# Lab1实验报告

2019.12.16
181240035
刘春旭

> 直接`return (a*b)%m`不仅会出现截断低64位后，最高位是1引发的UB行为；
>
> 还有可能出现先做乘法保留`a*b`后，再进行`mod`操作引发的结果错误问题；

## 1. 实现multimod

1. 所以为了解决上面出现的问题，那我们肯定不能先做截断再处理模运算；故我们要现保存128位的乘法结果，再用这个结果去做模运算；

   使用这样的代码：

```
//使用两个70位的int型数组把A和B的每一位都储存起来
int A[70]={};
int B[70]={};
int C[140]={};
while (a > 0)
{
  A[la] = a%10; //保存A的十进制的每一位
  a /=10; //a除10，表示
  la++; //la表示a的十进制位数
}
//B也做相同操作
//...
//把A*B的每一位都分别相乘，加到C的对应位上（i+j+1是为了后面进位操作方便）
for (int i = 0; i < la; i++)
  for (int j = 0; j< lb; j++)
    C[i+j+1] += A[i] * B[j];
//统一处理进位;
for (int i = 1; i <= la+lb; i++)
{
  C[i-1] = C[i] % 10;
  C[i+1] += C[i] / 10;
}
//然后应该继续拿C来操作一个高精度除法，但是当时忘记了
```

   所以在胡乱拿了一个`uint64_t`格式的`result`暂且写了一个部分正确的代码后进行测试，测试数据只通过了106087个，也就刚刚超过1/10，所以说这个溢出的错误率还是很高的：

```
chauncey@debian:~/ics-workbench/multimod$ gcc p1.c -o tst1
chauncey@debian:~/ics-workbench/multimod$ ./tst1
testing
Success! Time: 1.527574, correct: 106087
```

那么先去看任务2吧！于是我们就从任务2中吸取了灵感：

> $$\begin{aligned} c &= c_0 \cdot 10^0 + c_1 \cdot 10^1 + \ldots + c_{lc-1} \cdot 10^{lc-1} \\ &= (\ldots((c_{lc-1} \cdot 10) + c_{lc-2}) \cdot 10 + \cdots + c_0) \end{aligned}$$

那么我们只要对已经求出来的c的每一位都乘10、取模，然后慢慢加起来就好了；

但是这里溢出的风险很大，所以不能够轻易乘10，要进行如下操作：

```
uint64_t result = 0;
    uint64_t ans2x = 0;
    uint64_t ans4x = 0;
    uint64_t ans8x = 0;
    uint64_t ans10x = 0;
    for(int i = lc;i >= 0;i--)
    {
      ans2x = (uint64_t)(result<<1)%(uint64_t)m;
      ans4x = (uint64_t)(ans2x<<1)%(uint64_t)m;
      ans8x = (uint64_t)(ans4x<<1)%(uint64_t)m;
      ans10x = (uint64_t)(ans8x+ans2x)%(uint64_t)m;
      result = (uint64_t)(ans10x%m + C[i]%m) %(uint64_t)m;
    }
    return (int64_t)result;
```

我们最多只能保证乘二的时候，在`uint64_t`的意义下，是不会溢出的；所以每次进行乘2操作，我们都要模一次$m$，最终通过这样的操作得到一个10倍的$result$，于是结果正确；

```
chauncey@debian:~/ics-workbench/multimod$ gcc -O3 p1.c -o tst1 && ./tst1
testing
Success! Time: 2.297555, correct: 1000000
```

# 2：性能优化

### 2.1 优化代码

首先，我们有$a \cdot b (mod\ n) = a \cdot (b\ mod\ n)$， 所以结合任务二中给出的提示，可以写出：

> $$a \cdot b = (a_0 \cdot 2^0 + a_1 \cdot 2^1 + \ldots + a_{62} \cdot 2^{62}) \cdot b = (\ldots((a_{62} \cdot b) \cdot 2 + a_{61} \cdot b) \cdot 2 + a_{60} \cdot b) \cdot 2 + \ldots + a_0 \cdot b)$$

其中a的二进制每一位与b相乘都不会溢出，因为要么等于b本身，要么等于0；然后用这个结果乘以2也不会溢出，因为b最高62位，左移一位也是不会溢出的（在`uint64_t`意义下）；当然，左移之后要进行模m的操作，否则肯定还是有溢出的可能性的。

> 刘某人我一开始就是因为左移之后没模$m$，郁闷了一整天，对于已有的100w组数据，测试通过率始终在87%，实属智障。

模完之后数据就可以继续保持在最长63位，然后再加上下一位，这也可以保证不会溢出；以此类推，我们就可以得到正确的结果。

代码实现：

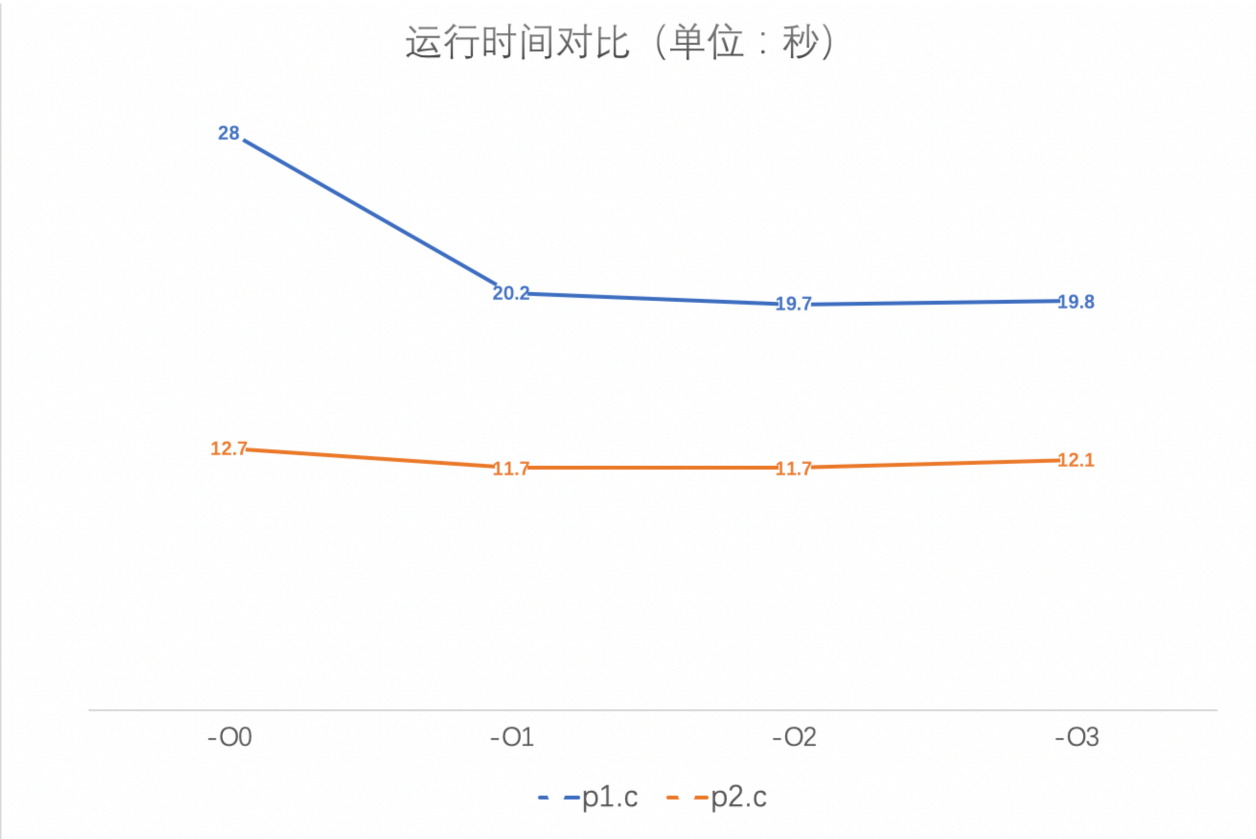
```
    b = (uint64_t)b % (uint64_t)m;
    for (int i = la; i >= 0; i--)
    {
      result  = (uint64_t)((result << 1)%m + (uint64_t)(A[i]*b)) %m;
    }
```

测试结果：

```
chauncey@debian:~/ics-workbench/multimod$ gcc p2.c -O3 -o tst2
chauncey@debian:~/ics-workbench/multimod$ ./tst2
testing
Success! Time: 1.152671, correct: 1000000
```

**2.2 运行时间测试**

如下图所示：（控制变量：根据1000w组相同表达式的测试结果，每组都重复运行三次，取平均结果）




可以看出，编译器优化的程度对于p1.c是比较可观的，但对于p2.c来说提升比较少；

结论1：优化的结果与源程序的代码效率有关；

结论2：随着优化程度的提高，程序运行速度提升越来越慢，甚至有可能时间会增加；

# 3：解析神秘代码

```
int64_t multimod_fast(int64_t a, int64_t b, int64_t m) {
  int64_t t = (a * b - (int64_t)((double)a * b / m) * m) % m;
  return t < 0 ? t + m : t;
}
```

首先，我们可以知道，`int64_t`类型，精度64位，所以减号之前的`a*b`一旦溢出，即超过64位，就无法保证结果的正确性了；`double`类型，尾数部分的精度是52位，加上前面的1可以达到53位的精度，如果超过53位，精度损失是一定的，所以我们的目的是来找出究竟在什么数据范围内，这份代码是可以正常工作的。

### 3.1 为什么能work

如果我们考虑任何运算都没有溢出的情况下，设`a * b = km +r`，其中k为任意整数，r为正整数，小于等于m，则有：

$$\begin{aligned} a \times b - ((a \times b)/m) \times m &= a \times b - k \times m \\ &= r \end{aligned}$$

所以我们就能算出来余数是多少了

### 3.2 我爱测试

先用之前的数据测试一下：

```
chauncey@debian:~/ics-workbench/multimod$ gcc p3.c -O3 -o tst3
chauncey@debian:~/ics-workbench/multimod$ ./tst3
testing
Success! Time: 0.034464, correct: 25227
```

对于之前用来测试的100w组数据，它只通过了25227组，不过速度是真的很快；

那么我们来生成一些其他代码吧，首先，无论怎么变，这份代码首先把a和b乘积除以m的结果转化成了`double`类型，精度肯定不能超过53位，所以我们可以让生成的a和b除m的结果不超过$2^{53}-1$即可，我们只让数据生成在这个范围内（用一个值`pivot`来监测生成数据的大小），进行测试：

```
    pivot = (a * b) / m
    if pivot < 2**52:
        f.write (str(a)+' '+str(b)+ ' '+str(m)+ ' ' +str(ans) +'\n')
```

结果还是有问题，错了3组：

```
chauncey@debian:~/ics-workbench/multimod$ ./gen_expr.py && gcc p3.c -O3 -o
tst3 && ./tst3
testing
wrong at: ( 652122142582493499 * 45618283202733028 ) mod 8023511218050142028 =
3618978868770762724
p3.c ans : 1219257231161495164
wrong at: ( 7319812263879761 * 5010698660403933449 ) mod 9097728872760435486 =
777764917033521319
p3.c ans : 526478588844840675
wrong at: ( 606696137163777057 * 62241478240348131 ) mod 8761014162920238577 =
998675595101173181
p3.c ans : 73959847232098719
Success! Time: 0.000062, correct: 2172 / 2175
```

那这是为什么呢？短暂思考之后，意识到；

> double类型能够表示的范围肯定足够$2^{128}-1$，但是精度只能表示前面的52位，后面的精度全部损失掉了；那么如果`(double)a*b`的结果大于某个数，以至于除以`m`之后也不足以弥补丢失的精度；比如说，如果`(double)a*b`的结果有127bit，生成的`m`却只有30bit的话，那么$127-53-30=44$个bit的精度就丢失掉了；

现在我们对上面这个可能的解释进行探究，对测试代码进行一波更新，来比较`(double)a * b / m)`的精度究竟有没有损失：

```
if(c != result){
        correct--;
        printf("wrong at: ( %ld * %ld ) mod %ld = %ld\n", a,b,m,c);
        printf("p3.c ans : %ld\n", result);
        printf("pivot: %ld\n",pivot);
        printf("mypivot: %ld\n", (int64_t)((double)a * b / m));
      } //错误情况
```

而正确情况只输出`pivot`和`mypivot`

从运行结果中发现：

```
#正确情况（均相同）：
pivot: 2048536579440630
mypivot: 2048536579440630
pivot: 4374839491679616
mypivot: 4374839491679616
pivot: 1045722654631183
mypivot: 1045722654631183
...
#错误情况（均有精度损失）：
wrong at: ( 15652754725681194 * 1761569793468815461 ) mod 6148972687033702917
= 3774957025974190377
p3.c ans : 3775131013365747512
pivot: 4484231970566421
```

```
mypivot: 4484231970566420
...
```

可见上面的推理是正确的

### 3.3 寻找正确范围

根据上面的推导，我们必须保证`a*b/m`的精度不能有损失：

m最少取到1位的精度，则a*b最多取到53+1=54位精度；

这样才能保证结果万无一失，即，既考虑了`double`类型的精度，又考虑了`a*b/m`的精度不能有损失；

### 3.4 结论（吐槽）

那按照这个结果来看，这份代码的正确率还不如直接`int64_t a*b %m`呢，毕竟这样做至少还能保证`a*b`的精度最多63位；当然，如果限定了m的大小比较大的话，这份代码还是有其用武之地的：

结论：设m的精度位数为`sizeof(m)`，那么能使这份代码保持正确的`a*b`精度范围为：`53+sizeof(m)`

## 4：谈谈如何测试

### 4.1 用python 生成数据代码：

```
#!/usr/bin/env python3 //可以让*.py在终端里直接执行的神秘代码
import random
MAX = 9223372036854775807 //2^63 -1
for i in range(1,1000000): //生成100w组数据
    a = random.randint(0,MAX)
    b = random.randint(0,MAX)
    m = random.randint(1,MAX)
    ans = (a * b) % m
    f.write (str(a)+' '+str(b)+ ' '+str(m)+ ' ' +str(ans) +'\n')
```

然后在终端里cd到文件的位置 ，然后给`*.py`加上权限：

```
$chmod a+x gen_expr.py
$./gen_expr.py
```

然后就可以在终端里收获100w组测试数据啦

### 4.2 在c文件中进行测试

```
FILE *fp = fopen("test.txt","r"); //这是已经用python生成的100w组正确数据
  struct timeval t1, t2;
  double _time; //为了使用gettimeofday()函数获取时间
  if(NULL==fp) printf("Can not find file\n");
  else
```

```
  { printf("testing\n");
    while (fgets(tmp, 256, fp)) //如果还能得到某行的数据，那么说明还没测试完
    {
      num = strtok(tmp,delim);  sscanf(num, "%ld", &a); //得到a
      num = strtok(NULL,delim); sscanf(num, "%ld", &b); //b
      num = strtok(NULL,delim); sscanf(num, "%ld", &m); //m
      num = strtok(NULL,delim); sscanf(num, "%ld", &c); //结果c
      memset(tmp,0,sizeof(tmp)); //将tmp置零，为取下一行的数据做准备
      gettimeofday(&t1, NULL); //在进行模运算之前得到一个时间t1
      result = multimod_p2(a,b,m);
      gettimeofday(&t2,NULL);//运算之后得到一个时间t2
      cnt++; //计数器++
      if(c != result) cnt--; //结果若错了，则计数器--
      _time += t2.tv_sec - t1.tv_sec+(t2.tv_usec - t1.tv_usec)/1000000.0; //得
到时间
    }
    printf("Success! Time: %lf, correct: %d\n",_time, cnt); //输出时间和正确组数
  }
  fclose(fp); //fclose是美德
```