int main()
{
    test(20);
    return 0;
}
```

递归调用的流程

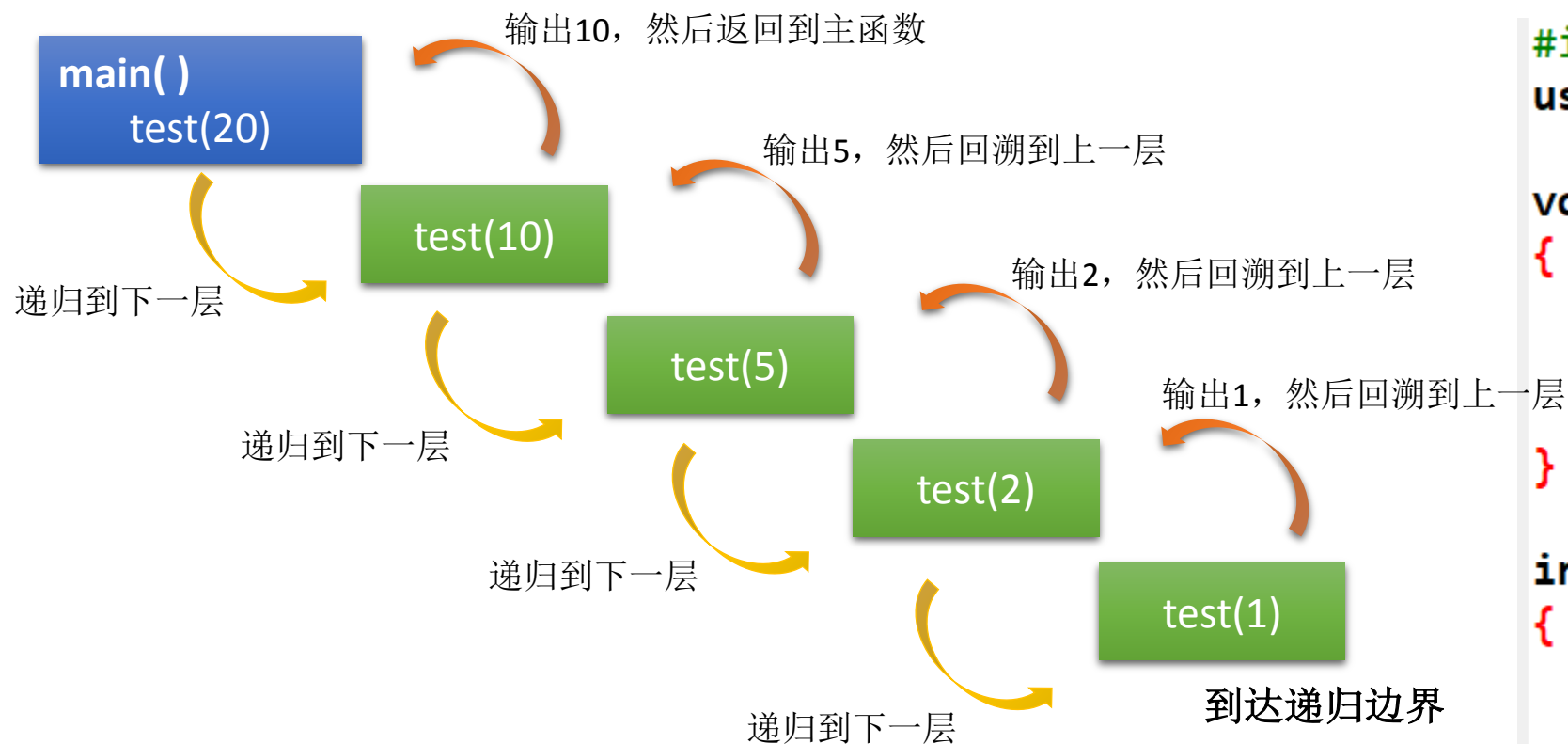


```
#include<bits/stdc++.h>
using namespace std;

void test(int x)
{
    int m = x / 2;
    cout << m << endl;
    if (m > 1) test(m);
}

int main()
{
    test(20);
    return 0;
}
```

递归调用的流程



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

void test(int x)
{
    int m = x / 2;
    if (m > 1) test(m);
    cout << m << endl;
}

int main()
{
    test(20);
    return 0;
}
```

递归调用的优缺点

这里的“保留”，无需我们写代码实现，而是程序自动借助于一个叫“栈”的数据结构来完成



- 程序在递归调用的过程中，递归到的每一层，其中间结果都需要**按递归顺序**保留住（直到当前层已经回溯完毕），否则将无法逆序回溯。
- 因为这个特性，所以（同一个问题的）递归形式在空间消耗和时间消耗上，其实比非递归形式要大。
- 若一个问题可以写成递归的形式，将极大缓解人脑的思维强度。

递归和非递归

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

long long fac(int n)
{
    long long ans = 1;
    for (int i = 2; i <= n; i++)
        ans = ans * i;
    return ans;
}

int main()
{
    int n;
    cin >> n;
    cout << fac(n);
    return 0;
}
```

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

long long fac(int n)
{
    if (n == 0) return 1;
    return n * fac(n - 1);
}

int main()
{
    int n;
    cin >> n;
    cout << fac(n);
    return 0;
}
```

挑战： 以函数形式改写如下程序

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int n, m, r;
    cin >> n >> m;
    while (n % m != 0)
    {
        r = n % m;
        n = m;
        m = r;
    }
    cout << m;
    return 0;
}
```

挑战： 以递归形式继续改写

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

int gcd(int n, int m)
{
    while (n % m != 0)
    {
        int r = n % m;
        n = m;
        m = r;
    }
    return m;
}

int main()
{
    int n, m;
    cin >> n >> m;
    cout << gcd(n, m);
    return 0;
}
```

挑战：以三元运算符继续精简

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

int gcd(int n, int m)
{
    if (m == 0) return n;
    return gcd(m, n % m);
}

int main()
{
    int n, m;
    cin >> n >> m;
    cout << gcd(n, m);
    return 0;
}
```

这里的递归边界，要仔细理解



最后写这样

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

int gcd(int n, int m)
{
    return (m == 0) ? n : gcd(m, n % m);
}

int main()
{
    int n, m;
    cin >> n >> m;
    cout << gcd(n, m);
    return 0;
}
```

挑战

- 把一个十进制正整数 x ($1 \leq x \leq 100,000$)，转换成 m 进制数 ($2 \leq m \leq 10$)

Sample input		Sample output
12	//x	14
8	//m	

分析

十进制数 x 转 m 进制数的办法是：

1. x 除 m 直到 x 被除尽，
2. 然后逐个记录余数，
3. 余数的逆序列即答案

- 然后我们遇到了困难：我们可以记录这些余数，但我们无法预知余数会有多少个，也无法逆序输出它们

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

void convert(int x, int m)
{
    while (x > 0)
    {
        int r = x % m;
        x = x / m;
    }
    逆序输出记录的这些余数;
}

int main()
{
    int x, m;
    scanf("%d%d", &x, &m);
    convert(x, m);
    return 0;
}
```

分析

- 怎么应用递归的思路呢？
- 递归的边界？
- 递归的方向？
- 怎么解决逆序输出？

参考代码

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

void convert(int x, int m)
{
    if (x / m > 0) convert(x / m, m);
    printf("%d", x % m);
}

int main()
{
    int x, m;
    scanf("%d%d", &x, &m);
    convert(x, m);
    return 0;
}
```

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

void convert(int x, int m)
{
    int r[50010], i = 0;
    while (x > 0)
    {
        r[++ i] = x % m;
        x = x / m;
    }
    for ( ; i > 0; i --) printf("%d", r[i]);
}

int main()
{
    int x, m;
    scanf("%d%d", &x, &m);
    convert(x, m);
    return 0;
}
```

Rand 函数

- Rand()函数用于生成随机数，其范围是 [0, 32767)

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a, n = 100;
    while (n --) //随机生成 100个数
    {
        a = rand();
        printf("%d\n", a);
    }
    return 0;
}
```

Rand 函数

伪随机的意思，就是虽然每次生成的数看起来无规律，但其实每次都是同样的数



- 但是 rand 函数内部是用线性同余法实现的，需要给一个不同的种子（seed），才能实现真正的随机

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a, n = 100;
    srand(time(0)); // 通常使用 time 来做种子
    while (n --) // 随机生成 100 个数
    {
        a = rand();
        printf("%d\n", a);
    }
    return 0;
}
```

Rand 函数

- 如果要限定随机数产生的范围，可以这样：

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a, n = 100;
    srand(time(0));
    while (n --)
    {
        a = rand() % 100; //生成不大于 100 的随机数
        printf("%d\n", a);
    }
    return 0;
}
```

Rand 函数

- 或者:

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a, n = 100;
    srand(time(0));
    while (n --)
    {
        a = rand() % 100 + 300; //生成随机数于 300~400之间
        printf("%d\n", a);
    }
    return 0;
}
```

趣味程序：猜数游戏

- 计算机随机给出一个不大于100的正整数，用户每猜一次，都会提示“猜大了”、“猜小了”、“恭喜猜中了”
- 给五次机会，五次后都未猜中，提示“失败了”

分析

- 基本思路：计算机调用随机函数，生成一个数，然后用户输入一个数，判断这两个数是否相等即可，并根据判断结果给出提示

```
随机产生 a;  
while (没猜中 && 次数没用光)  
{  
    猜一个数 b;  
    次数减 1;  
    if (a > b) 提示猜小了;  
    if (a < b) 提示猜大了;  
    if (a == b) 提示猜中了;  
}
```

分析

- 这里有四种不同的提示，我们可以把它们分成两类：
- “猜大了”、“猜小了”是一类，“恭喜猜中了”、“失败了”又是一类，划分的依据则是：前一类可以发生多次，应该放在循环内，后一类只可能发生一次，可放在循环外

分析

```
随机产生 a;  
while (没猜中 && 次数没用光)  
{  
    猜一个数 b;  
    次数减 1;  
    if (a > b) 提示猜小了;  
    if (a < b) 提示猜大了;  
}  
if (a == b) 提示猜中了;  
else 提示失败了;
```

参考代码

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a, b, cnt = 5;
    srand(time(0));
    a = rand() % 100 + 1;
    while (cnt > 0 && a != b)
    {
        scanf("%d", &b);
        cnt --;
        if (a > b) printf("猜小了");
        if (a < b) printf("猜大了");
    }
    if (a == b) printf("恭喜猜中了");
    else printf("失败了");
    return 0;
}
```

完善一下

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a, b, cnt = 5;
    srand(time(0));
    a = rand() % 100 + 1;
    while (cnt > 0 && a != b)
    {
        printf("请输入一个不大于100的正整数: ");
        scanf("%d", &b);
        cnt --;
        if (a > b && cnt > 0) printf("猜小了,您还有%d次机会\n\n", cnt);
        if (a < b && cnt > 0) printf("猜大了,您还有%d次机会\n\n", cnt);
    }
    if (a == b) printf("恭喜猜中了\n");
    else printf("失败了\n");
    return 0;
}
```

不定长输入

- 输入数据个数未知，称为不定长输入

```
3 5 1 2 9^Z
5个数的和是: 20
-----
Process exited after 7.321 seconds wi
th return value 0
请按任意键继续. . .
```

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int x, sum = 0, cnt = 0;
    while (scanf("%d", &x) == 1)
        //原来 scanf函数居然有返回值!
        {
            sum += x;
            cnt ++;
        }
    printf("%d个数的和是: %d", cnt, sum);
    return 0;
}
```

不定长输入

- 也可以用 cin 输入，效果是一样的

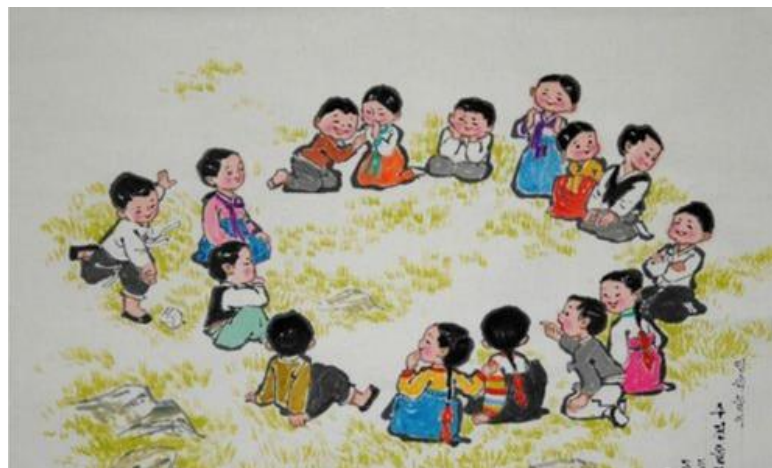
```
3 5 1 2 9^Z
5个数的和是: 20
-----
Process exited after 7.321 seconds wi
th return value 0
请按任意键继续. . .
```

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int x, sum = 0, cnt = 0;
    while (cin >> x)
        //输入数据成功, 即返回 true
        {
            sum += x;
            cnt ++;
        }
    cout << cnt << "个数的和是: " << sum;
    return 0;
}
```

约瑟夫问题

- 编号为 $1 \sim n$ 的 n ($n \leq 1000$) 个人围坐一圈，从第一个人开始报数，报到 m ($m \leq n$) 的人出局，重复这一过程直至剩下一个人即为获胜者。
- 输出所有人出列的先后顺序（最后一位即为获胜者）

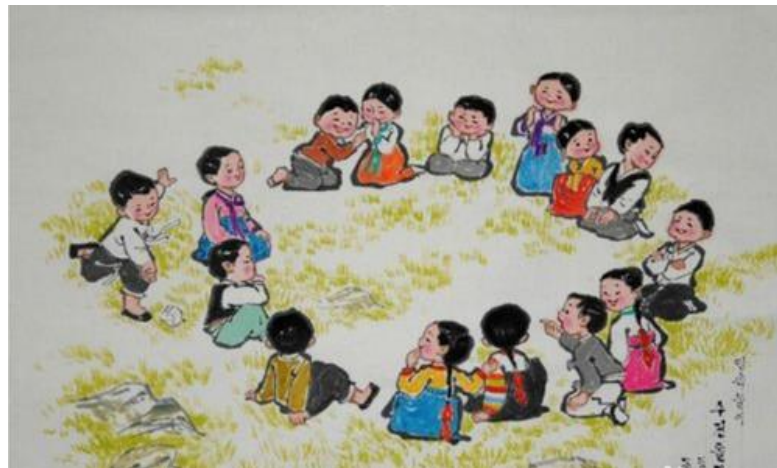
Sample input		Sample output
7	//n	3 6 2 7 5 1 4
3	//m	



分析

- 首先我们无法预知该游戏进行几轮后结束，所以判定游戏结束的标志是所有人出局，需要记录出局人数

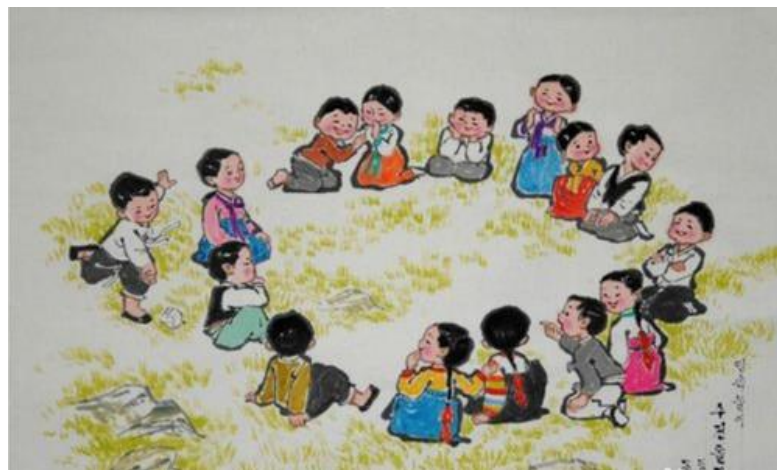
```
while (记录的出局人数 < 初始人数 n)
{
}
}
```



分析

- 循环内要做什么呢？
- 首先需要有一个大圈报数，而且要往复进行（因为是环形）

```
while (记录的出局人数 < 初始人数 n)
{
    从编号1开始大圈报数
    一旦发现报数到  $n+1$ ，就要重置为 1
}
```

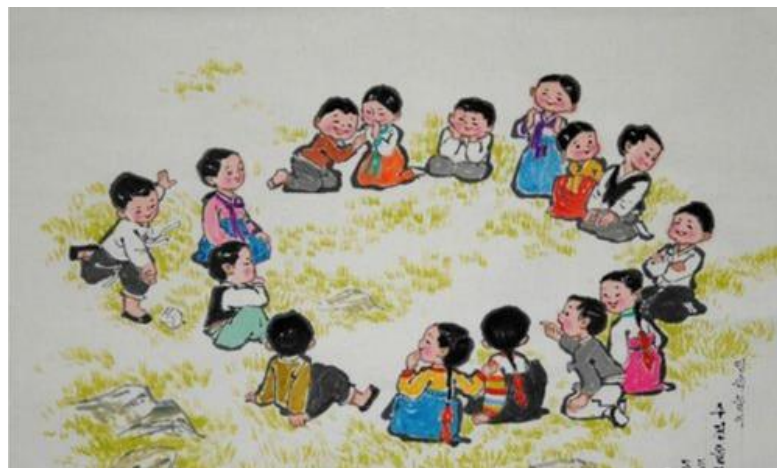


分析

- 接下来要小圈报数
- 小圈报数只有未出局的人才能参与，所以需要（用数组）记录每个人的状态

```
while (记录的出局人数 < 初始人数 n)
{
    从编号1开始循环报数
    一旦发现报到  $n+1$ ，就要重置为 1

    未出局的人，参与小圈报数直到 m
}
```



分析

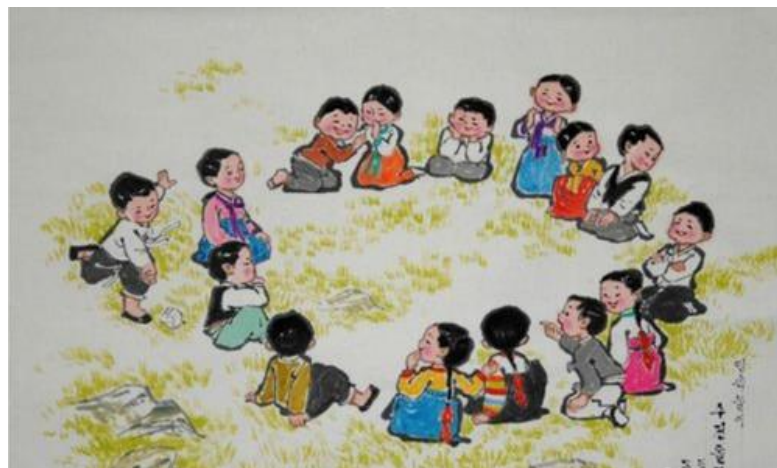
- 接下来就要处理小圈报数到 m 的人

1. 输出该位置并修改对应位置的数组标记为“出局”
2. 出局人数+1，并且小圈报数要清零（大圈报数不必）

```
while (记录的出局人数 < 初始人数 n)
{
    从编号1开始循环报数
    一旦发现报到  $n+1$ ，就要重置为 1

    未出局的人，参与小圈报数

    处理小圈报数到 $m$ 的那个人（出局）
}
```



参考代码

这里的
if(!a[i]) 就是:
if(a[i]==0)



```
#include<bits/stdc++.h>
using namespace std;
int a[1010], n, m, i, j, sum; // 数组初始值 0代表未出局
int main()
{
    scanf("%d%d", &n, &m);
    while (sum < n)
    {
        i ++; if (i == n + 1) i = 1; // 从编号 1 开始循环报数
        if (!a[i]) j ++; // 未出队的参与小圈报数
        if (j == m) // 处理出队的编号
        {
            printf("%d ", i);
            a[i] = 1; // 修改对应数组元素为 1, 代表已出局
            sum ++;
            j = 0;
        }
    }
    return 0;
}
```

memset 函数

- 用于数组整体初始化
- 如果是对高维数组/结构体数组之类，方便程度更甚

```
int a[10010];  
memset(a, 0, sizeof(a));
```

```
int a[10010];  
for (int i = 0; i < 10010; i ++)  
    a[i] = 0;
```

memset 函数

- 但要注意它是按字节对指定内存区域进行初始化的
- 所以memset一般只用来清零或者置-1，或者赋极大值，而不可贸然赋 1 之类

```
#include<bits/stdc++.h>
using namespace std;
int a[100010], b[100010], c[100010];
int main()
{
    memset(a, 0, sizeof(a));
    memset(b, -1, sizeof(b));
    memset(c, 0x3f, sizeof(c));
    return 0;
}
```

漫谈无穷大/极大值：INF



- 我们现在设置无穷大/极大值（常表示为：INF），一般是用2147483647
- 但是这个数有个问题：它处在溢出的边沿，稍有不慎就溢出，实在危险
- 那怎么办？因为我们以前说过：int 型上限可视作九位数，所以可以设置为999999999，它足够大了，又比2147483647安全
- 但一来看起来很不专业，二来非常容易数错，尬
- 所以又衍生出1234567890这种，但依然很LOW的样子

漫谈无穷大/极大值：INF



- 有一种做法是设置为十六进制数：0x7fffffff
- 除了“看起来”很专业这一点外，并没有比2147483647更高明

```
#include<bits/stdc++.h>
using namespace std;
const int INF = 0x7fffffff; // const为定义常量
int main()
{
    int a = INF;
    printf("%d", a);
    return 0;
}
```

```
2147483647
-----
Process exited after 0.1018 seconds w
ith return value 0
请按任意键继续. . .
```



漫谈无穷大/极大值：INF

- 终极做法是设置为十六进制数：0x3f3f3f3f
- 这个值1061109567，足够大，又安全
- 但看起来平淡无奇？

```
#include<bits/stdc++.h>
using namespace std;
const int INF = 0x3f3f3f3f; // const为定义常量
int main()
{
    int a = INF;
    printf("%d", a);
    return 0;
}
```

1061109567

Process exited after 0.1422 seconds w
ith return value 0
请按任意键继续. . .

0x3f3f3f3f的精妙之处



- 0x3f3f3f3f不仅仅是凭“看起来更专业”战胜1234567890之流的
- 首先，1234567890处理不了无穷大+无穷大的情况，但0x3f3f3f3f可以
- 其次，使用memset为数组整体赋极大值的时候，只能使用0x3f（之前介绍的0x7f也可以）


memset 赋极大值



读音: [miào] 🔊

部首: 女 五笔: VITT

释义: 1.好; 美妙。 2.神奇; 巧妙; 奥妙。 3.姓。



```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a[10];
    memset(a, 1234567890, sizeof(a));
    for (int i = 0; i < 10; i ++)
        printf("%d ", a[i]);
    return 0;
}
```

```
-757935406 -757935406 -757935406 -757
935406 -757935406 -757935406 -7579354
06 -757935406 -757935406 -757935406
```

```
-----
Process exited after 0.02245 seconds
with return value 0
请按任意键继续. . .
```

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int a[10];
    memset(a, 0x3f, sizeof(a));
    for (int i = 0; i < 10; i ++)
        printf("%d ", a[i]);
    return 0;
}
```

```
1061109567 1061109567 1061109567 1061
109567 1061109567 1061109567 10611095
67 1061109567 1061109567 1061109567
```

```
-----
Process exited after 0.1341 seconds w
ith return value 0
请按任意键继续. . .
```

Sort 函数

- STL自带的排序方法，书写简单+排序高效，简直是bug一般的存在

```
#include<bits/stdc++.h>
using namespace std;
int a[1010];
int main()
{
    int i, n;
    scanf("%d", &n);
    for (i = 0; i < n; i ++) scanf("%d", &a[i]);

    sort(a, a + n); //对数组[0,n)范围内的元素按从小到大排序

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

Sort 函数

- 特别的，如果数组下标是从 1 开始：

```
#include<bits/stdc++.h>
using namespace std;
int a[1010];
int main()
{
    int i, n;
    scanf("%d", &n);
    for (i = 1; i <= n; i++) scanf("%d", &a[i]);

    sort(a + 1, a + n + 1); //对数组[1,n+1)范围内的元素按从小到大排序

    for (i = 1; i <= n; i++) printf("%d ", a[i]);
    return 0;
}
```

Sort 函数

- 冒泡排序和选择排序的时间复杂度都是 $O(n^2)$ 的
- `Sort`函数内部是用快速排序实现的，时间复杂度 $O(n\log n)$

还是 `Sort` 大法好啊！

Unique 函数

```
5
1 0 0 8 6
1 0 8 6 6
-----
Process exited after 9.114 seconds with return value 0
请按任意键继续. . .
```

- STL自带的去重方法，可以把重复数字从序列中去除（只保留一个）

```
#include<bits/stdc++.h>
using namespace std;
int a[1010];
int main()
{
    int i, n;
    scanf("%d", &n);
    for (i = 0; i < n; i++) scanf("%d", &a[i]);

    unique(a, a + n); // 去除数组[0,n)范围内的重复数字

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



Unique 函数

```
5
1 0 0 8 6
1 0 8 6
-----
Process exited after 7.011 seconds with return value 0
请按任意键继续. . .
```



- 去重后，数字个数 \leq 原有个数，所以我们还需要取得去重后的序列长度

```
#include<bits/stdc++.h>
using namespace std;
int a[1010];
int main()
{
    int i, n;
    scanf("%d", &n);
    for (i = 0; i < n; i++) scanf("%d", &a[i]);

    int len = unique(a, a + n) - a; // 去重的同时，记录去重后的序列长度

    for (i = 0; i < len; i++) printf("%d ", a[i]);
    return 0;
}
```

严格第k小整数

- 现有n ($n \leq 10000$) 个正整数，要求其中严格第k ($k \leq 1000$) 小的整数（相同大小的数只计算一次），若该数不存在则输出-1

Sample input	Sample output
10 3 //n和k 1 3 3 7 2 5 1 2 4 6	3

参考代码

```
#include<bits/stdc++.h>
using namespace std;
int a[10010], n, k;

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

    sort(a, a + n);
    int len = unique(a, a + n) - a;

    if (k < len) printf("%d", a[k]);
    else printf("-1"); //注意无解的情况
    return 0;
}
```

Eratosthenes 筛素数法

- 埃拉托斯特尼，古希腊数学家，由他首创的该算法简称埃氏筛法
- 大家应该还记得我们之前写的按素数定义判定的算法：从 $2 \sim \sqrt{n}$ 扫一遍，看看是否能被整除

定义判定法

- 默认忽略数字 1

```
#include<bits/stdc++.h>
using namespace std;
bool prime(int x)
{
    for (int i = 2; i * i <= x; i++)
        if (x % i == 0) return false;
    return true;
}

int main()
{
    int n;
    scanf("%d", &n);
    for (int i = 2; i <= n; i++)
        if (prime(i)) printf("%d ", i);
    return 0;
}
```

Eratosthenes 筛素数法

- 埃氏筛法基于唯一分解定理：任意整数 n ，都可以分解为若干个素数的乘积，而且这种分解方式是唯一的。

$$n = a_1^{p_1} \times a_2^{p_2} \times a_3^{p_3} \times \dots \times a_k^{p_k} \quad (a_1, a_2, \dots \text{均为素数})$$

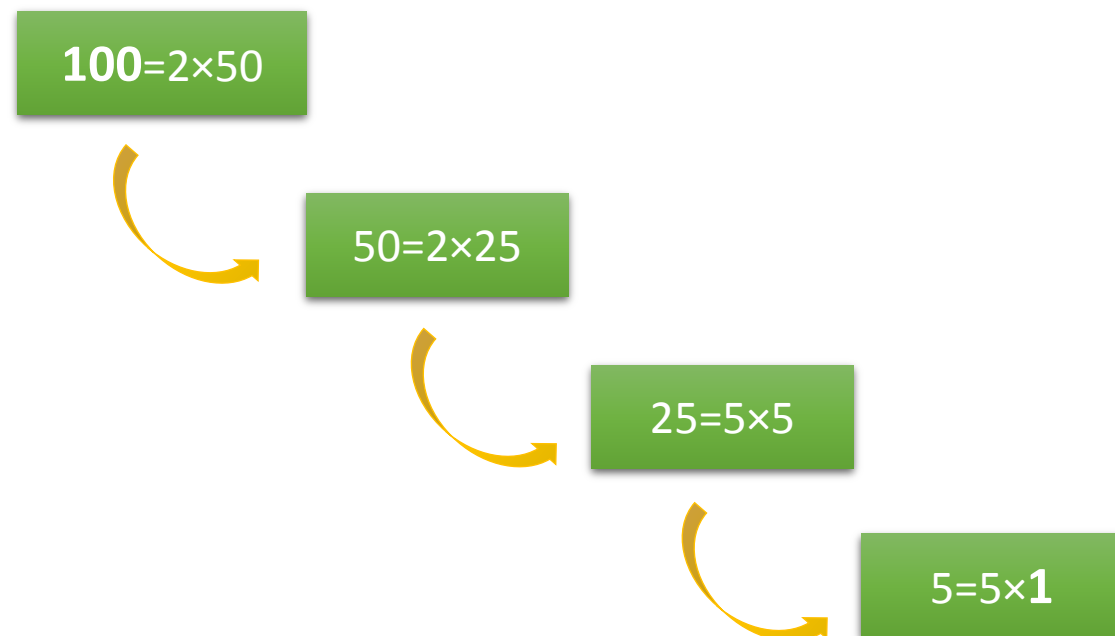
分解质因子

- 为了更清楚了解这一点，我们很生硬地插入一道题：把任意正整数 n （由输入给定，`int`范围），分解为若干素数的乘积

Sample input	Sample output
100	$2 \times 2 \times 5 \times 5$

分析

- 每次拆出一个数 2
- 如果拆不出，就试一试 +1
- 一直拆到剩下 1，算法结束
- 需要判断素数吗？
- 如果不需要，那拆出非素数？
- 小心处理乘号



分析

- 每次拆出一个数 2
- 如果拆不出，就试一试 +1
- 一直拆到剩下 1，算法结束
- 需要判断素数吗？
- 如果不需要，那拆出非素数？
- 小心处理乘号

```
while (n 还可以被拆开)
{
    if (能拆出数 i)
    {
        拆出 i;
        处理乘号;
    }
    else i ++;
}
```

参考代码

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    int n, i = 2;
    scanf("%d", &n);
    while (n > 1)
    {
        if (n % i == 0)
        {
            printf("%d", i);
            n /= i;
            if (n > 1) printf("x ");
        }
        else i ++;
    }
    return 0;
}
```

2147483
13×13×97×131

Process exited after 2.388 seconds with return value 0
请按任意键继续. . .

Eratosthenes 筛素数法

- 埃氏筛法基于唯一分解定理：素数的倍数不可能再是素数。

1. 从2开始，把2的所有倍数划掉（不包括2自己，下同）
2. 依次类推，只要未被划掉的数，就把它的所有倍数划掉
3. 最后剩下的数，即为素数

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30

参考代码

同学们要
仔细研究内层
循环的范围



```
#include<bits/stdc++.h>
using namespace std;
int a[100010], i, j, n;    // a[] 初值 0 表示未被筛去
void prime()
{
    for (i = 2; i * i <= n; i++) if (!a[i])
        for (j = i + i; j <= n; j += i) a[j] = 1;
}

int main()
{
    scanf("%d", &n);
    prime();
    for (i = 2; i <= n; i++)
        if (!a[i]) printf("%d ", i);
    return 0;
}
```

效率对比

- 数据规模 5×10^6

试题		选手				
名称	排名	prime	总分	总用时(s)	测试时间	
埃氏筛法	1	100	100	0.625	2018/12/20 20:43:09	
定义判定法	1	100	100	3.312	2018/12/20 20:43:17	

选手: 埃氏筛法

试题 prime (重新测试)
源程序: prime.cpp

测试点	输入文件	测试结果	运行用时	内存消耗	得分
#1	prime.in	答案正确	0.625 s	21.281 MB	100

选手: 定义判定法

试题 prime (重新测试)
源程序: prime.cpp

测试点	输入文件	测试结果	运行用时	内存消耗	得分
#1	prime.in	答案正确	3.312 s	2.207 MB	100

改进

- 埃氏筛素数法还可以在一些小细节上做改进，比如一开始就把除2以外的偶数全筛掉，然后循环从3开始，这样数据规模缩小一半
- 但无论怎样改进细节，它的复杂度一般认为是 $O(n \log \log n)$ ，接下来我们再谈一个复杂度可以达到 $O(n)$ 的算法

筛质数

- 大家已经熟知埃氏筛质数了
- 埃氏筛是可以继续优化的
 - 比如一开始就把除2以外的所有偶数筛掉，然后循环从3开始并且只考虑奇数
 - 或者更彻底一点，数组里就只存奇数，`a[0]`存3、`a[1]`存5、`a[2]`存7、.....

线性筛

- 但是埃氏筛无论如何优化，也仅仅是优化常数而无法做到 $O(n)$
- 原因是它有一个天然的缺陷：一个数有很大几率被筛了多次
- 比如12，被当作2的倍数筛过一次，又被当作3的倍数筛了一次
- 105更严重，分别被3、5、7各筛一遍（还记得那道分支结构题吗）
- 所以线性筛的核心思想就是：一个数只被筛一次

线性筛

- 要想做到一个数只被筛一次，那就必须做到一个数只能被一个因子数筛，不许其他的因子数“多管闲事”
- 所以，12应该被谁管？（它的因子数有2、3、4、6，挑一个吧）
- 首先可以明确的是，只有离12尽可能近一点才是正解，否则就会回到2筛一遍，3再筛一遍的局面
- 所以选6？现在还不明朗
- 再看105，它的因子有3、5、7、15、21、35
- 现在好像清晰一点了，感觉选35会更优

线性筛

- 为了摸清规律，我们反过来考虑：
 - 2，筛掉4，停下（6不归2筛，否则会 and 3冲突）
 - 3，筛掉6、9（3若不筛9，9就筛不掉了）
 - 4，筛掉8，停下（道理同2）
 - 5，筛掉10、15、25（道理同3）
 - 6，筛掉12，停下（刚刚才选过，道理同2）
 - 11？

线性筛

–11?

–22、33、55、77、121（44交给22，66、99交给33，88交给44，110交给？）

- 如果你能完整回答11，那基本上规律就理清了：
- 设 p 是 n 的最小质因子，则数 n 交给 n / p 来筛
- 比如110，因子有2、5、10、11、22、55，最小质因子是2，所以交给 $110 / 2 = 55$ 来筛

线性筛

- 这个规律虽然是正解，但是并不方便写代码
- 因为我们若考查一个数 n ，还要回头去找它的最小质因子，无疑是浪费时间
- 要记得我们是从2出发往大数方向走的，所以这个规律需要有另一种表述

线性筛

- 最后再看一个数：35，按总结出来的规律，70、105都要交给它
- 别忘了还有： $175 = 35 \times 5$ ，完整的就是： **35×2** ， **35×3** ， **35×5**
- $35 \times 4 = 140$ 不被35筛，被70筛； $35 \times 6 = 210$ 被105筛，所以只考虑质数倍数
- $35 \times 7 = 245$ 也不被35筛，因为 $245 = 49 \times 5$ ，所以如果 n 被质数整除就停下
- 验证一下：49筛掉 **49×2** 、 **49×3** 、 **49×5** 、 **49×7** ，而 $49 \times 11 = 539 = 77 \times 7$
- 上式中的7，是539的最小质因子，这条规律的正反两种表述终于合龙了

线性筛

- 回头再看下面这组数就无比清晰了：

- 2筛掉4

- 3筛掉6、9

- 4筛掉8

- 5筛掉10、15、25

- 6筛掉12

- 11筛掉22、33、55、77、121



- 这个线性筛质数的办法是伟大的数学家欧拉最早提出来的，因此也叫欧拉筛

Euler

```
for(int i = 2; i <= n; ++i)
{
    if (a[i]) p[++cnt] = i; //若 i是质数就放入 p数组, cnt是放入质数个数
    for (int j = 1; i * p[j] <= n && j <= cnt; ++j) // 在 p数组里扫一遍
    {
        a[i*p[j]] = false; //把 i的质数倍逐一筛掉
        if (i % p[j] == 0) break; //一旦 i被 p中的质数整除, 就退出讨论 i
    }
}

for(int i = 1; i <= cnt; ++i)
    printf("%d ", p[i]); //最后 p数组里的就是全部放入的质数
```

效率对比

- 测试数据规模5,000,000

试题	选手					
名称	排名	prime	总分	总用时(s)	测试时间	
欧拉筛法	1	100	100	0.578	2019/1/19 18:58:42	
埃氏筛法	1	100	100	0.625	2018/12/20 20:43:09	
定义判定法	1	100	100	3.312	2018/12/20 20:43:17	

课后练习

- luogu 1028 数的计算（递归得25分）
- luogu 1138 第k小整数
- luogu 1427 小鱼的数字游戏
- luogu 3383 线性筛
- luogu 3152 正整数序列