A - Sum the Sequence!!!

难度	知识点
2	数学、浮点运算

题目分析

直接根据课件中的例子，利用循环按项计算该极限值即可。

下面我们从一个较粗浅的层面论证需要计算的项数 N 随精度要求 ε 减小的增长情况。

考虑将数列 $\left\{\dfrac{1}{i^3}\right\}$ 分组：

i_L	i_R	$L=i_R-i_L+1$	$\max\limits_{i_L\le i\le i_R}\dfrac{1}{i^3}$
1	9	9	1
10	99	90	10^{-3}
100	999	900	10^{-6}
$\vdots$	$\vdots$	$\vdots$	$\vdots$
10^S	$10^{S+1}-1$	9×10^S	10^{-3S}

假设我们计算到数列的第 10^3 项。设误差为 u。下面估计 u 的大小。

$$
\begin{aligned}
u &= \left(\lim_{n\to\infty}\sum_{i=1}^{n}\frac{1}{i^3}\right)-\sum_{i=1}^{1000}\frac{1}{i^3}\\
&= \lim_{n\to\infty}\sum_{i=1001}^{n}\frac{1}{i^3}\\
&= \lim_{S\to\infty}\left(\sum_{i=1001}^{10000}\frac{1}{i^3}+\sum_{i=10^4+1}^{10^5}\frac{1}{i^3}+\sum_{i=10^5+1}^{10^6}\frac{1}{i^3}+\cdots+\sum_{i=10^{S-1}+1}^{10^S}\frac{1}{i^3}\right)\\
&\leqslant \lim_{S\to\infty}\left(9000\times 10^{-9}+9\times 10^4\times 10^{-12}+9\times 10^5\times 10^{-15}+\cdots+9\times 10^{S-1}\times 10^{-3(S-1)}\right)\\
&\leqslant \lim_{S\to\infty}\left(9\times 10^{-6}+9\times 10^{-8}+9\times 10^{-10}+\cdots+9\times 10^{-2(S-1)}\right)\\
&< \lim_{S\to\infty}\left(9\times 10^{-6}+9\times 10^{-8}+9\times 10^{-10}+\cdots+1\times 10^{-2(S-2)}\right)\\
&\vdots\\
&< \lim_{S\to\infty}\left(9\times 10^{-6}+9\times 10^{-8}+1\times 10^{-8}\right)\\
&= \lim_{S\to\infty}\left(9\times 10^{-6}+1\times 10^{-7}\right)\\
&< \lim_{S\to\infty}\left(9\times 10^{-6}+1\times 10^{-6}\right)\\
&= 1\times 10^{-5}
\end{aligned}
$$

类似上面推导过程可以得到，以前 N 项和代替极限值的误差 u 满足：

$$u \leqslant \frac{10}{N^2}$$

因此，当我们计算的项数超过 10^4 时，可以保证误差小于 10^{-7}。

但由于四舍五入的需要，我们还需要 10^{-8} 这一位的信息，因此计算的项数在 10^5 左右为宜。

示例代码

示例代码 1

```c
#include <stdio.h>

int main()
{
    int n = 100000, i;
    double a = 0.0;

    for (i = 1; i <= n; ++i)
    {
        a += 1.0 / i / i / i;
    }
    printf("%.7f", a);

    return 0;
}
```

示例代码 2

```c
#include <stdio.h>

int main()
{
    puts("1.2020569");

    return 0;
}
```

常见错因

- 计算项数不够，导致精度不满足要求。
- 使用 `1 / (i * i * i)` 之类的写法，其中 `i` 是 `int` 型变量，导致相乘时产生溢出。

B - math.h

难度	知识点
1	math.h头文件

题目分析

根据题目描述，使用相应的公式即可。

示例程序

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

int main()
{
    double x, y, z;

    scanf("%lf%lf%lf", &x, &y, &z);
    printf("%.2lf", pow(acos(sin(x)), log(1 + fabs(sinh(y)))) / (cos(z) + 2));

    return 0;
}
```

C - 火把

难度	考点
2	数学计算

题目分析

题目中一共有两种交易：

- 用 1 根木棍换取 x 根木棍
- 用 y 根木棍换取 1 块煤炭

针对第一种交易，需要求出总共需要换取木棍数，制作火把需要 k 根，换取煤炭需要 $k \times y$ 根，因为开始时有一根，那么需要换取的总数 $sum = k + k \times y - 1$ 。因为每次换取实际增加的木棍数是 $x - 1$ 根，所以第一种交易需要的次数是 $sum/(x-1)$ 向上取整，C语言的一种表示方法如下：$(sum - 1 + x - 1)/(x - 1)$ 。

针对第二种交易，制作的火把数就是需要交易的次数，因此第二种交易需要的次数就是 k 。

另外这道题的数据范围很大，需要使用 $long\ long$ 型整数。

示例程序

```c
#include <stdio.h>

int main()
{
    int i, t;
    long long x, y, k;
    long long sum, ans;

    scanf("%d", &t);
    for (i = 0; i < t; i++)
    {
        scanf("%lld%lld%lld", &x, &y, &k);      //读入数据
        sum = k + k * y - 1;                    //求出第一种交易需要换取的木棍总数
        ans = (sum - 1 + x - 1) / (x - 1) + k; //答案为 sum/(x - 1)向上取整并加上k
        printf("%lld\n", ans);
    }

    return 0;
}
```

D - 木桶

难度	考点
3	数据类型

题目分析

由于只能切一次木板，故考虑将最长的木板和最短的木板切、拼成一样长，设它们的初始长度分别为 max, min，则切割拼接后长度为 $(max + min)/2$。将该长度与第二短的木板比较输出即可。

注意，第二短的木板不一定和最短的木板长度不同。

示例代码

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

int main()
{
    int i, n, r, mx = -1, mn = 1e9, mn2 = 1e9, a;
    double height, ans;

    scanf("%d%d", &n, &r);
    for (i = 0; i < n; i++)
    {
        scanf("%d", &a);
        if (a > mx) mx = a;
        if (a < mn)
        {
            mn2 = mn;
            mn = a;
        }
        else if (a < mn2)
        {
            mn2 = a;
        }
    }

    height = 1.0 * mn2;
    if ((mn + mx) / 2.0 < height)
        height = (mn + mx) / 2.0;
    ans = acos(-1) * r * r * height;

    printf("%.4f", ans);

    return 0;
}
```

E - 壁画

难度	考点
3	模拟、数据类型

题目分析

本题是一个求和问题的简单变形，需要注意的是两段壁画可能会有重叠部分，要对边界进行判断后再计算。另外还需要注意题目的数据范围，并选用相应的数据类型。

示例代码

```c
#include <stdio.h>
int main()
{
    long long L[200], R[200], sum;
    int n, i;

    scanf("%d", &n);
    scanf("%lld%lld", &L[1], &R[1]);
    sum = R[1] - L[1];
    for (i = 2; i <= n; i++)
    {
        scanf("%lld%lld", &L[i], &R[i]);
        if (L[i] < R[i - 1])
            sum += R[i] - R[i - 1];
        else
            sum += R[i] - L[i];
    }
    printf("%lld", sum);

    return 0;
}
```

F - 百团大战

难度	考点
3	模拟、数据类型

题目分析

本题主要考察在读入数据的同时处理数据的能力。

考虑内存限制与数据规模，本题**不能**使用数组求解，故需要在读入的过程中维护变量，分别保存“热闹摊位”的数量与“热闹摊位”前的总人数，同时在每次读入数据后判断是否有新的“热闹摊位”，并实时更新维护的变量。

需要注意的有以下两点：

- 考虑到数据规模与数据范围，本题应该使用 long long int 类型变量存储人数；
- 在本题中，摊位是首位相接的，也就是需要额外判断第一个和最后一个摊位是否满足条件。

示例代码

```
#include <stdio.h>

int main()
{
    int i, n, num = 0;
    //变量 i 是临时变量，在循环中代表摊位的序号；n 变量表示总摊位数量，num 变量则存储“热闹摊位的数量”。
    int left, right, now;
    //left, right, now 分别代表左、右、中的摊位人数，从而判断 now 对应的摊位是否是“热闹摊位”。
    int i1, i2, in_1, in;
    //分别存储第一个、第二个、倒数第二个、倒数第一个摊位的人数。
    long long int sum = 0;

    scanf("%d", &n);
    scanf("%d %d", &i1, &i2);
    now = i1;
    right = i2;
    for (i = 0; i < n - 2; i++)
    {
        //事先读入了两个摊位的人数，故循环条件是 i < n - 2。
        left = now, now = right;
        scanf("%d", &right);
        if (now > left && now > right)
        {
            sum += now;
            num += 1;
        }
    }
    in = right;
    in_1 = now;
    if (in > now && in > i1)
    {
```

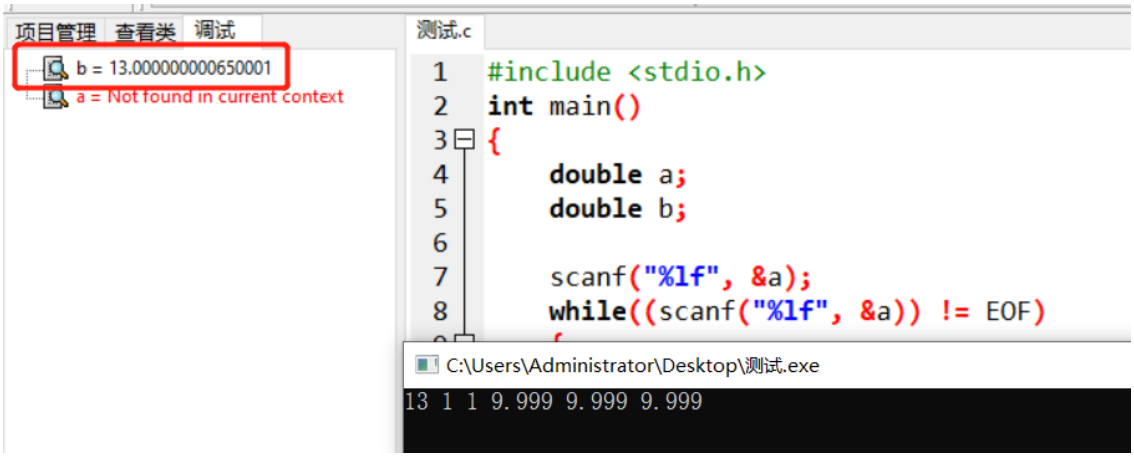
```
        sum += in;
        num += 1;
    }
    if (i1 > in && i1 > i2)
    {
        sum += i1;
        num += 1;
    }
    printf("%d %lld", num, sum);

    return 0;
}
```

G-简单计算器

难度	考点
3	输入、输出、数据类型

题目分析

1. 对于输入部分，题目指出“所有数和运算符之间都由至少一个空格隔开”，所以在进行数据读入的时候，需要处理掉若干个空格，这里可以利用`scanf(" %c",…)`来实现，这里的空格会匹配若干个不可见字符。
2. 对于输出部分，题目要求“保留 9 位小数的运算结果、输出最小域宽为 14 个字符、左边如有空余部分用0补齐”，左边如有空余用0补齐，那么应该使用右对齐输出，这里可以使用printf的格式化输出来实现，格式化输出可采用`“%0x.yf"`，0表示用0补齐，不输入则用空格补齐，x表示最小域宽，y表示保留的小数位数。
3. 对于double类型的变量，输出是应使用%f而非%lf。
4. 题目中，前三个数为不大于10000的正整数，考虑其运算可能为乘法，有可能超出int的表示范围，故应该使用long long进行储存或者直接存入double类型的变量。
5. double类型的变量只能保证15-16位有效数字绝对准确，对于本题，实际上可能存在超出有效数字的情况，仅因为题目说明”测试数据保证计算的所有中间结果不超出`double`的表示范围，所有计算过程不产生浮点异常“才可直接使用double进行储存和运算，否则应考虑使用其他数据类型（如long double）进行储存，或者将整数部分和小数部分分别进行储存减少double存储的数据的有效数字位数。 6.使用%lf直接读入整数其读入精度是编译器依赖的，在不同优化下其读入结果可能不同，或将存在一定误差。如下图所示：



实例代码

```c
#include <stdio.h>

int main()
{
    int getinInt, i;
    double getinFloat, ans;
    char op;

    for (i = 0; i < 6; i++)
    {
        if (i == 0)
        {
```

```
            scanf("%d", &getinInt);
            ans = getinInt; //直接将数据存入double类型的ans中
        }
        else
        {
            if (i < 3)
            {
                scanf(" %c%d", &op, &getinInt);
                //使用空格匹配若干个不可见字符进行读入，下同
                getinFloat = (double)getinInt;
                //将整型数据改为double类型储存，以备使用
            }
            else
                scanf(" %c%lf", &op, &getinFloat);

            //判断符号种类并进行相应运算
            if (op == '+')
                ans = ans + getinFloat;
            else if (op == '-')
                ans = ans - getinFloat;
            else if (op == '*')
                ans = ans * getinFloat;
            else
                ans = ans / getinFloat;
        }
    }
    printf("%014.9f", ans);
    return 0;
}
```

H - PRNG

难度	考点
4	数组查找、暴力计算

题目分析

本题按照题目给定的算法计算完成足够的次数即可，查找是否重复可以通过遍历数组的暴力方式实现。需要注意的是，尽管输入数据均在`int`范围内，但是计算结果可能超出`int`范围，因此需要使用`long long`类型。

示例代码

```c
#include <stdio.h>
int main()
{
    long long res[50005];
    long long m, a, c, seed, ans;
    int i, j, flag = 0;

    scanf("%lld %lld %lld %lld", &m, &a, &c, &seed);
    for (i = 0; i < 50000; i++)
    {
        ans = (seed * a + c) % m;
        res[i] = ans;
        for (j = 0; j < i; j++)
        {
            if (res[j] == ans)
            {
                flag = 1;
                break;
            }
        }
        if (flag == 1)
        {
            printf("Duplicate found\n%lld %d", ans, i + 1);
            return 0;
        }
        seed = ans;
    }

    printf("Not repeated\n%d", ans);

    return 0;
}
```

I - 十万以内的正整数除法

难度	考点
4	模拟、数组、字符串常量

题目分析

- 首先，判断 `y` 能否整除 `x` 是容易的：如果满足 $x\%y=0$ 则 `y` 能整除 `x`，答案就是 x/y，可以直接用 C 语言中的整除运算。
- 否则，需要计算小数：
  - 主要思路是模拟竖式除法，需要使用一个数组存储计算结果（如实例代码中的 `shang[i]` 存小数点后第 i 位的结果）。
  - 难点在于判断是否循环小数：
    1. 首先，容易想到，当余数为 0 时，计算结束，结果有限。
    2. 然后考虑在手写竖式除法的过程中什么情况可以让自己能够判断这是个循环小数。每一位商的结果由余数除以除数得到，余数变成模除数后的结果，那么可以想到：如果第 i 次运算时的被除数 x_{i-1} 在之前第 k （$k<i$）次运算时也出现过，那么商 $shang[i]=x_{i-1}/y$ 和当前运算后的余数 $x_i=x_{i-1}\%y$ 都与之前的第 k 次运算时的情况相同，即 `shang[i]==shang[k]`，$x_i==x_k$。
    3. 然后，x_i 和 x_k 又分别是第 i+1 次和第 k+1 次运算时的被除数，重复第二步的判断，即这一位的商和余数也分别相等。
    4. 因此，判断这是个循环小数的一个充要条件是：当前计算出的余数在之前出现过。而且，第一个被发现重复的余数所在的位置即每组循环节的起始位置，两个重复余数所在位置之差即为循环节长度。
- 也许你注意到了实例代码中记录结果的数组大小为 N，N是一个比题面所给的 y 的上界大一点的数字。
  1. 由计算小数的过程可以看出：计算过程中出现的每个不同的余数只对应着唯一一种情况（商和下一次运算的被除数）。也就是说，假如可能的余数有 tot 个，则 $x\div y$ 按本题所要求输出的结果长度将是 $tot+C$。（C 是一个常数，是要输出的整数部分和小数点、中括号、下划线等非数字符号的个数）
  2. 又因为 $x\%y\in[0,y-1]$，即可能的余数的个数必然不超过 y，所以不用担心结果的长度可能不小于 N 而发生数组越界。
  3. 与 “367个同学中必然出现重复的生日”的思想类似，本题中是“ 计算第 y 位结果时必然已经出现重复的余数”。
- 实例代码中的 `where[num]` 的含义是：数值 x 在第几次运算时第一次作为被除数出现。用这样一个记录位置的数组，可以简化“寻找 x 之前是否出现过以及第一次出现的位置” 这个操作，不需要再循环遍历一次之前算出来的余数。
- 利用字符串常量如 `const char a[7] = "integer";` 可以起到简化代码的效果。

易错点

- 数组开小了，导致提交到 OJ 上后反馈 REG。
- 在 main 函数里开了过大的数组，导致本地运行时反馈运行错误，无法输入数据。
- 用 `(y%5==0||y%2==0)` 判断是有限小数，反例如 $1\div 6$。
- 循环小数包括纯循环小数和混循环小数，有的 WA 是因为忽略了混循环小数，其循环节的起始位置并非小数点后第一位，例如 $1\div 15$。

- “用一个数组依次记录下出现过的余数，然后每次计算时遍历一遍该数组寻找是否有当前余数的值”的做法会导致某些数据点 TLE 。更快的做法是直接把余数的数值当作数组下标。
- 如果发现自己的代码没有明显的数组越界和运算次数过大，一些 REG 和 TLE 结果可能是由于代码中有无法跳出的循环结构。可通过调试查看循环变量的变化情况、手动模拟循环运行的边界等方法解决此问题。

示例代码

```c
#include <stdio.h>
#define N 100005

int main()
{
    int shang[N], where[N], x, y, i, j;
    const char a[7] = "integer";
    const char b[20] = "terminating decimal";
    const char c[20] = "recurring decimal";

    scanf("%d%d", &x, &y);
    if (x % y == 0)
    {
        printf("%s\n", a);
        printf("%d", x / y);
    }
    else
    {
        shang[0] = x / y;
        x %= y;
        for (i = 1; (!where[x]) && x; i++)
        {
            where[x] = i;
            shang[i] = 10 * x / y;
            x = 10 * x % y;
        }
        if (!x)
        {
            printf("%s\n%d.", b, shang[0]);
            for (j = 1; j < i; j++)
                printf("%d", shang[j]);
        }
        else
        {
            printf("%s\n%d.", c, shang[0]);
            for (j = 1; j < where[x]; j++)
                printf("%d", shang[j]);
            printf("_");
            for (; j < i; j++)
                printf("%d", shang[j]);
            printf(" [%d]", i - where[x]);
        }
    }

    return 0;
}
```

J - Long Long Factorial

难度	考点
4	数组、高精度

题目分析

本题主要让同学们了解简单的单精度乘高精度数的运算。主要思路是按类似于竖式乘法的方式模拟乘法，需要使用一个数组存储计算结果，其中低位存储计算结果的低位，并向高位延伸。首先我们不断循环计算$(i-1)!\times i$，对某一个待计算的$(i-1)!\times i$，设$(i-1)!$是被乘数，i是乘数。从被乘数的低位开始，按位依次乘以乘数，并加上之前的“进位”，每次所得结果的最后一位作为该位的计算结果，对10取余的结果作为“进位”参与下次运算，超过被乘数的最高位后再把“进位”依次向高位写出，即可完成一次计算。最后将结果数组从高位向低位依次输出即可

示例代码

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

int main()
{
    int i, j, n, temp, d = 1, carry; //temp为阶乘元素与临时结果的乘积，carry是进位，d是位数
    int a[30000];                    //确保数组足够大

    scanf("%d", &n);          //n的阶乘
    a[0] = 1;                 //先初始化为1，方便后面运算
    for (i = 2; i <= n; i++) //从2开始阶乘 ,每次循环计算i阶乘的结果
    {
        for (j = 1, carry = 0; j <= d; j++) // 每次循环初始化进位的值
        {
            temp = a[j - 1] * i + carry; //相应阶乘中的一项与当前所得临时结果的某位相乘加上进位
            a[j - 1] = temp % 10;        //更新临时结果的位上信息
            carry = temp / 10;           //如果有进位就进入下面的循环
        }
        while (carry) //如果有进位
        {
            ++d;                    //增加进位，位数加一
            a[d - 1] = carry % 10; //给新的进位赋值
            carry = carry / 10;    //看还可不可以再进位
        }
    }
    for (j = d - 1; j >= 0; j--) //从高位向低位依次输出
        printf("%d", a[j]);
    printf("\n");

    return 0;
}

```