

Lab1实验报告

高辰潇

人工智能学院

SID：181220014

任务1：实现multimod

- 我的实现的原理为将64位整数相乘转变为64次整数加法，并在每次相加的过程中累加计算结果。详情请见p1.c。
- 为了说明该实现的正确性，下面从两个方面给出证明

理论层面

我在p1中给出的实现的原理为，将 $a * b$ 分解为 $a_0 * b * 2^0 + a_1 * b * 2^1 + \dots + a_{62} * b * 2^{62}$ 。由于在计算 $2^i * b$ 时实际上还是会导致溢出，因此需要寻找一个在同余意义上和 $2^i * b$ 相等、但却不会溢出64位整形表达能力的等价表达式。简单分析可知， $(2^i * b) \bmod m = ((\dots((b \bmod m) * 2) \bmod m \dots) * 2) \bmod m$ ，其中共顺次进行了 $i+1$ 次mod m操作， i 次 $\times 2$ 操作。因此可基于此原理设计迭代算法，计算 $2^i * b \bmod m$ 的余数。在p1中，每次取出 a 的最低位，如果是1，则计算 $2^i * b \bmod m$ 的余数，然后将它与ret（当前的计算结果）相加，并再次模m。循环结束后，得到的结果即为 $(a * b) \bmod m$ 。

```
int64_t multimod_p1(int64_t a, int64_t b, int64_t m) {
    uint64_t ret = 0;
    assert(m);
    b = b % m;
    if (a==0 || b==0) return 0;

    while(a>0) {
        if (a%2) {
            ret = (uint64_t)(ret + b)%m;
        }

        b = (uint64_t)(b*2)%m;
        a = a/2;
    }
    return ret;
}
```

下面讨论溢出问题。可能溢出的语句为if中的 $(ret + b)$ 和更新b时的 $b * 2$ 。由题目所给条件知， $0 \leq a, b, m \leq 2^{63} - 1$ ，由 ret 和 b 的更新公式知每次迭代开始时均有 $ret, b < m \leq 2^{63} - 1$ ，因此 $(ret + b) < 2m < 2^{64} - 1$ ， $b * 2 < 2m < 2^{64} - 1$ ，所以使用强制类型转换uint64_t即可避免溢出问题。

代码测试

在p1.c中定义宏 DEBUG，编译并运行 test.py 即可进行代码测试，测试代码见 test.py test.py中共进行了100000+63次测试。每次测试都是随机从 $[0, 2^{63} - 1]$ 中随机生成a，b。对于m，前63次测试中m取值为 $2^0, 2^1, \dots, 2^{62}$ ，后100000次测试中m取值为 $[0, 2^{63} - 1]$ 中的随机数。对于每一组输入，将p1的输出与python中的运算输出比对。

实验结果：对于每一组输入a,b,m，p1的结果均与python给出的正确答案吻合。

任务二：性能优化

- 我作出的优化为：将任务一中的乘除运算尽可能替换为移位操作.优化后代码见p2.c
- 首先测量p2实现的正确性。在p2.c中定义宏DEBUG，运行 test.py ，发现p2成功通过测试。
- 下面衡量性能差距

性能对比

- 使用time.h模块中的clock函数对multimod函数的性能进行度量，测量脚本为 timing.py .
- 首先在p1.c和p2.c中定义宏TIMING。 timing.py 中会生成1000000条样例输入写入文件 timing.txt ，而p1和p2将会为这些样例输入计算输出，并打印计算所耗费的总时间。 timing.py 接收到总时间后，计算平均时间作为性能度量，实验结果如下：
(表格中每一项为执行一次multimod所需要的时间，单位为毫秒。)

	p1	p2
-O0	1.364525	1.367435
-O1	1.145668	1.127761
-O2	1.129744	1.103874

- 分析：可以发现，在更高的优化选项下p1和p2的性能有一定提升。横向对比p1和p2可发现二者的性能有一定差距，但并不明显。

任务三：分析神秘代码

```
int64_t multimod_p3(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;
}
```

分析

- 计算结果的正确性要求以下两个条件
 - 不能溢出，即 $(a * b - (int64_t)((double)a * b / m) * m)$ (*) 属于 $[0, 2^{63} - 1]$
 - 数值正确，即 $(a * b - (int64_t)((double)a * b / m) * m)$ 和 $(a * b)$ 模 m 同余。
- 显然，由于强制类型转换，条件2总是能被满足
- 对于条件1， $(int64_t)((double)a * b / m)$ 可看作 $(double)a * b / m$ 朝0方向的截断。
 - 为了表示上的方便，定义 $a * b / m$ 为 a 乘 b 除以 m 的精确计算结果， $/$ 可被理解为python中的除号。不妨设 $a * b / m$ 在数值上等于 $M.N$ ，即 M 为整数部分， N 为小数部分
 - 若 $(double)a*b$ 可以精确表示 $a * b$ ，
 - 则 $(a * b - (int64_t)((double)a * b / m) * m) = (M.N)*m - M*m = (0.N)*m$ 。由于 m 小于 $2^{63} - 1$ ， $0.N$ 小于1，因此(*)式小于 $2^{63} - 1$ ，条件1满足
 - 若 $(double)a*b$ 不能精确表示 $a * b$ ，且偏大，
 - 则 $(int64_t)((double)a * b / m) * m = M*m$ or $(M+1)*m$ 。因此(*)式 $= (0.N)*m$ or $(0.N)*m - m$ ，由于 $(0.N)*m - m$ 小于0，故当后一种情况出现时会发生下溢，条件1可能不被满足
 - 若 $(double)a*b$ 不能精确表示 $a * b$ ，且偏小，
 - 则 $(int64_t)((double)a * b / m) * m = M*m$ or $(M-1)*m$ 。因此(*)式 $= (0.N)*m$ or $(0.N)*m + m$ 。由于 $(0.N)*m + m$ 大于 $2^{63} - 1$ ，故当后一种情况出现时，可能会上溢，条件1可能不被满足。
- 综合上述分析可作出结论
 - 若 $(double)a*b$ 可以精确表示 $a * b$ ，则计算结果为准确值。为了能够精确表示， $a*b$ 的大小不能超过 $2^{53} - 1$ ，这样double类型的52位尾数+1位隐藏位才能精确表示整数部分。
 - 若 $(double)a*b$ 不能精确表示 $a * b$ ，则计算时**可能会**发生溢出，**可能**无法得到正确的结果。
- **注：**上述分析中我仅考虑了 $(double)a * b$ 能否精确表示 $a * b$ ，而并未考虑 $(double)a * b / m$ 中相除后使用double致使小数部分被舍入而带来的误差。这是因为：由于 $a * b$ 在数值上可能很大，而double类型在数值较大时“表示能力”很弱，即能表示的数之间间隔很大，这些“间隔”是导致不精确表示的主要因素。相较于做除法时的小数舍入， $a*b$ 时的整数迁移实际上更加宏观。

性能对比

- 测试方法同上，结果如下。

	p1	p2	p3
-O0	1.279005	1.225306	0.404211
-O1	1.156583	1.135988	0.379488
-O2	1.094874	1.102756	0.375445