实验设备：
12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz
实验环境：
C++：Visual Studio 2022
python：PyCharm 2022.2.3

Project1 实现降约 SM3 的幼稚生日攻击

哈希碰撞

所谓哈希（hash），就是将不同的输入映射成独一无二的、固定长度的值（又称"哈希值"）。它是最常见的软件运算之一。如果不同的输入得到了同一个哈希值，就发生了"哈希碰撞"（collision）。

生日攻击原理

哈希碰撞的概率取决于两个因素（假设哈希函数是可靠的，每个值的生成概率都相同）。

这个问题在数学上早有原型，叫做"生日问题"（birthday problem）：一个班级需要有多少人，才能保证每个同学的生日都不一样？

答案很出人意料。如果至少两个同学生日相同的概率不超过 5%，那么这个班只能有 7 个人。事实上，一个 23 人的班级有 50%的概率，至少两个同学生日相同；50 人班级有 97%的概率，70 人的班级则是 99.9%的概率（计算方法见后文）。

这意味着，如果哈希值的取值空间是 365，只要计算 23 个哈希值，就有 50%的可能产生碰撞。也就是说，哈希碰撞的可能性，远比想象的高。实际上，有一个近似的公式。

上面公式可以算出，50% 的哈希碰撞概率所需要的计算次数，N 表示哈希的取值空间。生日问题的 N 就是 365，算出来是 23.9。这个公式告诉我们，哈希碰撞所需耗费的计算次数，跟取值空间的平方根是一个数量级。

这种利用哈希空间不足够大，而制造碰撞的攻击方法，就被称为生日攻击（birthday attack）。

哈希碰撞公式

上面的公式，可以进一步推导成一般性的、便于计算的形式。

根据泰勒公式，指数函数 ex 可以用多项式展开。

$$\exp(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \cdots$$

如果 x 是一个极小的值，那么上面的公式近似等于下面的形式。

$$e^x \approx 1 + x$$

现在把生日问题的 1/365 代入。

$$e^{-\frac{1}{365}} \approx 1 - \frac{1}{365}.$$

因此，生日问题的概率公式，变成下面这样。

$$\begin{aligned}\bar{p}(n) &\approx 1 \cdot e^{-\frac{1}{365}} \cdot e^{-\frac{2}{365}} \cdot e^{-\frac{n-1}{365}} \\ &= e^{-\frac{1+2+\cdots+(n-1)}{365}} \\ &= e^{-\frac{n(n-1)/2}{365}} = e^{-\frac{n(n-1)}{730}}.\end{aligned}$$

$$p(n) = 1 - \bar{p}(n) \approx 1 - e^{-\frac{n(n-1)}{730}}.$$

假设 d 为取值空间（生日问题里是 365），就得到了一般化公式。

$$p(n,d) \approx 1 - e^{\frac{-n(n-1)}{2d}}$$

上面就是哈希碰撞概率的公式。

Project2 实施还原 SM3 的 Rho 方法

问题模型

给一个数 n，你需要快速求出它的一个非平凡因子。

对于一个比较小的数（$n \leq 10^9$），我们直接暴力枚举质因子就行但太大了我们就必须考虑一下随机算法了。

对于一个非常大的合数 $n \leq 10^{18}$（如果可能是质数，我们可以用Miller Rabin判一下）我们要求 n 的某一个非平凡因子，如果 n 的质因子很多（就是约数很多）我们也可以暴力随机弄一弄，但如果是一些（像由两个差不多的而且很大的质数乘得的 n）它的非平凡因子就只有两个而且 n 本身还很大，此时如果我们随机的话复杂度 $O(n)$，这个太难接受了，所以我们想办法优化一下。

求最大公因子 GCD

如果我们直接随机求 n 的某一个约数复杂度很高，我们可以考虑求一下 gcd 。因为我们可以保证一个合数它绝对存在一个质因子小于 $\sqrt{n}$ ，所以在 n 就存在至少 $\sqrt{n}$ 个数与 n 有大于一的公约数。于是我们随机的可能性就提高到了 $O(\sqrt{n})$ 。

玄学随机法（平方加常数——x^2+c）

其实网上大多都叫这种方法为生日悖论（悖论是不符合经验但符合事实的结论)，但个人觉得这个方法跟我们的 PollardRho 的关联。在 1 中我们用 gcd 的方案，现在我们考虑有没有办法能够快速的找到这些与 n 有公约数的数（或者说找到一种随机生成方法，使随机生成这类我们需要的数的概率变大一点)。
这个随机生成方法最终被 Pollard 研发出来了:就是标题那个式子!我们定义
(y = x　x = x * x + c　g = ged(y - x,p))为一次操作，我们发现 g >1(就是找到我们需要的数)的几率出奇的高，根据国际上大佬的疯狂测试，在 n 中找到一对 y - x 使两者有公约数的概率接近 $n^{1/4}$.
判环
这种随机生成方法虽然优秀，但也有一些需要注意之处，比如有可能会生成一个环，并不断在这个环上生成以前生成过一次的数，所以我们必须写点东西来判环:
1.我们可以让 y 根据生成公式以比 x 快两倍的速度向后生成，这样当 y 再次与 x 相等时，x 一定跑完了一个圈且没重复跑多少!
2.我们可以用倍增的思想，让 y 记住 x 的位置，然后 x 再跑当前跑过次数的一倍的次数。这样不断让 y 记住 x 的位置，x 再往下跑，因为倍增所以当 x 跑到 y 时，已经跑完一个圈，并且也不会多跑太多(这个必须好好想一想，也可以看看代码如何实现的)(因为这种方法会比第一种用的更多，因为它在 PollardRho 二次优化时还可以起到第二个作用)

Project3 实现 SM3、SHA256 等的长度扩展攻击。

长度扩展攻击（length extension attack），是指针对某些允许包含额外信息的加密散列函数的攻击手段。对于满足以下条件的散列函数，都可以作为攻击对象：
① 加密前将待加密的明文按一定规则填充到固定长度（例如 512 或 1024 比特）的倍数；
② 按照该固定长度，将明文分块加密，并用前一个块的加密结果，作为下一块加密的初始向量（Initial Vector）。

SHA256 原理将算法中可以单独分出的模块，包括常量的初始化、信息预处理、使用到的逻辑运算。
信息的预处理分为两个步骤：附加填充比特和附加长度
附加填充比特
在报文末尾进行填充，使报文长度在对 512 取模以后的余数是 448
填充是这样进行的：先补第一个比特为 1，然后都补 0，直到长度满足对 512 取模后余数是 448。
需要注意的是，信息必须进行填充，也就是说，即使长度已经满足对 512 取模后余数是 448，

补位也必须要进行，这时要填充 512 个比特。
因此，填充是至少补一位，最多补 512 位。
例：以信息“abc”为例显示补位的过程。
a,b,c 对应的 ASCII 码分别是 97,98,99
于是原始信息的二进制编码为：01100001 01100010 01100011
补位第一步，首先补一个“1” ： 0110000101100010 01100011 1
补位第二步,补 423 个“0”：01100001 01100010 01100011 10000000 00000000 … 00000000
附加长度值
附加长度值就是将原始数据（第一步填充前的消息）的长度信息补到已经进行了填充操作的消息后面

SHA-256 长度扩展攻击是一种针对哈希函数的攻击方式。攻击者利用哈希函数的性质，在不知道输入数据具体内容的情况下，能够构造出另一个与原始哈希值相同的新哈希值，并且在新哈希值后面添加任意数据内容，而不影响新哈希值的正确性。攻击者可以通过这种方式来伪造数据，或者绕过哈希值校验。
具体来说，SHA-256 长度扩展攻击的原理是：攻击者知道原始哈希值 H 和输入数据 M，他可以通过构造一个新的数据块 M'，然后计算出新的哈希值 H'，满足 H' = SHA-256(M'||P)，其中 P 是攻击者自己添加的任意数据。攻击者不需要知道 M 的具体内容，只需要知道 M 的长度即可进行攻击。
为了防止 SHA-256 长度扩展攻击，可以采用一些特定的哈希函数设计方案，如 HMAC（Hash-based Message Authentication Code）或者使用 SHA-3 等其他安全性更高的哈希函数。此外，在实际应用中，可以对输入数据进行必要的处理和校验，避免受到长度扩展攻击的影响。