《现代密码学》实验报告

实验名称： 伪随机数生成器	**实验时间：** 2023 年 12 月 27 日
学生姓名： 刘天健	**学号：** 21307379
学生班级： 21 保密管理	**成绩评定：**

一、实验目的

实现一个伪随机数产生器，以提升对于随机数生成算法的了解和代码实现能力。

二、实验内容

使用 C++ 编写一个随机数生成器，其从 stdin 读取输入，前三行为 p,g,s 的值，第四行为要生成的随机比特数 n。将生成的随机比特，0 和 1 的个数输出到 stdou。其中，输入参数 p 的数据类型为 uint64。n<=262144。单次运行时间限制为 15s。

在此次实验中，需要自行实现 128bit 的数据类型定义以及乘法，取模和模逆运算，同时不允许使用大数库等和 128bit 相关的数据。

最终，在周一晚上的程序实现了前 5 道样例的正确，但之后的样例由于超时没有通过。在实验步骤中，自己在（2）中说明了源程序存在的问题以及优化的过程。

三、实验原理

对于参数 p,g,s,p 是一个素数，g 是 Z_p 的一个生成元，种子 s 是 Z_p 的任意元素。该随机数生成器重复 n 轮，每轮循环中，计算 $s_{i+1} = g^{s_i} \bmod p$，输出随机比特:

$$z_i = \begin{cases} 0, & s_i < \frac{p}{2} \\ 1, & s_i \geq \frac{p}{2} \end{cases}$$

图 1: 生成规则

最后，这些随机比特组合起来就是我们的生成的随机数。

在此次实验中，需要实现 128 位数字的减法，比较和取模，也需要实现 64 比特数字的乘法（因为直接相乘可能会溢出），对于 64 比特数字的乘法，可以采用分解移位的方式去获得高 64 位的数字。例如 a 和 b 相乘，采用 lo 和 hi 两个数去分别表示 a 和 b 的高 32 位和低 32 位，然后分为不同的部分（高 × 高，高 × 低，低 × 低）去进行计算，最后将各部分进行移位相加，就得到了高 64 位的结果。

而对于取模（128 比特模 64 比特），可以采用俄罗斯农民乘法的除法方式去进行运算。该算法的伪代码如下所示：

```
X = B;

while (X <= A/2)
{
    X <<= 1;
}

while (A >= B)
{
    if (A >= X)
        A -= X;
    X >>= 1;
}
```

图 2: 取模算法

对于 128 位数字的减法，可以将其转化为高 64 位和低 64 位的计算，对于低 64 位，可以直接相减，高 64 位则在相减的时候需要考虑低 64 位减法的进位。

四、实验步骤（源代码）

在自己的源程序中，我通过了 vector<uint64-t> 的方法来表示一个 128bit 的数，其中，向量第 0 位表示高 64bit，向量第 1 位表示低 64bit。对于两个 64bit 数的相乘，可以通过简单的 a*b 得到低 64bit，但溢出的高 64bit 可以通过分解移位的方式来实现：

```
1  uint64_t mul_high(uint64_t a, uint64_t b) {
2      uint64_t a_lo = (uint32_t)a;
3      uint64_t a_hi = a >> 32;
4      uint64_t b_lo = (uint32_t)b;
5      uint64_t b_hi = b >> 32;
6
7      uint64_t a_x_b_hi = a_hi * b_hi;
8      uint64_t a_x_b_mid = a_hi * b_lo;
9      uint64_t b_x_a_mid = b_hi * a_lo;
10     uint64_t a_x_b_lo = a_lo * b_lo;
11
12     uint64_t carry_bit = ((uint64_t)(uint32_t)a_x_b_mid +
       ↪  (uint64_t)(uint32_t)b_x_a_mid + (a_x_b_lo >> 32)) >> 32;
13     uint64_t multhi = a_x_b_hi + (a_x_b_mid >> 32) + (b_x_a_mid >> 32) + carry_bit;
14     return multhi;
15 }
16 uint64_t mul(uint64_t a, uint64_t b, uint64_t c) {
17     uint64_t lo_result = a * b;
18     uint64_t hi_result = mul_high(a, b);
19     vector<uint64_t>result;
20     result.push_back(hi_result);^^I
21     result.push_back(lo_result);
22     uint64_t answer = mod(result, c);
23     return answer;
24 }
```

然后，对于取模运算，这里让一个 128bit 的数去模 64bit 的数，因此可以采用俄罗斯农民的除法方式去计算。具体代码如下所示：

```
uint64_t mod(vector<uint64_t>s, uint64_t p) {
    uint64_t x = p;
    vector<uint64_t>judge_x;
    vector<uint64_t>judge_p;
    judge_x.push_back(0);
    judge_x.push_back(x);
    judge_p.push_back(0);
    judge_p.push_back(p);
    while (compare(right_move_one(s), judge_x)) {
        judge_x = left_move_one(judge_x);
    }
    while (compare(s, judge_p)) {
        if (compare(s, judge_x)) {
            s = sub(s, judge_x);
        }
        judge_x = right_move_one(judge_x);
    }
    return s[1];
}
```

这其中涉及到一些运算，具体实现代码如下所示。对于移位，这里我采用了转为 bitset 后左移右移的方法（其实这样转换是多余的，但是当时自己感觉这样写起来比较直观所以还是转换了一次）。而对于减法，自己也是采用了转为 bitset 后每位进行相减，然后生成对应的结果和进位（这样对位进行的做法也是低效的，在考试后自己也进行了优化）。

```
bool compare(vector<uint64_t>a, vector<uint64_t>b) {
    if (a[0] > b[0]) {
        return true;
    }
    else if (a[0] < b[0]) {
        return false;
    }
    else if (a[1] >= b[1]) {
        return true;
    }
    else {
        return false;
    }
}

vector<uint64_t>left_move_one(vector<uint64_t>p) {
    bitset<64>bit_hi = p[0];
    bitset<64>bit_lo = p[1];
    bit_hi <<= 1;
    bit_hi[0] = bit_lo[63];
    bit_lo <<= 1;
    p[0] = bit_hi.to_ullong();
    p[1] = bit_lo.to_ullong();
    return p;
}

vector<uint64_t>right_move_one(vector<uint64_t>p) {
    bitset<64>bit_hi = p[0];
    bitset<64>bit_lo = p[1];
    bit_lo >>= 1;
    bit_lo[63] = bit_hi[0];
    bit_hi >>= 1;
    p[0] = bit_hi.to_ullong();
    p[1] = bit_lo.to_ullong();
```

```
    return p;
}

vector<uint64_t>sub(vector<uint64_t>a, vector<uint64_t>b) {
    bitset<128>a_hi = a[0];
    bitset<128>a_lo = a[1];
    bitset<128>bit_a = (a_hi << 64) ^ a_lo;
    bitset<128>b_hi = b[0];
    bitset<128>b_lo = b[1];
    bitset<128>bit_b = (b_hi << 64) ^ b_lo;
    bitset<128> result;
    bool borrow = false;
    for (size_t i = 0; i < 128; ++i) {
        bool a_bit = bit_a[i];
        bool b_bit = bit_b[i];
        if (a_bit == b_bit) {
            result[i] = borrow;
        }
        else
        {
            result[i] = !borrow;
        }
            borrow = (b_bit && !a_bit) || (borrow && !a_bit) || (borrow && b_bit);
        }
        vector<uint64_t>p;
        p.push_back((result >> 64).to_ullong());
        p.push_back(((result << 64) >> 64).to_ullong());
        return p;
}
```

要实现模次方，我们可以采用快速幂的方式进行运算即可：

```
uint64_t fun(uint64_t a, uint64_t b, uint64_t c) {
    if (b == 0) {
        uint64_t answer = 1;
        return answer;
        }
    else if (b % 2 == 1) {
        uint64_t temp = fun(a, (b - 1) / 2, c);
        temp = mul(temp, temp, c);
        a %= c;
        return mul(a, temp, c);
        }
    else {
        uint64_t temp = fun(a, b / 2, c);
        return mul(temp, temp, c);
    }
}
```

最后，我们在主函数中实现输入输出，对于输出，我们采用一个 string 储存每次循环生成的 0 和 1，并每次生成 0 时，count 也加 1，最终输出结果和 0，1 的个数：

```
int main() {
    uint64_t p;
    cin >> p;
    uint64_t g;
    cin >> g;
    uint64_t s;
    cin >> s;
    uint64_t n;
    cin >> n;
    string answer;
    int count = 0;
    for (uint64_t i = 0; i < n; ++i) {
```

```
            s = fun(g, s, p);
            if (s < p / 2) {
                answer += "0";
                count++;
            } else {
                answer += "1";
            }
        }
        cout << answer << endl;
        cout << count << endl;
        cout << n - count << endl;
    }
```

五、实验结果

从结果来看，自己的程序可以生成正确的内容，然而，在生成超过 10000 比特的数字时，其就已经需要超过 15s。例如在下图中，生成 20000 比特的数字需要 30s，然而，本次实验限制在 15s 内，而且最长的甚至超过了 26w 比特，因此，这样的程序在性能上是不佳的。

```
01111011111001010101101000110010000101100110011000111010100111010111111001101110000000110100101010111010101001000011100
11010111100101101010001000111101110110111001101101010011101110111001011001100110110000110001000001101011111010111000100
01010000101001110001111101101001001110000001101101111010011100000111110011000101011100101111111111000100111010001101000111
01011101011011111000110010001100010011101110001001010101110111011011010110100101001
9872
10128

--------------------------------
Process exited after 33 seconds with return value 0
请按任意键继续. . .
```

图 3: 2w 比特需要的时间

为什么会超时？经过调试后，自己发现主要是 vector 的使用和 sub 的设计导致了运算的低效：由于 vector 是动态长度的，但自己只需要使用前两位，所以其实额外开销是非常大的。经过测试，自己发现 vector 是造成超时的主要原因，将其改为结构体后（结构体储存两个 uint64 的数字），其运行速度由 30s 变为了 11s。

```
00100001000100001111111101011110010000010110101100000011010000111110010011010001000011110001101110011100111011111000010011
11110110000000101001000110000111100011011010101110011001111100100111000111011011101100100010001110001101100100100111100
10001011000110000010001011001011010000001101111000010000101100001111101110000101110001001110101001101101110100001011100
01111011111001010101101000110010000101100110011000111010100111010111111001101110000000110100101010111010101001000011100
11010111100101101010001000111101110110111001101101010011101110111001011001100110110000110001000001101011111010111000100
01010000101001110001111101101001001110000001101101111010011100000111110011000101011100101111111111000100111010001101000111
01011101011011111000110010001100010011101110001001010101110111011011010110100101001
9872
10128

--------------------------------
Process exited after 11.29 seconds with return value 0
请按任意键继续. . .
```

图 4: vector 变为结构体后 2w 比特需要的时间

造成超时的另一个原因是减法只是对每位进行运算，再改为让两个 uint64 相减的方法后速度也有较大的提升，可以由 11s 变为 1.3s。

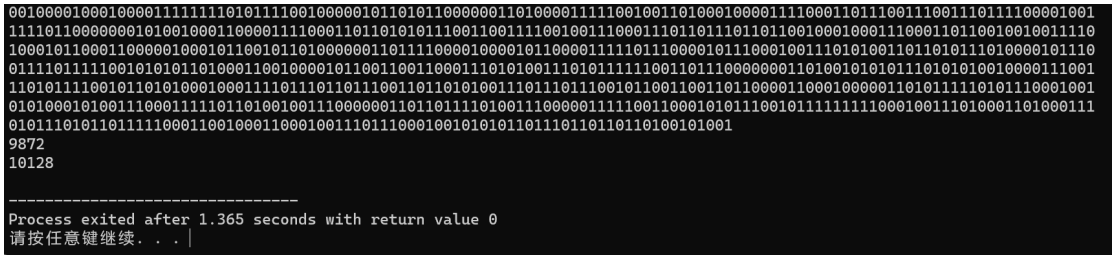

图 5: 优化减法后 2w 的需要时间

最终，优化后的程序输入 p=215944803539, g=167919746238, s=2147483647,n=262144（最长的长度），这里 p,g,s 采用的是样例四的数据（因为只是希望计算其在高位 n 中的速度，所以没有选择其他的 p,g,s，只是选取了最长的 n），结果如下所示，可以看到，此时只需要 8s 即可产生，在最开始的 10s 要求内。

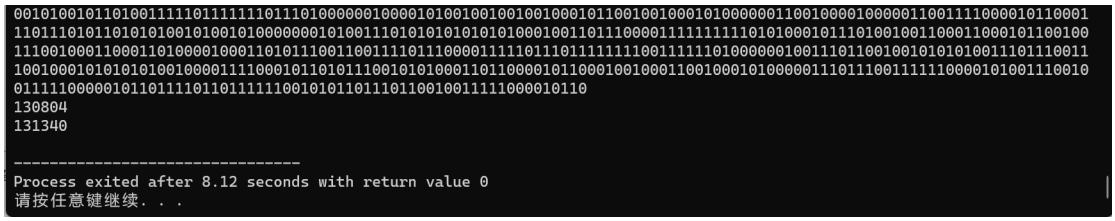

图 6: 优化后的 26w 长度计算结果

六、实验总结

本次实验实现了一个伪随机数生成器，算法较为简单，自己虽然实现了算法但在性能上没有过关，因此也比较遗憾。刚结束的时候还是有些憋屈和无奈的感受，但是在经过分析修正性能问题后，还是感到豁然开朗（鉴定为写 STL 写的.JPG）。也感谢助教的帮助。