从G 浮点取模 看待 浮点误差

原题地址: https://buaacoding.cn/problem/1572/index

出这道题的本意，就是让大家对浮点误差有一个极其彻底的认识。只有**了解**它，在设计的时候才不会畏惧它，对它操纵自如。

类似的还有数据类型的溢出问题。这种问题并不是逼迫大家什么数据都用long long。而是提醒大家要了解各种类型的计算范围，根据实际情况选择合适的，不分场合地使用long long只会造成代码臃肿，效率低下。

对于浮点误差，我们首先要看到它的根源。以下是我在“蓝笔与黑笔”那题中写的详细的Hint

> 浮点数的存储方式，无论**float**还是double，计算机里面都是分成两个部分的，一部分是底数部分，代表小数去掉小数点之后的整数数值A。
> 一部分是指数（移码），代表乘上的幂次B，于是小数可以表示成A*2^(-B)。
>
> 为什么乘上2的幂呢，因为计算机只认二进制。
> 当然啦，这已经是简化过的模型啦，更详细的跟本题无关，请移步百度。
>
> 很容易发现，假如一个小数的小数部分不能表示成Σ1/(2^k)的形式，换句话说，不存在一个2的整数幂，使得乘上小数部分后成为整数，那么计算机是无法存储它的，可以证明，这是二进制下无限循环小数的情况。
> 另一种情况是小数虽然能够表示成有限二进制小数的形式，但是由于内存限制，无法达到所需的精度。
>
> 举例：2.3125 可以完全正确地表示，因为0.3125=5/16。而3.72则不能。
>
> 当计算机赋值变量a=3.72时，很可能a的实际值时3.71999999或者3.720000001
>
> 出现了误差……
>
> 浮点误差的另一个来源是计算过程，例如1.0/3*3，原来完美的1.0就会变成一个和1.0稍微有误差的小数，而计算过程越复杂，往往误差越大。
>
> 因此在判断浮点数大于，小于或者相等时，Coder们往往会设置一个极小量eps，当两个浮点数之差的绝对值小于这个eps时，我们认为两者相等，这样的话1.0/3*3就可以被认定与1.0相等。

再补充一句，double类型的表示方式是51bit的底数和13位的指数码，基本上，14位有效数字的浮点类型，都能以**极其小以至于可以忽略不计**的误差存入double，当然**误差**是客观存在的，**我们实际上担心的不是误差本身，而是一不小心误差会被放大。**例如解浮点数二次方程式，delta=0和delta=-0.000000000001，就是一个根和无根的分别（误差被放大了）。

首先我们看几个本题的非完美做法

```
#include<stdio.h>

int main()
{
    double a,b;
    scanf("%lf%lf",&a,&b);
    printf("%.4lf",(int)(a*10000)%(int)(b*10000)/10000.0);
    return 0;
}
```

当3.2419 0.0163这组数据时，3.2419已经不能被一个double完美存储了，有的同学一定会很惊讶，才四位小数double就不行了吗，不是说double厉害到可以连10^100都能存储吗？但事实上，这种误差无论double存储能力多强依旧存在，计算机是二进制的，3.2419将会以一个误差极小的二进制法表示，**而这种误差在取int时被放大了**，3.2419实际上是3.24189999999999999999，乘以10000，取int，变成了3418，而不是我们想要的3419。但这个方法稍加改正就完全正确了——把取整方式从向下取整修正为四舍五入取整（见下文）。

```
#include<stdio.h>

double a,b;

int main()
{
    scanf("%lf%lf",&a,&b);
    while(a>b)
        a-=b;
    printf("%.4lf",a);
    return 0;
}
```

类似的也是利用浮点误差，这次主要是计算误差，在a连续减去多次b的时候，误差会被放大，例如10000.0000，0.0001

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

double a,b;

int main()
{
    scanf("%lf%lf",&a,&b);
    printf("%.4lf",fmod(a,b));
    return 0;
}
```

fmod不用说了，误差很大，题干里已经提示会WA了，即使使用long double和fmodl，结果也一样。

```
#include<stdio.h>

int main()
{
    double a, b;
    int x;
    scanf("%lf %lf", &a, &b);
    a = a/b;
    a = a - (int)a;
    printf("%.4lf", a*b);
}
```

这个做法我放过了，是能够AC的，但也不完美，误差在int(a/b)z中，向下取整把误差放大了。例如9999.0003 0.0001

我怕再卡下去，AC率太低没法交代，就没有刻意造数据，单独对这种做法网开一面了。

下面是两种正确做法，第一种是官方解法。大家应该熟悉了，用两个int来表示一个浮点数

```
#include<stdio.h>

int main()
{
    int a,b,c,d;
    scanf("%d.%d %d.%d",&a,&b,&c,&d);
    a=a*10000+b,c=c*10000+d;
    printf("%.4lf",a%c/10000.0);
    return 0;
}
```

后来在大家的代码之中发现了**更加优美的做法**

```
#include<stdio.h>

int main()
{
    double a,b;
    scanf("%lf%lf",&a,&b);
    printf("%.4lf",(int)(a*10000+0.5)%(int)(b*10000+0.5)/10000.0);
    return 0;
}
```

这种方法是在第一段代码中，分别加了0.5再用int取整，效果就是四舍五入。我们知道double类型的误差是非常小的，四位小数乘以10000之后，尾数更接近哪个整数，那么就取哪个整数

总体来说两种方法各有千秋。请大家细味

浮点误差并不是为了坑大家，（毕竟坑大家助教也不会获得欢乐）。事实上善于利用浮点误差可以成为解决问题的有力工具，比如计算一个[0,1]区间中一个连续函数的定积分值，用黎曼的划分法，普通人可能会选择分成100000等份再相加，但一旦了解浮点误差后，划分成131072=2^17等分才是更好的选择（想一想，为什么？）