Lab1: 数据的机器表示

匡亚明学院 张祎扬 181840326

任务1: 实现multimod

因为`int64_t`的乘法会造成溢出，为了不溢出，我首先想到的是将数据类型转化为`long double`进行运算，因为它最大可以表示的数超过了最大的`int64_t`的两个数的乘积。用整除运算，并将结果保存为`int64_t`整除类型，可以保证结果的准确性，同时又没有溢出。

$$(a*b)\%m = a*b - [\frac{a*b}{m}]*m$$

其中，`[]`表示向下取整，将$a*b/m$的值保存为`int64_t`类型，就可以将数据截断，达到取整的效果。因为最终的结果只看64位，所以在做减法时的溢出对结果没有影响。为了消除负数等的影响，最后的结果先+m,再%m.

代码见文件p1.c

为了验证代码的正确性，我用python随机生成了100万组测试样例（之所以用python是因为之前人工智能课学过python，并且python的数据类型非常开放，对大小和溢出没有限制）。用来生成测试文件的python代码如下：

```python
import random
MAX=9223372036854775807
#MIN=-9223372036854775808
with open("test","w") as file:
    for i in range(0,1000000):
        a = random.randint(0,MAX)
        b = random.randint(0,MAX)
        m = random.randint(0,MAX)
        result = (a*b)%m
        file.write(str(a)+" "+str(b)+" "+str(m)+" "+str(result)+"\n")
file.close()
```

然后我将生成的test文件放入multimod目录下，并且修改了main函数，用来判断p1.c的计算结果与python提供的是否相等，并输出count的值。修改部分的main函数代码如下：

```
    int64_t a,b,m,result;
    int64_t count=0;
    FILE *f = fopen("test","r");
    for(int i=0;i<1000000;i++){
        fscanf(f,"%ld %ld %ld %ld",&a,&b,&m,&result);
        int64_t ret = func(a,b,m);
        if(ret == result)count++;
    }
    printf("the passed number is:%ld\n",count);
```

一开始我发现count的值只有49万多，于是我继续修改main函数，输出不正确的数据，发现很多都是负数。于是我发现我并没有考虑a,b大于m的情况。所以我在p1.c中添加了判断，如果a(b)>m,就先对它取模。最后输出的结果全部正确。

```
katherine@debian:~/ics-workbench/multimod$ ./multimod-64 -i 1
the passed number is:1000000
```

任务2: 性能优化

衡量函数的执行时间

通过search the web,我找到了用`clock()`函数衡量程序运行时间的算法。它需要包括头文件`time.h`，`clock()`函数捕捉程序从开始运行到`clock()`被调用花费的时间，这个时间单位是`clock tick`,即时钟打点。还有常数`CLK_TCK`表示机器时钟每秒所走的时钟打点数。

为了衡量程序执行的时间，只需要在程序调用前后分别调用`clock()`，作差，再除以常数`CLK_TCK`即可。

为了防止程序被优化，我还是保留了原来的计算程序个数的代码，这样可以保证每一次调用func函数都真的进行了计算。

优化

根据讲义提示，可以将式子用乘法分配律展开。

$$\begin{aligned} a*b &= (a_0*2^0+a_1*2^1+\ldots+a_{62}*2^{62})*b \\ &= a_0*b*2^0+a_1*b*2^1+\ldots+a_{62}*b*2^{62} \end{aligned}$$

可以每一次取出a的最后一位，如果是1，就在结果上加b，并且把b的值乘2.在这一过程中随时进行取模运算，可以保证结果始终在一个较小的数据上，不会发生溢出。

但是运行之后我发现事情不是那么简单，大约只有50万左右的正确率。我打印了不正确的输出，发现他们很多是负数，所以我觉得即便是相加也可能发生溢出。但是因为之前进行过处理，所以可以保证相加的值小于原来任何一方的2倍。所以我在函数中间把运算的数据类型改为`uint64_t`,完美解决了问题，代码在p2.c中，正确率为100%。

```
katherine@debian:~/ics-workbench/multimod$ ./multimod-64 -i 2
the time is 1.013601 sec
the passed number is:1000000
```

一点解释：其实位运算也不算是特别快的算法，和第一种方法比起来它的时间复杂度大很多。但是因为在任务2中给了位运算的提示，但其实位运算是很容易想到的一个方法（可能因为学汇编的时候做多了，感觉为了防止溢出而采用位运算是挺正常的想法），所以就把位运算放在p2.c里了。但其实p1.c里的代码相比p2.c要优化得多。要笨一点的方法的话，可以先用高精度乘法，然后循环做减法，但这样的开销很大，我也就不把代码往上写了。

比较各个实现的优化时间

到目前为止，p1.c p2.c中代码的正确率都是百分之百，但是我试着跑了一下p3.c,正确率非常低（**这个到任务三中会继续探讨**），所以它会大量输出不一致的值，导致时间变慢。所以为了比较性能，我只保留最后对正确率的输出，而把中间输出不正确的结果注释掉。

p1.c

优化级别	第一次	第二次	第三次	平均时间
-O0	0.375292	0.371281	0.377603	0.374725
-O1	0.364261	0.363141	0.367167	0.364856
-O2	0.367642	0.366924	0.372898	0.369154

p2.c

优化级别	第一次	第二次	第三次	平均时间
-O0	1.117339	1.118612	1.114271	1.116741
-O1	1.122556	1.134512	1.112626	1.123231
-O2	1.131636	1.129229	1.136304	1.132390

p3.c

优化级别	第一次	第二次	第三次	平均时间
-O0	0.363877	0.366410	0.361293	0.363860
-O1	0.369893	0.368974	0.367031	0.368723
-O2	0.369804	0.364992	0.360500	0.365099

分析

p1.c中的代码并没有系统的循环，基本只要执行几行代码，所以时间复杂度是O(1).

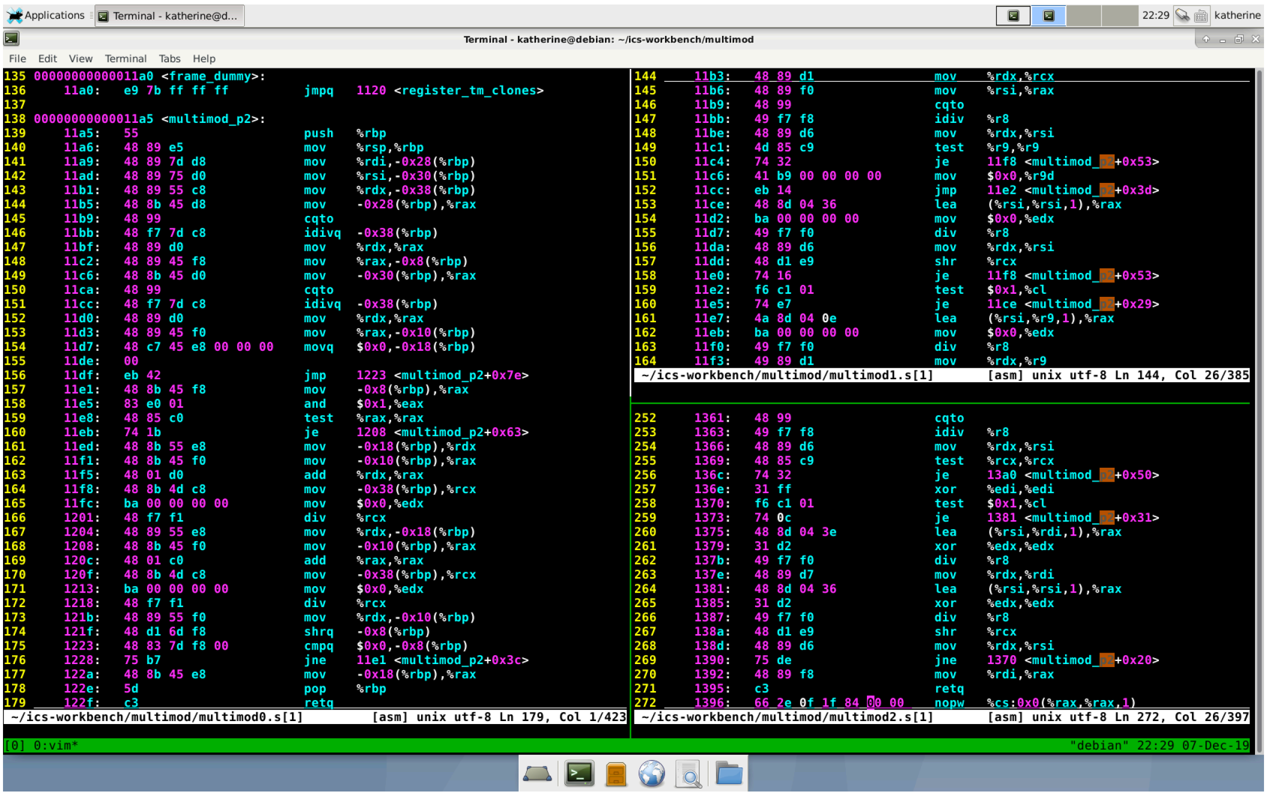
p2.c中有一个循环，循环次数为n，即数据的位数，每次循环的时间都是常数时间，所以时间复杂度是O(n).

p3.c函数体中只有一行代码，对于任何输入的消耗时间相同，时间复杂度是O(1).

从以上对于函数的执行时间的比较来看，p1.c 和 p3.c 的时间复杂度都是O(1),它们所花的时间相差无几，都在0.36秒-0.37秒（一百万次）。p2.c的时间复杂度是O(n),它所用的时间远长于p1.c 和 p2.c, 有1秒多（一百万次）。总体上来说是$t(1) \approx t(3) > t(2)$,测量结果和分析一致。

一些发现：程序执行的时间和优化等级并不一定有严格的相关规律。在p1.c中，-O1的优化效果最好，在p2.c中，随着优化等级的提升，程序执行所用的时间竟然不断增加。而在p3.c中，-O1优化的表现又是最差的。从中可以看出，优化等级的提升不一定可以体现出程序执行性能的提升，根据不同的代码，同样的优化等级也会有不同的效果，可以根据代码的特色选择最适合的优化等级。

我对三种优化编译生成的可执行文件分别进行objdump，以p2.c为例，我发现在不使用优化（等级为0）时主要是对栈进行操作，使用优化等级之后基本只使用寄存器。因为寄存器的存取速度比堆栈快得多，从而达到优化。

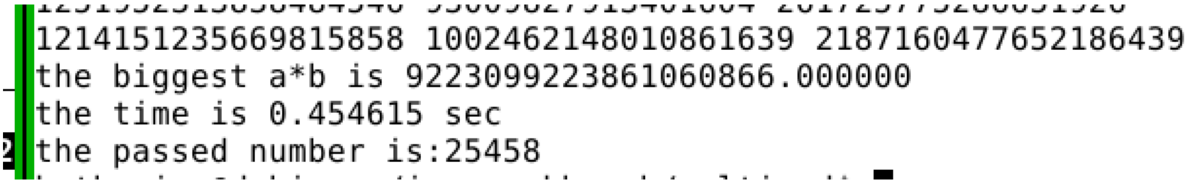


但是我能力有限，关于优化之后时间的反常增加，并没有找到原因所在，看来我还有很多要学习的地方。不过不管怎么说，编译器肯定是正确的。

任务3: 解析神秘代码

用p3函数测试我一开始准备的一百万条用例，只过了25458条，才百分之2，这个比例是极低的。所以它肯定有一定的限制条件。

我修改了main函数，当结果正确的时候判断ab的乘积是否比之前大，用long double来储存ab的乘积并不断更新直到其为最大值。最后输出的结果如下：

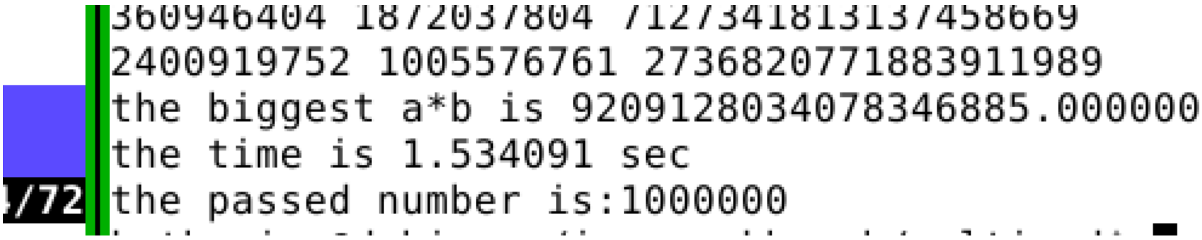


在这一百万组测试用例中，最大的a*b的乘积是9223099223861060866.(因为ab是等价的所以只关注乘积)，当ab乘积过大时肯定会溢出，所以我先假设ab不溢出（不超过double的表示范围）。因为double的表示范围大于int64，所以int64能表示的double都能表示。根据讲义提示，假设a*b定义为int64，那么将ab的乘积转化为double的那一步是不会溢出的，但是因为doube的精度不够，只有52+1=53位，不足64位，所以肯定是有精度损失的。一旦这里发生精度损失，再除以m乘以m并且强制类型转换，精度损失会更多，也就基本不可能正确了。于是我假设ab的乘积最大是2^53.我计算出这个值是9007199254740992,我发现它小于我得到的最大的ab乘积。

于是我用python写了一段程序来找这个值，代码如下：

```python
x=0
while (2**x <9223099223861060866):
    x+=1
print(x)
```

最后的输出是63，于是我猜测当ab的乘积小于2的63次方，m在$0\sim2^{63}-1$,满足条件。我用python算出2^31.5是3037000499.97605，取3037000499。我重新将a，b控制在这个范围内，生成100万组随机数和他们的结果。测试发现通过了所有样例。



再次重新生成100万个样例，结果还是全部通过，所以我有理由相信我得出的结论是对的，即当ab的乘积小于2^63次时，m为int64_t型数据时，程序的输出结果永远正确。再思考一下，其实就是当ab的乘积仍不超出int64_t的表示范围，即2^63-1时,该函数总能返回正确的数值。

这就更容易理解了。当ab的乘积没有超过int64_t的表示范围，无论是转化为double还是转回int64_t，都不会溢出，而double虽然损失了一点精度，不过本来就要做整除，损失的精度对数据截断后对精度没有影响。

最终结论：当ab的乘积不超过$2^{63}-1$,m为int64_t型数据时，函数能返回正确的数值。

根据任务二中的性能比较，它确实优于p2.c的位算法。这个函数在某些方面和p1.c中的思想有些类似，它们都是O(1)时间复杂度，所以相差不大。