姓名：张浩南
学号：181860134

实验目标

`multimod(a,b,m)`:给定 64 位有符号整数 a, b, m(类型为int64_t)，你希望求出 $a \times b \mod m$的数值，即最小的非负整数t，满足 $a \times b \equiv t \mod m$
根据你所掌握的知识，实现正确、高效的multimod

实验中使用的一些测试方法

测试正确性的标准程序

本实验中需要测试设计的函数体是否正确，而在C++语言中自带`__int128`类型能表示128位有符号整数，表示范围在$[2^{-127}, 2^{127} - 1]$，可以正确存储实验中$a * b$的结果，因此得到以下可以正确返回结果的函数体。

```
int64_t multimod(int64_t a,int64_t b,int64_t m)
{
    int64_t ans=(__int128)a*b%m;
    return ans;
}
```

随机数据生成器

实验过程中需要大量数据来测试所写程序是否能正确输出预期结果，因此用以下代码生成a, b, m的数据。

该程序每次生成200组数据，共分为4类：
1.$a, b, m < 2^{31}$
2.$a, b < 2^{63}, m < 2^{31}$

$3.a,b<2^{31},m<2^{63}$
$4.a,b,m<2^{63}$
且在第一类中生成了5组$b=0$的数据

```
#include<cstdio>
#include<iostream>
#include<cstdlib>
#include<algorithm>
using namespace std;
long long a,b,m;
int main()
{
        freopen("input.in","w",stdout);
        srand(time(0));
        for (int i=1;i<=50;++i)//a,b<2^31,m<2^31
        {
                a=rand();
                b=rand();
                if (i%10==0) b=0;
                m=rand();
                if (a>=0&&b>=0&&m>0) cout<<a<<" "<<b<<" "<<m<<" "<<endl;

        }
        for (int i=1;i<=50;++i)//a,b<2^63,m<2^31
        {
                a=rand();
                a=((a<<31LL)|rand());
                b=rand();
                b=((b<<31LL)|rand());
                m=rand();
                if (a>=0&&b>=0&&m>0) cout<<a<<" "<<b<<" "<<m<<" "<<endl;

        }
        for (int i=1;i<=50;++i)//a,b<2^31,m<2^63
        {
                a=rand();
                b=rand();
                m=rand();
                m=((m<<31LL)|rand());
                if (a>=0&&b>=0&&m>0) cout<<a<<" "<<b<<" "<<m<<" "<<endl;
        }
        for (int i=1;i<=50;++i)//a,b,m<2^63
        {
                a=rand();
                a=((a<<31LL)|rand());
                b=rand();
```

```
                b=((b<<31LL)|rand());
                m=rand();
                m=((m<<31LL)|rand());
                if (a>=0&&b>=0&&m>0) cout<<a<<" "<<b<<" "<<m<<" "<<endl;
        }
        return 0;
}
```

对比程序

有了数据生成器后，我们可以分别运行需要测试的程序和标准程序，然后通过对比输出结果来判断测试程序是否生成预期结果，而这一过程可以通过对比程序来实现。

共循环100次，每次测试共200组数据（如上）

```
#include<cstdio>
#include<cstdlib>
using namespace std;
int main()
{
        int cnt=0;
        for (int i=1;i<=100;++i)
        {
                system("./random");//运行随机数据生成器
                system("./test");//运行需要测试的程序(p1/p2/p3)
                system("./mul");//运行标准程序
                if (system("diff my.out std.out")) break;
                //如果输出结果不同则中止程序
                printf("success:data %d\n",++cnt);//在该组数据下测试正确
        }
        return 0;
}
```

测试代码运行时间

在C的`time.h`库中有`clock()`函数，功能是计算返回程序执行起，处理器时钟所使用的时间，因此分别在程序开头和结尾调用`clock()`并保存，相减后得到中间程序段运行时间（如果想以秒为单位还需要除以`CLOCKS_PER_SEC`）。

任务一

思路：将a,b拆成二进制位，即$a=a_0*2^0+a_1*2^1+...a_{62}*2^{62},b=b_0*2^0+b_1*2^1+...b_{62}*2^{62}$

其中$a_i,b_i\in\{0,1\}$

所以$c=a\times b=\sum_{i=0}^{124}(2^i\cdot\sum_{j=0}^{i}a_jb_{i-j})=\sum_{i=0}^{125}c_i*2^i$，其中$c_i\in\{0,1\}$且当$i\geq 63$时有$a_i,b_i=0$

我们很容易用双重循环得出每项i的$\sum_{j=0}^{i}a_jb_{i-j}$（结果显然在`int`范围内的）并存在数组c'中，之后我们要将系数c'_i转化成二进制表示c_i，这与竖式计算中的进位有异曲同工之处，即令$c_{i+1}\leftarrow c_{i+1}+\lfloor\frac{c_i}{2}\rfloor,c_i\leftarrow c_i \mod 2$，直至达到最高位。

值得注意的是在等式前一项中我们仅将i枚举至124，但实际上在c的二进制表示中最多可能有126位，所以还需要考虑c_i最高位是否还存在进位，使得最高位前移一位。

显然，无论是计算系数c'_i还是通过c'计算c时，内部运算结果都远小于$2^{31}-1$，不会发生溢出。

得到二进制表示后c_i就可以用秦九韶算法将c展开并进行取模运算，即用64位无符号整数存储当前答案ans，从二进制高位向低位枚举，将ans乘2并加上当前位0/1后对m取模，因为$0\leq ans<m\leq 2^{63}-1$，所以$0<2*ans+1<2^{64}-1$未发生溢出。

最终得到的结果ans即为所求$a*b$对m取模的结果，且由上分析可知，整个过程中未发生计算溢出，因此从理论上证明该算法正确。

实现函数体代码如下：

```
int64_t multimod_p1(int64_t a, int64_t b, int64_t m) {
  int wa[140],wb[140],wc[140],la,lb,lc;
  for (int i=0;i<128;++i) wa[i]=wb[i]=wc[i]=0;
  la=lb=lc=0;
  int64_t x=a;
  do
  {
        wa[la++]=(x&1);
        x>>=1;
  }while (x);
  x=b;
```

```
  do
  {
          wb[lb++]=(x&1);
          x>>=1;
  }while (x);
  for (int i=0;i<la;++i)
          for (int j=0;j<lb;++j)
                  wc[i+j]+=(wa[i]&wb[j]);
  for (int i=0;i<la+lb;++i) wc[i+1]+=(wc[i]/2),wc[i]&=1;
  lc=la+lb;
  while (lc>=2&&wc[lc-1]==0) --lc;
  uint64_t ans=0;
  for (int i=lc-1;i>=0;--i) ans=(ans*2+wc[i])%m;
  return (int64_t)ans;
}
```

利用上文提到的随机数据生成器和对比程序进行测试，得到结果为100次测试全部正确，且多次运行对比程序后仍无错误
部分测试数据如下（前10组）

```
1619743825472255004 4450393839207742734 2127858513718690587
3807895103352991582 3589134851058142221 1081924793237355051
1930750270118973498 3812449278319792831 796627819012514531
150910431964366596 2337124458532897271 4399962564323409194
1614787608364137935 1499772434769136381 1044812834781474291
3618748372492800083 2389600542452528174 3549582737564400166
2106131723480073785 3411534728213515661 3044654502355923776
1033158145613521038 1050220233848921230 3766480053647615242
3492189362038946760 1923835316960740369 3727536693906327893
335413295920414539 3738624938017277797 3697144286979317406
```

标准程序输出

```
1453338970750436304
83565670559936406
710507343399404523
215508354653124280
86189756160106079
2255890019651592380
2530843578301959909
3159244495946417362
143049600824207268
2038473275604972969
```

需要测试的程序输出

```
14533389707504363O4
8356567O559936406
71O5O7343399404523
2155O8354653124280
86189756160106079
22558900196515923800
2530843578301959909
31592444959464 17362
143049600824207268
2038473275604972969
```

任务二

优化

考虑对任务一中的方法进行优化，我们注意到如果只对一个数（如b）做二进制拆分，那么计算式就变成了若干个$2^i * a$相加的形式，因此我们只要就可以从低位向高位递推，通过$2^{i-1} * a \mod m$计算$2^i * a \mod m$，再由b的二进制表示判断哪些$2^i * a$需要加进答案即可。代码如下：

```
int64_t multimod_p2(int64_t a, int64_t b, int64_t m) {
  uint64_t ta=a,tb=b,ans=0,tm=m;
  for (;tb;tb>>=1,ta=(ta+ta)%tm)
          if (tb&1) ans=(ans+ta)%tm;
  return (int64_t)ans;
}
```

与任务一的分析类似，因为计算过程中$2^i * a \mod m$和答案始终比m小，所以无论是计算$(2^i * a) * 2 \mod m$还是将其加到答案里都不会超过`uint64_t`的表示范围，所以不会发生溢出。

通过类似任务一的方法使用对比文件进行测试，无错误出现。

比较

对随机数据生成器稍加修改，使其生成10^6组第4类数据输入到输入文件`input.in`中。

将`p1`,`p2`函数合并到同一代码文件下，并用上文所说的clock()函数方法分别计算用时，边读入`input.in`中每组`a,b,m`边计算结果，且为防止编译优化，将结果进行之前所有结果的异或和做异或，最后输出运行时间。

这一方法使程序运行变量仅为调用函数的不同(`p1`或`p2`)，并控制其他无关变量保持不变，且不会导致编译优化非预期地改变程序行为，可以较为准确地衡量和比较两函数的运行时间
同时考虑读入大量数据时消耗的时间，还测试了仅读入数据且不做任何乘法取模操作所需的时间。

```
int main()
{
        freopen("input.in","r",stdin);
        clock_t start_t,end_t;
        start_t=clock();
        while (~scanf("%lld%lld%lld",&a,&b,&m)) ans^=multimod_p1(a,b,m);
                                                /*multimod_p2(a,b,m)*/
        end_t=clock();
        printf("running time=%.6lf s\n",(double)(end_t-start_t)/CLOCKS_PER_SEC);
        return 0;
}
```

不同编译优化级别的测试结果如下

运行时间(s)	multimod_p1	multimod_p2	只读入数据，不做运算
-O0	12.629681	1.326082	0.755667
-O1	4.735367	1.216147	0.744100
-O2	4.514547	1.245649	0.747293

分析

显然，程序的主要时间复杂度取决于所用的计算函数的的复杂度。

`multimod_p1`在对a,b二进制转换后，计算c'时调用了双重循环，可见该函数的复杂度是$O(\log a \log b)$，数量级为$O(\log^2 n)$，且操作步骤多，常数较大。

而`multimod_p2`仅对b做二进制转换，单层循环中操作均为$O(1)$，因此函数的复杂度为$O(\log b)$，数量级为$O(\log n)$，且操作简单步骤少，常数较小。

不过由于`multimod_p1`中的双重循环仅涉及位运算和加操作，此处常数很小，因此在运行时间表现上并未出现$logn$倍左右的差距，但从数值上看，即使在`-O2`优化级别下`multimod_p1`的运行时间仍是后者的4倍左右，差距较大；而在`-O0`选项下的差距更是达到10倍了，可见复杂度数量级存在差距。

还有就是`multimod_p2`运行时间在不同优化级别下差异不大，一是因为其本身就在1s左右，优化效果不明显；二则是函数中操作少且均为基本运算，优化余地小。

任务三

正确性分析

在表达式`(a*b-(int64_t)((double)a*b/m)*m)%m`中，前面直接计算的`a*b`等价于`(int64_t)((uint64_t)a * (uint64_t)b)`，而对于无符号64位整数的乘法则相当于是在模2^{64}意义下的乘法，之后又强制类型转化为有符号64位整数，因此`a*b`的计算结果为$A = \begin{cases} c & c \le 2^{63}-1 \\ c-2^{64} & 2^{63} \le c < 2^{64} \end{cases} = c - x_1$，其中$c = a \times b \mod 2^{64}$且$x_1 \in \{0, 2^{64}\}$

而`(double)a*b/m`相当于将a,b,m转化成`double`类型进行乘除运算，由浮点数的机器级表示可知`double`类型的位数为47，即对于a,b,m这样的有符号64位整数来说，类型转换后只能保证二进制表示下较高的48位（最高位为1）能被保留，而更低的位数会被丢失，且在乘除运算中同样存在位数丢失，导致计算结果与真正结果($\lfloor \frac{ab}{m} \rfloor$)存在偏差，而最终将其转化为`int64_t`类型后，其真值为$\lfloor \frac{ab}{m} \rfloor + \varepsilon$，其中$\varepsilon \in Z$表示存在的误差。

`(int64_t)((double)a*b/m)*m`令上面的`int64_t`结果与m相乘，由`int64_t`计算方法可得表达式结果为$B = [(\lfloor \frac{ab}{m} \rfloor + \varepsilon) \times m \mod 2^{64}] - x_2$，其中$x_2 \in \{0, 2^{64}\}$

因此表达式`a*b-(int64_t)((double)a*b/m)*m`相当于计算$A-B$再将其转化为`int64_t`下的表示，而由以上推导知

$$
\begin{aligned}
A - B &= a \times b \mod 2^{64} - x_1 - [(\lfloor \frac{ab}{m} \rfloor + \varepsilon) \times m \mod 2^{64}] + x_2 \\
&= (a \times b - \lfloor \frac{ab}{m} \rfloor \times m - \varepsilon \times m) \mod 2^{64} - x_1 + x_2 \\
&= (a \times b \mod m - \varepsilon \times m) \mod 2^{64} - x_1 + x_2 \\
&= a \times b \mod m - \varepsilon \times m + (x_2 - x_1)
\end{aligned}
$$

当`(int64_t)((double)a*b/m)`没有发生溢出且$a \times b \mod m - \varepsilon \times m$不超过$2^{64}$时等式成立，因为前者溢出或后者超出范围时都可能会导致误差ε较大，且$a \times b \mod m \leq m$，需要$|\varepsilon \times m| \leq 2^{64}$，否则a乘b对m取模的结果会在模$2^{64}$的意义下被破坏。

同时$a \times b \mod m - \varepsilon \times m \leq 2^{63} - 1$，因为$x_2 - x_1 \in \{-2^{64}, 0, 2^{64}\}$，所以前者加上后者时会在`int64_t`的范围下进行修正，从而$A - B$最终结果为$a \times b \mod m - \varepsilon \times m$

而令`t=(...)%m`，相当于将$a \times b \mod m - s \times m (s \in \{0,1\})$赋值给`t`，因为C语言中的`%m`运算结果范围为`[-m+1,m-1]`而不一定时正数，所以取模后会存在0/1倍的m差值。

而由于$a \times b \mod m - m < 0$，因此函数最后的`return t<0?t+m:t`则修正了$s = 1$时的差值，最终得到的返回值就是$a \times b \mod m$

当然，以上分析成立的重要条件是ε和m范围合适，使得$\varepsilon \times m$不超过2^{64}，且尽可能使得其小于等于$2^{63} - 1$，而当m较大或double运算过程中误差较大时可能会导致结果错误。
##范围与对比
当$a, b < m \leq 10^{17}$时该方法能返回正确结果。

注意如果不令$a, b < m$的话，即使将其限定在较小范围（如10^{11}），当m很小而a,b较大时也还是会出现错误，而本身要求乘法取模的情况可以事先令`a=a%m,b=b%m`，不影响答案正确性，所以这里令数据中$a, b < m$

测试数据选择生成10^6组$a, b < m \leq 10^{17}$数据，将`multimod_fast`写入文件后分别调用三个函数对同一个输入文件测试运行时间。

该方法与之前做法在不同编译优化级别的测试结果如下

运行时间(s)	multimod_p1	multimod_p2	multimod_fast	只读入数据，不做运算
-O0	10.768888	1.215573	0.362762	0.344108
-O1	4.088931	1.097721	0.365572	0.342250
-O2	4.107808	1.144421	0.361166	0.338707

理论上三种方法复杂度分别为$O(\log^2 n), O(\log n), O(1)$，从运行时间上可以看出，三种方法复杂度差距较为明显，在减去读入操作消耗时间后，`multimod_fast`可以说是“十毫秒过”，而p2则花了秒级时间，p1最慢，花了数秒。