Lab1 实验报告

任务一：实现multimod

由任务二得到的启示，因为C语言提供了__int128类型数据，所以直接采用此类型数据进行模运算的实现。

代码如下：

```
int64_t multimod_p1(int64_t a, int64_t b, int64_t m){
    int64_t ans = ((__int128t)(a * b) % m);
    return ans;
}
```

在128位的帮助下，正确率可达100%，不会发生溢出的情况；

正确率的判断借用python的随机数，并将两者的结果进行比较：

```
import random
MAX = 2 ** 63 - 1
for i in range(1, 500):
    a = random.randint(0, MAX)
    b = random.randint(0, MAX)
    m = random.randint(1, MAX)
    ans = (a * b) % m
    print (a, b, m, ans)
```

任务二： 性能优化

根据实验讲义的提示，可以通过位运算来实现上述的模运算，但同时也要考虑溢出的问题。

$$b = k_0 \cdot 2^0 + k_1 \cdot 2^1 + \ldots + k_{62} \cdot 2^{62}$$

通过适当变形后a*b可以变成

$$a \cdot b = a \cdot k_0 + 2 \cdot (a \cdot k_1 + 2 \cdot (a \cdot k_2 + 2 \cdot (a \cdot k_3 \ldots + 2 \cdot (a \cdot k_{62}))))$$

以下为代码部分的实现

```
int64_t multimod_p2(int64_t a, int64_t b, int64_t m){
    int64_t num = (int64_t)1 << 62;
    int64_t ans = 0;
    a = a % m;
    b = b % m;
    if (m < num){
        for (int i = 62; i >= 0; i--){
```

```
            ans = (ans << 1) % m;
            if (b & ((int64_t)1 << i){
                ans = (ans + a) % m;
            }
        }
    }
    else{
        for (int i = 62; i >= 0; i--){
            if (ans >= ((m >> 1) + (m & 1)))
                ans = ans - (m - ans);
            else ans = ans << 1;
            if (b & ((int64_t)1 << i)){
                int64_t k = m - ans;
                if (a >= k) ans = a - k;
                else ans = ans + a;
            }
        }
    }
    return ans;
}
```

由于存在溢出的问题（乘2左移运算时），所以需要对m进行讨论：

（1）当m小于2^62

由于此时模运算的结果必定是小于`2^62`的，所以乘2（即左移一位）操作不会发生溢出情况，因此可以直接进行左移。每次将当前的结果左移一位（乘2），再对m取模，如果此时对应的b的位为0，则不需要将a对m进行取模，若此时对应的b的位为1，则还要将a对m进行取模，即将（ans+a）对m进行取模，并且将此存放为中间结果，再继续进行循环。

（2）当m大于等于2^62

由于模运算后的结果可能大于等于2^62次方（2^63-1为64位所能表示的最大整数），若直接进行（1）中的左移操作，很可能会发生溢出现象，所以不能直接进行左移，需要考虑溢出问题。

需要考虑`2 * ans >= m`的情况，此时模运算会发生溢出，考虑到直接`m >> 1`时，当为奇数时会向下取整，所以采用`m >> 1 + m & 1`来解决此类情况。又考虑到此时中间结果除以m的值只可能为1或0，所以可以直接相减得到中间结果对m的模，若直接采用`2 * ans - m`来表示得到的模，由于`2 * ans`可能会发生溢出，所以选择`ans - (m - ans)`来实现，此时不会发生溢出；而其余情况下，可以直接左移一位（乘2）。

除此之外，若此时对应的b的位为0时，不需要再进行取模运算，而若此时对应的b的位为1时，若`ans + a >= m`时，中间结果应该改为`ans + a - m`,其余情况则直接为`ans + a`,如此往复循环即可。

经过python随机数测验后，结果表明，实现正确。

其他算法思考

1.重复的循环运算。

将b个a累加，在加的过程中，如果和超过m，则进行取模运算。

例如：（3 * 3）% 4 分成3次累加，3 + 3 = 6 > 4，所以取模，6 % 4 = 2；2 + 3 = 5 > 4，取模为1；

在此过程中仍然要对m的大小分开讨论，同时也要考虑到对a进行累加时的溢出问题，但此种算法还是存在潜在的溢出问题，所以不打算采用此种解法。

2.高精度数乘法和除法

通过高精度数乘法将结果存放在字符数组中，通过模拟手写算法得到模。由于此种算法复杂度较高，所以没有采取。

最后采用了C语言提供的128位和位运算来计算模。

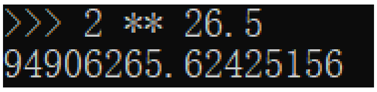
任务三：解析神秘代码

```c
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;
}
```

上述的取模思想与`(a * b - (a * b / m) * m) % m`相类似，奇妙的是这里采用了强制类型转换将a强制转换成double类型。

起初开始测试时，发现在小范围内正确率都很高，而随着范围越来越大，到达几百万之后，发现正确率极低，所以不得不开始思考数据类型不同所带来的差异。

这段代码将a转化为`double`类型，其实间接把ab乘积转换为`double`类型。并且此段代码中a,b的地位是等价的，所以只需要考虑`ab`，即a与b的乘积。`double`类型所能表示的范围大于`int64_t`，但精度不够，尾数部分52位的精度再加上前面的1，只能保证所提到的53位的精度，所以在测试的过程中，使用python计算`2^26.5`的值，大约为94900000，在测试的过程中将随机数调整到（1，94900000），所进行的检测几乎正确。



结论：`ab`乘积不超过`2^53`次方，`m`在`int64_t`范围内任意取值，可以保证`multimod_fast`总是能返回正确的数值

有关时间

采用`time.h`库函数计算运行时间。

```c
#include <time.h>

//函数体内添加
clock_t start, end;
double duration;
start = clock();
......
stop = clock();
duration = ((double)(stop - start)) / CLOCKS_PER_SEC;
printf("%f", duration);
```

时间：

	p1.c	p2.c	p3.c
O0	0.0006	0.0021	0.0005
O1	0.0006	0.0016	0.0005
O2	0.0005	0.0014	0.0005

原因分析：

记`a,b,m`的位数为n 任务一中直接采用128位整数进行计算，由于乘法计算效率较低，但总的来说只需要计算一次，时钟周不会很长。 任务二中优化过的时间需要循环次数是`O(n)`的，每次循环的操作都能在常数时间`O(1)`内完成，因此总时间复杂度为`O(n)`，位运算的次数较多，虽然每次位运算的时钟周期小于乘法的时钟周期，但总的时间超过了乘法的时间。 任务三中神奇的代码对任何输入消耗的时间都是相同的，时间复杂度为`O(1)`，且采用了64位的乘法和加法运算，与任务一中的时间几乎相当。 从运行时间上来看的确是`t(p2)>t(p1)>t(p3)`，符合预期。