Lab1: 数据的机器表示

181220075 周其乐

生成测试用例与计算程序执行时间思路与代码

因为 python 拥有任意精度数据类型，同时实验环境 linux 中已配置 python。所以选择通过 python 语言来生成测试用例。通过 random 包与 os 包，完成从生成 1000 个测试用例、在命令行输入、到最后验证正确性的整个过程。测试文件会输出所有测试用例的信息，以及最后统计 1000 个测试用例中的正确个数与错误个数。

在 main.c 中使用 time.h 中的 clock 函数，进行程序运行时间的计算，在 main.c 调用 func(a, b, m)前先调用 clock 函数计算 begin_time，在调用 func(a, b, m)后马上调用 clock 函数计算 end_time，最后用过 run_time = end_time - begin_time，将 run_time 与程序执行结果 ret 一起传入 test.py 中进行总的运行时间的求和，及平均运行时间的计算。这样的计算时间方法可以保证得到的 run_time 尽量接近于实际的程序执行时间。

test.py 代码如下（对 main.c 的修改如上述文字描述）：

```
import random
import os

#p1, p2 or p3
choosen_func_no = "2 "

#run time
total_runtime = 0
MS_PER_SEC = 1000

#test instances number
test_num=1000

#generate test a b m
LIMIT=2**63-1
wrong_num=0
for i in range(test_num):
    a=random.randint(0,LIMIT)
    b=random.randint(0,LIMIT)
    m=random.randint(1,LIMIT)

```

```
    whole_command ="/home/qile/ics-workbench/multimod/main -i "+\
                    choosen_func_no+str(a)+" "+str(b)+" "+str(m)
    opened_file=os.popen(whole_command)

    result_list=opened_file.readlines()

    run_time=result_list[1].strip()
    total_runtime+=int(run_time)

    my_result=result_list[0].strip()
    if(int(my_result)!=(a*b%m)):
        wrong_num=wrong_num+1
        print("wrong result: "+"\n"\
              +str(a)+" * "+str(b)+" % "+str(m)+"\n"\
              +"my_result:"+str(my_result)+"\n"\
              +"right result:"+str(a*b%m)+"\n")
    else:
        print("right result: "+"\n"\
              +"for a:"+str(a)+" b:"+str(b)+" m:"+str(m)+"\n"\
              +"result:"+str(my_result)+"\n")

print("in "+str(test_num)+"tests, right times:"+str(test_num-wrong_num)+\
      ", wrong times:"+str(wrong_num)+"\n")

total_runtime=float(total_runtime)/MS_PER_SEC
print("for "+str(test_num)+"tests, total runtime is: "+str(total_runtime)+"
ms\n"\
      "average time for one test is: "+str(float(total_runtime/test_num))+"
ms\n")
```

任务一

最开始打算先尝试使用暴力方法解决，于是直接将乘法拆分成若干加法。虽然在开始先进行预处理，比较 a 和 b 的大小、并选择较小那一个为循环次数，但是算法时间复杂度本身依赖于 **min{a,b}**，而本实验的 a 和 b 都是大数，这个算法仅仅测试一个用例都要花很长很长的时间。

代码如下：

```
int64_t multimod_p1(int64_t a, int64_t b, int64_t m) {
  int64_t result=0;

  if(b>a){
    int64_t temp = a;
```

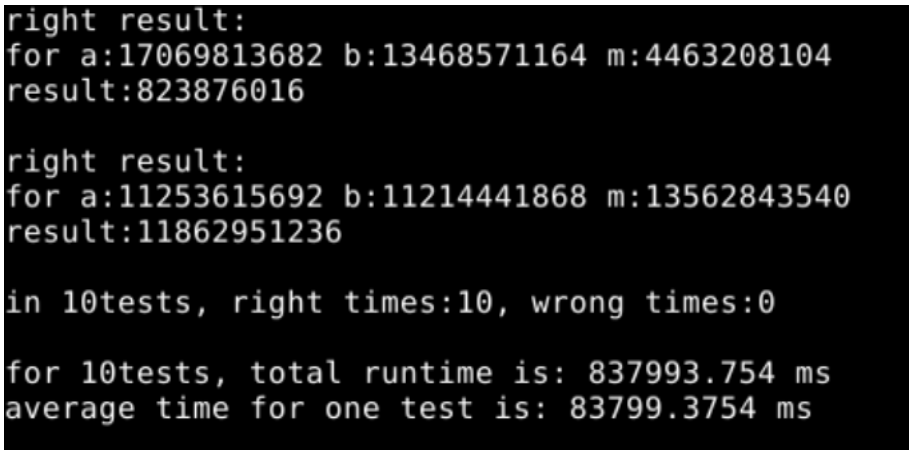
```
    a = b;
    b = temp;
  }

  for(int64_t i = 0; i<b; i++){
    result=(result+a%m)%m;
  }

  return result;
}
```

因为这个傻瓜算法**速度实在太慢太慢**，**为了让测试文件跑出结果**，于是修改 test.py，将 LIMIT 重新设置为 2^34-1，将测试用例的总数改为 10，进行测试。

运行 test.py，10 个测试中，全部正确。
运行结果如截图（只截图保留部分正确结果输出信息）：



任务二

暴力解法很不可取，通过 STFW，了解到，对于大数的幂，有一种计算快速幂的模算法。
快速幂循环体中对 a 的乘法可类比于加法。于是将其想法迁移到大数乘法的模算法：
当 b 是一个偶数，a*b = 2*a*(b/2)
当 b 是一个奇数，a*b = a*(b-1)+a

根据算法进行**初步代码实现**后，通过测试用例发现，并不能得到很好的结果，**算法在 m>2^62 时，a % m 仍然可能是一个大于 2^62 的数，此时若进行 a<<1，就会发生溢出**，导致计算错误。

运行 test.py，1000 个测试中，正确 524 个，错误 476 个。
运行结果如下页截图（只截图保留部分正确或错误结果输出信息， 最后会报告最终版本的代码）：

```
right result:
for a:238682140528548144 b:68343697335037145 m:2703593322528423983
result:1556921677178777777

wrong result:
4841101554408789461 * 4848437139076789145 % 5474985700991258100
my_result:-1940969548909360451
right result:393546125672192845

in 1000tests, right times:524, wrong times:476

for 1000tests, total runtime is: 2.232 ms
average time for one test is: 0.002232 ms
```

所以进行代码改进，对 m<2^62 的情况按照原来的方法处理，**对 m>2^62 的情况进行特殊考虑**。增加对 m-result 和 a，m-a 和 a 的大小比较，并分别进行处理。**（代码在后）**

运行 test.py，1000 个测试中，正确 1000 个，错误 0 个。
运行结果如下页中截图（只截图保留部分正确结果输出信息，最后会报告最终版本的代码）：

```
right result:
for a:1480018795354115349 b:52617114378657537 m:3423351104612451084
result:3194333983016008485

right result:
for a:6500743064446307085 b:2435969149317093723 m:3888211597693564249
result:2649508678788378789

in 1000tests, right times:1000, wrong times:0
```

尝试进行优化，在算法层面没能想到可行的优化，仅能进行一些浅层的优化。首先考虑加入基准实现中对 a，b 的大小比较，若 a 比 b 小，则对两者进行交换，从而使 b 尽量小。另外，对 a 和 b 相乘不溢出的特殊情况进行考虑，直接返回 a * b % m。

最终版本代码如下：

```
#include "multimod.h"

int64_t multimod_p2(int64_t a, int64_t b, int64_t m) {

  int64_t result=0;

  if(a<b){
    int64_t temp = a;
    a=b;
    b=temp;
  }

  if((a < (int64_t)1<<31) && (b < (int64_t)1<<31)){
```

```
    return a*b%m;
  }

  if (a>=m) {
    a%=m;
  }
  if (b>=m) {
    b%=m;
  }

  if(m< ((int64_t)1<< 62)){
    while (b) {
      if (b&1) {
        result=(result+a)%m;
      }
      a=(a<<1)%m;
      b>>=1;
    }
  }

  else{
    while (b) {
      if (b&1)  {
        if(m-result>a){
          result+=a;
        }
        else{
          result+=a-m;
        }
      }
      b >>= 1;
      if (b){
        if(m-a>a){
          a<<=1;
        }
        else{
          a=(a<<=1)-m;
        }
      }
    }
  }

  return result;
}
```

性能比较（P2、P3 测试在电脑充电状态下进行）

平均执行时间（单位：ms）	P1 (时间过长，限制数据 2^34-1)	P2 优化前	P2 优化后	P3 (仅约 2%的测试结果正确)
O1 第一轮测试	63409.2928	0.002461	0.002424	0.001917
O1 第二轮测试	66486.1597	0.002438	0.002242	0.001686
O1 第三轮测试	59343.5654	0.002736	0.002485	0.001733
O2 第一轮测试	32939.0276	0.002434	0.002341	0.001726
O2 第二轮测试	128340.0745	0.002293	0.002496	0.001685
O2 第三轮测试	83799.3754	0.002632	0.002304	0.001743
O3 第一轮测试	123371.9395	0.002329	0.002361	0.001714
O3 第二轮测试	88587.8301	0.003479	0.002551	0.001645
O3 第三轮测试	150906.5271	0.002379	0.002327	0.001710

更换 O1、O2、O3 三个编译优化选项并没有太大的影响（P1 部分测试在电脑不充电时进行，所以数据有些波动），说明启用各类优化选项后，程序的结构并没有发生有足够效果的改变。

同时，根据时间的比较，可看出时间复杂度为 O(min{a,b})的暴力算法，和 p2 中算法性能的巨大差异。而 p2 优化前与优化后的效果相比较，差异并不明显，但是优化后平均执行时间确实有少量的下降。分析原因，生成的随机测试数中，绝大部分都是大数，a*b 溢出几乎是全部情况，所以对不溢出情况的优化应该作用微乎其微。执行时间的少量下降应该归功于对 b=min{a，b}的处理，但因为 a，b 基本上处于同一数量级，所以这个优化有一定影响、但较小。

对于 p3 中的神秘代码，使用浮点黑科技，可以在这几个代码中以最短的时间得到结果，但是正确率仅有 2%。后续在任务三中讨论。

任务三

分析结论：

a*b 的二进制结果除去高 52 位后，设剩下的低二进制位的位数为 lowbit，
只要满足条件 m>=2^(lowbit)，就可以得到正确结果。

代码的初步分析：

首先从整体来看，第一个等式的右侧表达式为：

```
a * b % m = a * b - (a * b / m) * m
```

然后将计算强制类型转换为 double 类型。

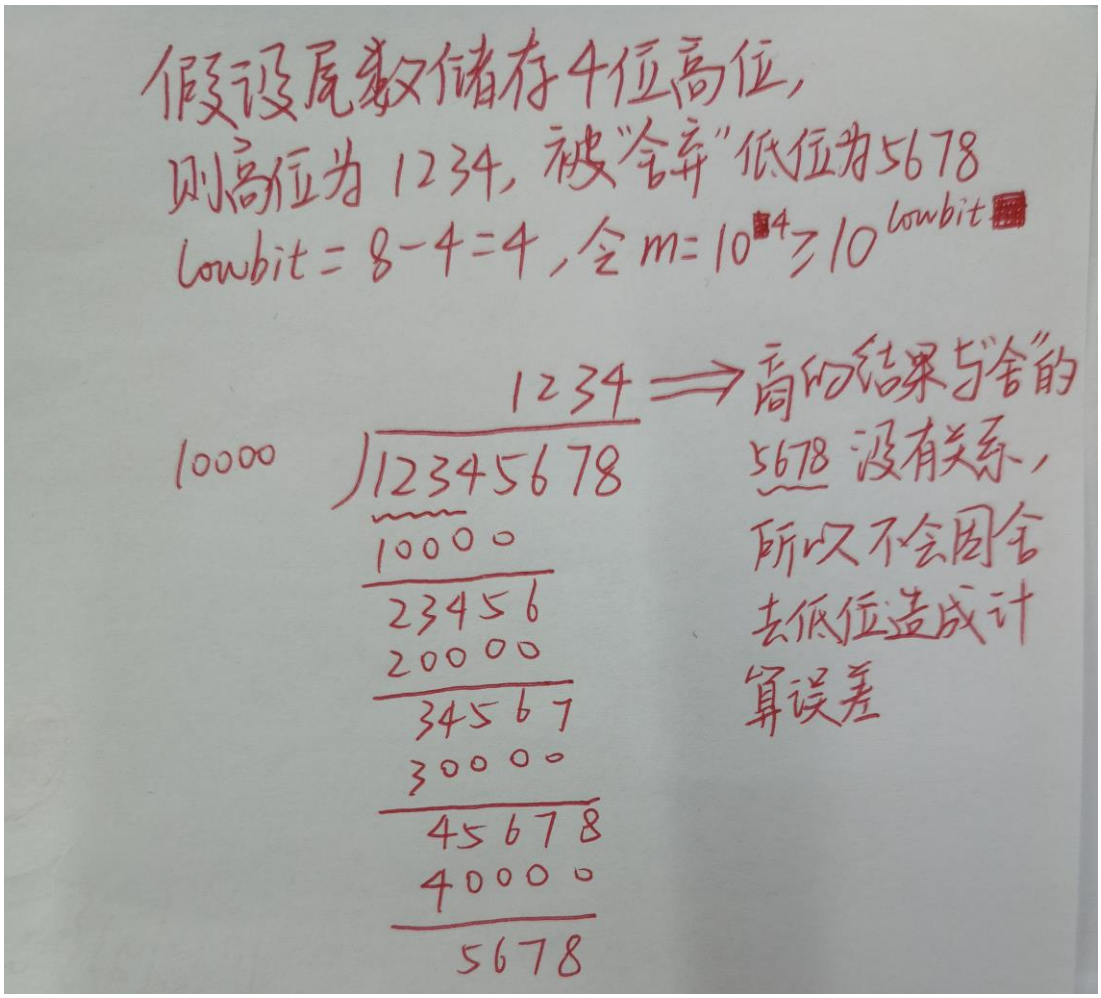
IEEE754 64 位双精度类型由 1 位符号位、11 位指数位、52 位尾数构成。

将数据强制转换为浮点数后，就像科学计数法，尾数部分的 52 位用来储存 a*b 计算结果的高 52 位数据段。高 52 位得以保存，但是对于除去这 52 位的剩下的低位没有位置存放，则会被舍弃掉。这样，在后面的分析中也会提到，a*b 的结果在 /m 时的误差就会与这些舍弃掉的低位有关。计算结果就这样**通过 double 浮点数进行近似**。

最后对 t 的正负进行判断，若为负数则加 m，保证得到的余数是用非负数表示。

对 a, b, m 应满足的条件的分析：
按我的理解，因为 double 尾数只能储存 52 位，a*b 的结果除去高 52 位后，设剩下的低二进制位的位数为 lowbit。按照除法的性质，作为除数 m，只要满足 **m>=2^(lowbit)**，对于 a*b 的 lowbit 个被舍弃低位，不管是多少，a * b / m 得到的商都与这些低位数无关，从而使舍弃低位带来的误差消除，使 multimod_fast 能返回正确的数值。（拿十进制举例如下）

十进制举例：



修改 test.py，借助 python 的 bin 函数，求得 a*b 的二进制表示。
然后通过 **int (len (bin (a * b))) - 2(0b 所占位) – 52(高位)**
获得剩下的低二进制位的位数：**lowbit**
最后的条件判断为：**if ((2 ** lowbit) <= m)**

最后经过多轮实验，在所有得到错误结果的测试样例中，没有任何一组(a，b，m)满足上述条件。且有足够的正确结果的测试样例，满足上述条件。

运行 test.py，10000 个测试中，正确 275 个，错误 9725 个。其中在所有错误中，没有任何一组（a, b, m）满足上述条件。且有 28 组满足该条件的测试用例，获得了正确的结果。说明只要满足上述条件，只会得到正确结果。
运行结果如截图：

```
in 10000tests, right times:275, wrong times:9725

for 10000tests, total runtime is: 17.743 ms
average time for one test is: 0.0017743 ms

for all 275 right results, 28 of them satisfy the condition

for all 9725 wrong results, 0 of them satisfy the condition
```

说明上述分析具有一定的合理性，得到了合理结论——获得正确结果的一个充分条件:
a*b 的结果除去高 52 位后，设剩下的低二进制位的位数为 lowbit，
只要满足条件 m>=2^(lowbit)，就可以得到正确结果。