Lab1 实验报告

谢逸 171240536

任务 1：

p1.c 的主要思路：

运用大数（这里是用 int 数组表示的），把所有可能溢出的地方的运算都改为大数运算。乘法就使用大数乘法，取模就用大数的除法，只是其中用大数减法以防溢出。

（小学生都会--jyy）

任务一代码：（和任务二一起测试了）

p1.c

```c
#include<stdio.h>
#include<stdint.h>
#include<string.h>
#include<stdlib.h>
void mul(int *a, int *b, int a_l, int b_l, int *c);
void minus(int *a, int *b);
int notsmaller(int *a, int *b);
int64_t mod(int *c, int *m);
int64_t multimod(int64_t a, int64_t b, int64_t m);
int64_t multimod(int64_t a1, int64_t b1, int64_t m1){
    int a[20] = {0}, b[20] = {0}, m[40] = {0}, c[40] = {0};
    memset(a, 0, sizeof(a));
    memset(b, 0, sizeof(b));
    memset(c, 0, sizeof(c));
    memset(m, 0, sizeof(m));
    int a_l = 0, b_l = 0, m_l = 0;
    while(a1 > 0){
        a[a_l] = a1 % 10;
        a_l ++;
        a1 /= 10;
    }
    while(b1 > 0){
        b[b_l] = b1 % 10;
        b_l ++;
        b1 /= 10;
    }
    while(m1 > 0){
        m[m_l] = m1 % 10;
        m_l ++;
        m1 /= 10;
    }
    mul(a, b, a_l, b_l, c);
   return mod(c, m);
}
void mul(int *a, int *b, int a_l, int b_l, int *c){
    for(int i = 0; i < a_l + b_l - 1; ++i){
        for(int j = 0; j <= i; ++j){
            if(a_l > j && b_l > i - j){
                c[i] += a[j] * b[i - j];
                if(c[i] > 9){
                    c[i + 1] += c[i] / 10;
                    c[i] %= 10;
                }
            }
        }
    }
}
int notsmaller(int *a, int *b){
    for(int i = 39; i >= 0; --i){
        if(i == 0 && a[0] == b[0])  return 1;
        if(a[i] > b[i])  return 1;
        else if(a[i] < b[i])  return 0;
    }
}
int64_t mod(int *c, int *m){
    int y[40] = {0};
    for (int i = 39; i >= 0; --i) {
        for (int j = 39; j > 0; --j) {
```

```
            y[j] = y[j - 1];
        }
        y[0] = c[i];
        while(notsmaller(y, m)){
            minus(y, m);
        }
    }
    int64_t ans = 0;
    int64_t t = 1;
    for (int i = 0; i < 39; ++i) {
        ans += t * y[i];
        t *= 10;
    }
    return ans;
}
void minus(int *a, int *b){
    for (int i = 0; i < 40; ++i) {
        if(a[i] < b[i]){
            a[i + 1] --;
            a[i] += 10;
        }
        a[i] -= b[i];
    }
}
```

（测试在下面）

任务二：

任务二思路：

可以发现 a 和 b 和 m 都是小于 $2^{63}-1$ 的，如果转化为无符号数，乘以二仍然不会溢出。而且有 $(a+b+c)\%m=((a+b)\%m+c)\%m$。根据这两点，我们采用的二进制的运算，先用数组 b1 存下 b 的二进制表示的每一位，用 64 位无符号数组 a_mod_m 来存(a<<i) % m 的值（过程中不会溢出，因为数组中每一个数都小于 2^{63}，乘以二之后仍然不会溢出）。要计算余数，只要将 a_mod_m[i]*b[i]累加，每次加法后都取模，以保证 ans 小于 2^{63}，因此此处加法永不溢出，累加结束后的 ans 就是要的答案。

任务二代码：

```c
#include<stdio.h>
#include<string.h>
#include<stdlib.h>
int64_t multimod(int64_t a, int64_t b, int64_t m);
int64_t multimod(int64_t a, int64_t b, int64_t m){
    u_int64_t a_mod_m[64];
    int b1[64];
    for(int i = 0; i < 64; ++i){
        b1[i] = (b >> i) & 1;
    }
    a_mod_m[0] = a % m;
    for (int i = 1; i < 64; ++i) {//求(a*2^i)% m的值(必定小于2^63所以即使乘以2也不会溢出)
        a_mod_m[i] = (a_mod_m[i - 1] << 1) % m;
    }
    u_int64_t ans = 0;
    for(int i = 0;i < 64; ++i){
        if(b1[i] == 1){
            ans = (ans + a_mod_m[i]) % m;//ans和a_mod_m[i]必定小于2^63, 所以相加之后不会溢出
        }
    }
    return ans;
}
```

任务一和二的正确性验证与时间验证：

采用如图的 python 代码产生了 100 0000 组数据

```python
import random
f = open("test", "w")
for i in range(0, 1000000):
    a = random.randint(0, 9223372036854775807)
    b = random.randint(0, 9223372036854775807)
    m = random.randint(1, 9223372036854775807)
    ans = (a * b) % m
    print >> f, a, " ", b, " ", m, " ", ans
f.close()
```

测试用例（部分）如下图，由于数据量太大（1000000 组）不在此一一截图。

test

```
1506355601198606798   1419816297442167078   6037568184595365937   2917839382999200618
4522450663843695172   4345628981587593588   4306624374032205696   403245985707407952
7044780478772482662   295147401865239682   4889282303240959379   1342838644645726540
2642536015378835877   8402552108788267009   2307357931053031179   616953047529787899
4293924685689621483   2222356056194398569   2533113205882835779   954535739761085771
3691642565526336614   7598890963921472762   1685813819998981360   433493874744751468
5899480451983108458   7455221359018412442   7282178211429985842   1835712794868449580
8850107126635034100   7470686735065918715   6766225340499365245   715193009895776275
7158001001986637354   1655937261194113843   120238844330305671   103245103665688202
6444687750608384523   6150308408639009052   3208077830555639490   2191043623625422896
1358037198903827028   4533467853576726281   8816758052645655780   7593912024573796848
3003385156156270231   8223318820190693815   3409018238220878433   3023359707254216416
973773085359999949   2079182460809336487   693387922016118503   213421895547112486O
2937727616618751831   8869259842657682296   3638924700073467881   773073034768148 58
3252668521863865191   5203341489234173216   1508819818905804899   786960681657409 38
1846373563740957971   665554151492236130   652598890110056633 6   608127887927594221 4
4233142880238647473   5620538585414562690   1286220602311503862   417428326262599952
2205435258989901884   4963754554292851483   5725538669140477671   3109151596588296182
860493905162997495   859388045974710010 1   250100864922509449 8   42506870086856331 9
3728317079652265096   4311161296558553669   998205388832418 97   243799738291862 79
2669974638261103208   4983021672404828852   2580331743021112721   646590519452612061
276880383004747443   1273677672395504670   1523380046520546726   1115571933424377630
2342327217173926235   6545936161428230193   4351763887711884986   2942610339926664405
613451489444081161   3724181790746517671   8443547494763391236   4563732671540472103
499811607272005527   4446195557881081295   557135670905300645   550150348610092590
5941800810114523833   3502115431591324596   7261765328927384645   1626472152946082383
5188345091900890168   1800823082578413540   1766825126559937297   376051109215237762
257316571693808535   6037214236782757259   3382934791702443110   1421091889594891325
5607439927001771857   8050814180385792164   8224391561002305691   3290858981165872078
1673313429649329981   5622021617712969416   5089393670280468131   952565057852823011
7608198341868408895   3532396443676665230   1161958105448581073   919510265137486518
3403486672171332433   7391799386203761146   8879763628451723189   4177597522156621887
973561714642532302   536067126859005565   677911286348232676 3   491976148936456812 0
6623672336391948378   3898005610685182352   4974848041735124063   1592177764214073979
1598991343273809140   5672231219806636464   932223866886912642   219601491084278184
2889640614203874489   8297579837739197293   3695616067146835995   3257411590267757382
5663527814428112575   6915354254974263343   1452463209910624200   1150739953790188225
7021394626190565010   1735159850513695834   7157370601160020262   5862442804513176020
8507282976319549460   8562227052451081573   3669291204938712565   2544471752481115520
8099066146357829742   3831977584828240116   3010081808641804427   2496046620176857308
7175002732524244582   9006728023008716236   1593674615833616211   294605812350608509
3447020259489849857   639079139751408 3   3962912011514493904   1670260629667797555
3369528186236027468   7982514572518238492   629540167062675672   494425632285956968
7436794465554300197   4040495530190513245   117515274750050529   743253733265710 10
8217227971297238390   2689942689806867073   8231485035970244838   1531484341515933372
1275568269500772147   3612288781808741767   6888694950411810340   3792442678627264449
6877644831481011364   1737731816765029626   5289304560248775053   5018031570181093908
5122995029763167240   926827706012400771   544516520621955683   141333434081786728
```

通过如图的 main. c 和 p. h 文件测试：

```c
#include<stdio.h>
#include<time.h>
#include<stdint.h>
#include "p.h"

int main(int argc, char *argv[]) {
    int64_t a, b, m, ans, ans2;
    FILE *fp =NULL;
    if( (fp = fopen("./test", "r")) == NULL)
        printf("File cannot open.\n");
    clock_t start, finish;
    start = clock();
    while(~fscanf(fp, "%lld %lld %lld %lld", &a, &b, &m, &ans)){
        ans2 = multimod(a, b, m);
        if(ans != ans2)
            printf("The answer of %lld * %lld %% %lld should be %lld, but get %lld.\n", a, b, m, ans, ans2);
    }
    finish = clock();
    printf("time = %lf s\n",(double)(finish - start) / CLOCKS_PER_SEC );
    fclose(fp);
    return 0;
}
```

p.h

```c
int64_t multimod(int64_t, int64_t, int64_t);
```

得到结果（用不同的优化等级试了一下）：

```
xy@debian:~/lab1_left$ gcc -O0 main.c p1.c -o p1.out
xy@debian:~/lab1_left$ ./p1.out
time = 34.425037 s
xy@debian:~/lab1_left$ gcc -O1 main.c p1.c -o p1.out
xy@debian:~/lab1_left$ ./p1.out
time = 14.142221 s
xy@debian:~/lab1_left$ gcc -O2 main.c p1.c -o p1.out
xy@debian:~/lab1_left$ ./p1.out
time = 14.161704 s
```

```
xy@debian:~/lab1_left$ gcc -O0 main.c p2.c -o p2.out
xy@debian:~/lab1_left$ ./p2.out
time = 2.101662 s
xy@debian:~/lab1_left$ gcc -O1 main.c p2.c -o p2.out
xy@debian:~/lab1_left$ ./p2.out
time = 1.874870 s
xy@debian:~/lab1_left$ gcc -O2 main.c p2.c -o p2.out
xy@debian:~/lab1_left$ ./p2.out
time = 1.839278 s
```

```
xy@debian:~/lab1_left$ gcc -O0 main.c magical_code.c -o magic.out
xy@debian:~/lab1_left$ ./magic.out
985318 cases are wrong.
time = 0.716590 s
xy@debian:~/lab1_left$ gcc -O1 main.c magical_code.c -o magic.out
xy@debian:~/lab1_left$ ./magic.out
985318 cases are wrong.
time = 0.697638 s
xy@debian:~/lab1_left$ gcc -O2 main.c magical_code.c -o magic.out
xy@debian:~/lab1_left$ ./magic.out
985318 cases are wrong.
time = 0.707553 s
```

可以看到，并没有打印出报错信息（后来我又加上了错误 case 计数，错误 case 数为 0）。由于一百万组的测试数据量已经不小了，所以我们暂且认为这两个实现都是正确的。由于上述运行时间除了与正确结果的比较之外，主要为一百万次 multimod 的时间，而这次比较用到了 multimod 的结果，正好可以避免编译器把 multimod 的运算忽略掉，因而我直接将其作为时间的计算依据。

所测试代码	p1.c			p2.c			神奇		
优化级别	00	01	02	00	01	02	00	01	02
一百万组所用时间(秒)	34.43	14.14	14.16	2.10	1.87	1.84	0.72	0.70	0.70

可以看到，对于 p1.c，优化的效果还是比较明显的，但是对于其他的代码，优化级别对时间的影响很小，几乎可以忽略。并没有因为优化使很多代码没有执行，因为每次的参数都是不同的，而且结果用于了比较，并非没有使用，也不会被优化掉。可以看到，p2.c 的实现方式大概比 p1.c 快了 17 倍，而神奇的代码大概比 p2.c 快接近三倍，但是这其中有非常多的错误（后来我又换了几次测试用例，差距和这个都不是很大）。

任务三：

首先我还是用测试任务一和任务二的方法测试了一下任务

三的神奇代码，发现虽然很快（大概比任务二的代码还要快了 3.3 倍），但是错误率非常高（因为随机生成的话生成较大的数的概率非常高），因此我们需要得到正确的 a、b、m 的范围。

```
xy@debian:~/lab1$ gcc main2.c magical_code.c -o magic.out
xy@debian:~/lab1$ ./magic.out
985225 cases are wrong.
time = 0.664093 s
```

我又自己找了一些不大的数据试试：

Linux 下

```
xy@debian:~/lab1$ gcc main2.c magical_code.c -o magic.out
xy@debian:~/lab1$ ./magic.out
99999999 99999999 3
3
```

Windows 下

```
99999999 99999999 3
a * b = 9999999800000001
(double)a * b = 9999999800000000.000000
(double)a * b / m = 3333333266666666.500000
((double)a * b / m + 1e-8) = 3333333266666666.500000
(int64_t)((double)a * b / m + 1e-8) = 3333333266666666
(int64_t)((double)a * b / m + 1e-8) * m = 9999999799999998
t = 3
3
```

可以发现，对于这组一点都不大的值（仍然在 int 范围内），他居然都算不对！我又打印出了过程中的值，来看看为什么不对。首先，a * b 完全没有溢出，(double)a * b 发生了溢出，值减少了 1，然后除以 m 的结果也越发不准了，加上 1e-8 并没有什么影响（double 精度不够），最终的结果就变成了 3。其实这个例子直接 return a * b % m 都能算对的，所以我猜测只要 double 精度不够，就有可能错误，而且我手动测了几组，发现 m 在 1-3 的时候还是挺容易出错的，于是我

把 m 定为 1，a 和 b 上限定在 99999998 开始尝试。

```
The answer of 97224697 * 96372397 % 1 should be 0, but get 1.
The answer of 94546591 * 98226043 % 1 should be 0, but get 1.
The answer of 98160521 * 94920485 % 1 should be 0, but get 1.
The answer of 95217413 * 96972965 % 1 should be 0, but get 1.
The answer of 99031577 * 92288817 % 1 should be 0, but get 1.
The answer of 97557913 * 92964277 % 1 should be 0, but get 1.
The answer of 95188459 * 95166711 % 1 should be 0, but get 1.
The answer of 95624269 * 94198285 % 1 should be 0, but get 1.
The answer of 98263665 * 94818317 % 1 should be 0, but get 1.
The answer of 99577717 * 96492889 % 1 should be 0, but get 1.
The answer of 98996785 * 91177897 % 1 should be 0, but get 1.
The answer of 91695919 * 99173895 % 1 should be 0, but get 1.
The answer of 92157663 * 99176619 % 1 should be 0, but get 1.
The answer of 95938207 * 94450923 % 1 should be 0, but get 1.
The answer of 96410357 * 94191829 % 1 should be 0, but get 1.
The answer of 93853239 * 98133547 % 1 should be 0, but get 1.
The answer of 98287427 * 95391111 % 1 should be 0, but get 1.
The answer of 96198445 * 99825317 % 1 should be 0, but get 1.
The answer of 97967945 * 93769781 % 1 should be 0, but get 1.
```

发现错误还是挺多的，所以我缩小了 a 和 b 的范围到 90000000，然后发现没有错。我随机多次改动了 m 的范围，从很小到大范围，将测试用例规模改到了一千万，发现并没有出错。随后我尝试扩大范围，将 a 和 b 范围扩大到了 95000000，依旧用较小和较大范围的 m 分别测试，并未出错，而 a 和 b 范围是 96000000 就会有错。至此，我觉得已经测试得差不多了，猜测正确范围为 a、b 小于等于 95000000（后来分析代码后发现仍然不够精确，还是偏大了），m 为任意的 int64_t 正数。（有试过去掉 1e-8 但是影响不大）。

```
The answer of 95920427 * 93945607 % 1 should be 0, but get 1.
The answer of 95507293 * 94478349 % 3 should be 0, but get 3.
The answer of 95461953 * 95245629 % 1 should be 0, but get 1.
The answer of 94019539 * 95808035 % 1 should be 0, but get 1.
The answer of 95213895 * 95667571 % 1 should be 0, but get 1.
The answer of 94588797 * 95704101 % 3 should be 0, but get 3.
The answer of 94069285 * 95946737 % 1 should be 0, but get 1.
The answer of 95179315 * 95372015 % 1 should be 0, but get 1.
The answer of 94668721 * 95978785 % 1 should be 0, but get 1.
The answer of 94599585 * 95666457 % 3 should be 0, but get 3.
The answer of 94953335 * 95329543 % 1 should be 0, but get 1.
The answer of 94999345 * 95079013 % 1 should be 0, but get 1.
The answer of 95493691 * 94342407 % 1 should be 0, but get 1.
The answer of 95099103 * 95275755 % 1 should be 0, but get 1.
The answer of 95155221 * 95169221 % 1 should be 0, but get 1.
The answer of 94486689 * 95606025 % 3 should be 0, but get 3.
The answer of 95286545 * 95948161 % 1 should be 0, but get 1.
time = 0.270579 s
```

所以我们需要来分析一下这个神奇的代码：

首先分析一下运算顺序和每个表达式的类型：

a * b: int64_t，按照假设，a * b 的值为 (int64_t)((uint64_t)a * (uint64_t)b)

(double)a: double

(double) a * b: double * int64_t = double

(double)a * b / m: double/int = double

((double)a * b / m + 1e-8): double

(int64_t)((double)a * b / m + 1e-8): int64_t

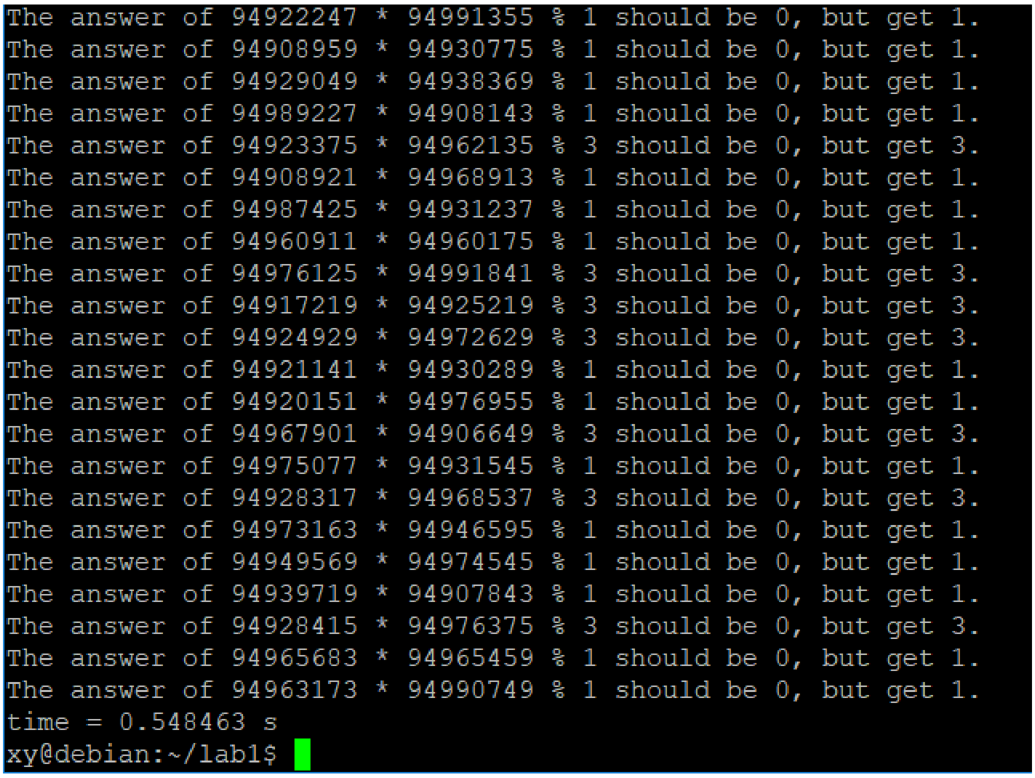
(int64_t)((double)a * b / m + 1e-8) * m int64_t

t int64_t

返回值 int64_t

根据我的测试大概可以得出，在 a*b 超过 double 可以表示的精度后就会导致结果可能不准，而由于 double 尾数位数的限制，尾数最大只能 1.111…1（小数点后 52 个 1），因此

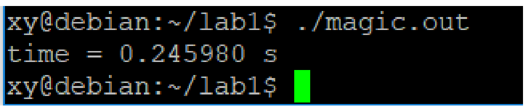
a*b 超过 2^{53}-1 就可能不能用 double 精确表示。因此，a 和 b 应该最大为 94906265。因此我重新开始了测试，将 a 和 b 的范围定在了我的理论值 94906265 和测试值 95000000 之间，



```
The answer of 94922247 * 94991355 % 1 should be 0, but get 1.
The answer of 94908959 * 94930775 % 1 should be 0, but get 1.
The answer of 94929049 * 94938369 % 1 should be 0, but get 1.
The answer of 94989227 * 94908143 % 1 should be 0, but get 1.
The answer of 94923375 * 94962135 % 3 should be 0, but get 3.
The answer of 94908921 * 94968913 % 1 should be 0, but get 1.
The answer of 94987425 * 94931237 % 1 should be 0, but get 1.
The answer of 94960911 * 94960175 % 1 should be 0, but get 1.
The answer of 94976125 * 94991841 % 3 should be 0, but get 3.
The answer of 94917219 * 94925219 % 3 should be 0, but get 3.
The answer of 94924929 * 94972629 % 3 should be 0, but get 3.
The answer of 94921141 * 94930289 % 1 should be 0, but get 1.
The answer of 94920151 * 94976955 % 1 should be 0, but get 1.
The answer of 94967901 * 94906649 % 3 should be 0, but get 3.
The answer of 94975077 * 94931545 % 1 should be 0, but get 1.
The answer of 94928317 * 94968537 % 3 should be 0, but get 3.
The answer of 94973163 * 94946595 % 1 should be 0, but get 1.
The answer of 94949569 * 94974545 % 1 should be 0, but get 1.
The answer of 94939719 * 94907843 % 1 should be 0, but get 1.
The answer of 94928415 * 94976375 % 3 should be 0, but get 3.
The answer of 94965683 * 94965459 % 1 should be 0, but get 1.
The answer of 94963173 * 94990749 % 1 should be 0, but get 1.
time = 0.548463 s
xy@debian:~/lab1$
```

(这里我又重新用了 ssh 所以界面有点不同)又发现了很多错误，这说明之前的测试仍存在一些问题，不够精确。接下来我开始测试 a 和 b 在（0，94906265）的情况，尤其是重点测试了边缘数据(94000000, 94906265)，发现并没有错误。

因此我的最终结论是，a 和 b 在[0，94906265]（其实只要 a*b<=2^{53}-1），m 为任意非 0 的 int64_t 无符号数的时候这个代码一定是正确的。



```
xy@debian:~/lab1$ ./magic.out
time = 0.245980 s
xy@debian:~/lab1$
```

此图是针对较小的 m(1 或 2 或 3)，较大的 a 和 b（0 到

94906265）测试的。

```
xy@debian:~/lab1$ ./magic.out
time = 0.553432 s
```

此图是我针对在所有我上述假设的可行范围内的 a，b，m 生成随机数据测试的，可以看到并没有错误，因为数据规模小，所以速度更快了。

```
xy@debian:~/lab1_left$ ./magic.out
time = 0.541060 s
xy@debian:~/lab1_left$ gcc main2.c magical_code.c -o1 magic.out
xy@debian:~/lab1_left$ ./magic.out
time = 0.531399 s
xy@debian:~/lab1_left$ gcc main2.c magical_code.c -o2 magic.out
xy@debian:~/lab1_left$ ./magic.out
time = 0.530916 s
```

之前说了为什么 a*b 较大会是错的，下面分析一下为什么这个范围内是对的。首先，这个范围内 double 可以准确表示，(double)a * b / m 可以精确地表示除法之后的值，加上 1e-8 等于没加（我去掉以后测试过了），接下来转为 int64_t 不损失精度，而减了以后必定是正的，最后那个判断也是没有用的（我也去掉之后测试过了），而在不溢出不损失精度的情况下，整型运算中 a * b % m = a * b - (a * b / m) * m，因此结果是正确的。（似乎是废话？）

总而言之，从正确率至上的角度考虑，我似乎觉得这个神奇的代码一点都不神奇。