《密码学》课程设计实验报告

***本次实验源文件和实验报告均已上传至 PRNG_and_TRNG 文件夹下：**

sterzhang/Cryptology: Jianshu Zhang's cryptological experiments (github.com)

实验序号：附加题 1　　实验项目名称：实现伪随机素数生成算法

学　　号	2021302181216	姓　　名	张鉴殊	专业、班	21 信安
实验地点	NCC	指导教师	余荣威	时间	2023.12.17

一、实验目的及要求

实现伪随机素数生成算法，包括概率性算法和确定性算法

二、实验设备（环境）及要求

Windows 操作系统，高级语言开发环境（python）

三、实验内容与步骤

随机数的生成的相关概念：

有两种不同的策略来生成随机数，真随机数生成器(True Random Number Generator，TRNG)和伪随机数生成器(Pseudo-Random Number Generator，PRNG).

在密码学中，生成随机位流是一个重要的密码函数。大量基于密码学的网络安全算法和协议都使用随机二进制数，用于密钥分发、互相认证、生成会话密钥等。

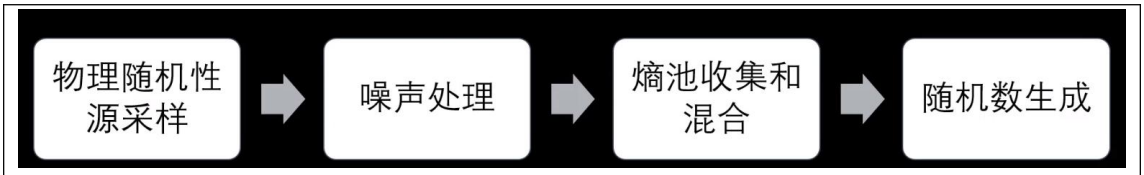
对生成随机数的有两个要求:随机性和不可预测性。

随机性:在某种明确定义的统计意义下，数序列是随机的。包含分布均匀性，即序列中的位分布应是均匀的，即 0 和 1 出现的概率相等。以及独立性:，即序列中的任何子序列都不能由其他子序列推导出来。

不可预测性：序列后续成员不可预测。

真随机数生成器(True Random Number Generator，TRNG)

基本思路是使用某种不确定的物理源生成非确定的随机位，这里我选用的是利用物理世界的图片的不确定性来生成非确定的随机位。



具体实现如下：

1）导入相关实现的包与初始化：

```
import imageio
import numpy
import math
from itertools import chain
from moviepy.editor import *

# 图像数据存储格式
# image[y][x][rgb]
#    y: 行坐标
#    x: 列坐标
#    rgb: 颜色通道
```

```
numNeeded = 10000  # 所需随机数的数量
maxRange  = 256    # 数值范围的最大值
```

```
# 初始化变量
numPossibles = 0  # 可能的数值数量
schouldCont = True  # 控制循环是否继续的标志
frame = 0  # 当前处理的帧数
bitsRange = math.ceil(math.log(maxRange, 2))  # 计算所需位数
tmp = 0  # 临时变量，用于构建最终的数值
count = 0  # 计数器
sublist = []  # 存储单个位
tempSublist = []  # 存储临时子列表
outputList = []  # 输出的数值列表
```

2）从视频中截取帧作为输入源：

```
# 从视频中截取帧
def frameCut():
  clip = VideoFileClip('example.mp4')  # 打开视频文件
  clip.save_frame('frame.png', t=frame)  # 保存当前帧为 PNG 图片
```

3）将 numNeeded 的原始值将乘以 bitsRange 的值，然后新的结果将再次赋给 numNeeded 所表示的所需随机数数量，而 bitsRange 是基于 maxRange 计算出的所需的位数。调整 numNeeded 的值，以确保拥有足够的单个位来生成所需数量的随机数：

```
# 从帧中读取数值
numNeeded *= bitsRange
```

4）遍历整个帧，提取位值：

```
while(schouldCont):
  frameCut()
  image = imageio.imread('frame.png')  # 读取图片
  imgHigth, imgWidth, imgChannel = image.shape  # 获取图片尺寸和通道信息

  # 遍历整个帧，提取位值
  for i in range(0,imgHigth):
    for j in range(0,imgWidth):
      for k in range(0,imgChannel):
        # 只考虑特定范围内的像素值
        if(image[i][j][k] >= 2 and image[i][j][k] <= 253):
          sublist.append(image[i][j][k] & 0b1)  # 提取最低位
          numPossibles += 1
  frame += 1
  if(numPossibles >= numNeeded):
    schouldCont = False
```

5）将提取的位值（即二进制的 0 和 1）组织成一个正方形矩阵，并对该矩阵进行转置和重新排列，最后再根据需要将额外的位值添加到列表中。首先使得矩阵中的元素数量（即位值的数量）接近于所需的位值总数 numNeeded，再开始填充矩阵，然后对这个矩阵进行转置与重新排列，并使用 chain.from_iterable 方法将其展平成一个一维列表。这一步重新排列了位值的顺序，增加生成随机数的随机性：

```
# 计算正方形矩阵的大小
square = math.floor(math.sqrt(numNeeded))
# 将提取的位值填充到矩阵中
for i in range(0,square*square):
  tempSublist.append(sublist[i])
# 转置并重新排列矩阵
tempSublist = numpy.array(tempSublist).reshape(square, square)
transpose = tempSublist.T
tempSublist = transpose.tolist()
tempSublist = list(chain.from_iterable(tempSublist))
# 如果所需位数大于矩阵大小，则继续添加
if(square*square < numNeeded):
  for i in range(0, numNeeded - square*square):
    tempSublist.append(sublist[i+(square*square)])
```

6）将从视频帧中提取的随机位值组合成完整的数值，并将这些数值保存到一个文本文件中：

```
x = 0
# 组合位值生成数值
while(count < numNeeded/bitsRange):
  for j in range(0, bitsRange):
    valTmp = tempSublist[x]
    tmp = tmp | (valTmp << j)
    x += 1
  outputList.append(tmp)
  tmp = 0
  count += 1

# 将生成的数值写入文件
output = open("output.txt", "w")
for element in outputList:
    output.write(str(element) + "\n")
output.close()
```

伪随机数生成器(Pseudo-Random Number Generator, PRNG)

方法一：线性同余生成器

线性同余生成器（Linear Congruential Generator，LCG）是一种用于生成伪随机数序列的算法。其工作原理基于一个简单的数学公式：Xn+1 =(aXn +c)mod m

LCG 的特点和性能主要取决于参数 a，c 和 m 的选择。

为了获得良好的随机性质和较长的周期，这些参数需要仔细选择。一个有效的 LCG 应该满足能够生成长度为 m 的完整周期，即在重复之前生成 m 个不同的伪随机数。

要实现这一点必须满足：

c 和 m 互质；

a-1 能被 m 的所有质因数整除；

如果 m 是 4 的倍数，那么 a-1 也必须是 4 的倍数。

具体实现步骤如下：

1）在 main 函数中自己定义参数 a，c 和 m，以及调用 LCG 函数：

```
if __name__ == '__main__':
    print("Xi = (a*X_i-1 + c)mod m")
    a = int(input('Enter multiplier \'a\': '))
    c = int(input('Enter increment \'c\': '))
    m = int(input('Enter modulus param \'m\': '))
    x0 = random.randint(0, m - 1)  # 随机生成初始值
```

```
    if m < 0 or a >= m or a <= 0 or c >= m or c < 0:
        print('Wrong Input, Please Follow the constraints')
    else:
        lcg(x0, a, c, m)  # 调用线性同余生成器
```

2）在 LCG 函数中，首先先检查 LCG 的全周期条件，这里检查的是我上述理论部分中写的，要使 LCG 应该满足能够生成长度为 m 的完整周期必须满足：c 和 m 互质；a-1 能被 m 的所有质因数整除；如果 m 是 4 的倍数，那么 a-1 也必须是 4 的倍数：

```
# 线性同余生成器（Linear Congruential Generator，LCG）
def lcg(x0, a, c, m):
    full = False  # 用于标记周期是否为完整的 m
    # 检查 LCG 的全周期条件
    if math.gcd(c, m) == 1 and divisibile(a - 1, m) and ((m % 4 == 0) and
((a - 1) % m == 0)):
        full = True
```

这里涉及另外两个函数的调用：

```
# 定义一个函数来找到一个数的所有质因数
def prime_factors(n):
    factors = []  # 存储质因数的列表
    # 首先处理所有的 2，使 n 成为奇数
    while n % 2 == 0:
        factors.append(2)
        n = n / 2
    # 检查所有可能的奇数因数，直到 sqrt(n)
    for i in range(3, int(math.sqrt(n)) + 1, 2):
        while n % i == 0:
            factors.append(i)
            n = n / i
    # 如果 n 大于 2，那么 n 本身是一个质数
    if n > 2:
        factors.append(n)
    return factors
```

```
# 定义一个函数来判断 a 是否可以被 b 的所有质因数整除
def divisibile(a, b):
    factors = prime_factors(b)  # 获取 b 的质因数列表
    for i in factors:
        if a % i != 0:
            return False
    return True
```

3）接下来就正常进入随机数生成的步骤：

```
    prandomnums = [0] * m  # 初始化随机数数组
```

```
    prandomnums[0] = x0  # 设置初始值
    print(prandomnums[0], end=" ")
    # 生成随机数序列
    for i in range(1, m):
        prandomnums[i] = ((prandomnums[i - 1] * a) + c) % m
        print(prandomnums[i], end=" ")

    count = 1  # 计数器，用于计算周期长度
    # 判断周期长度
    if full:
        count = m
    else:
        for i in range(1, m):
            if prandomnums[i] != x0:
                count += 1
            else:
                break

    print('\nCycle Length: ', count)
```

方法二：Blum-Blum-Shub 算法

Blum Blum Shub（BBS）算法是一种伪随机数生成器（PRNG），其工作原理基于模平方运算的性质。BBS 算法的安全性基于模平方运算的难逆性，特别是当模数 n 是两个大素数的乘积时。要从生成的序列中恢复初始种子或预测未来的位，攻击者需要能够有效地进行模 n 的因式分解，这是一个十分困难的问题。

以下是 BBS 算法的基本步骤：

首先选择两个大的质数 p 和 q，其中 p 和 q 都应该满足 $p \equiv q \equiv 3 \mod 4$。当被 4 除时，p 和 q 都应该有余数 3。

接着计算模数 n=p×q。这个乘积 n 将用作后续计算的模数。

再选择一个随机种子 x0 ，其中 x0 与 n 互质

通过以下迭代过程生成每一个伪随机位，计算 x_i+1=x_iˆ2 modn。产生一个位 bi 作为输出，通常是 x_i+1 的最低有效位。

重复上述过程，每次迭代产生一个新的伪随机位，从而生成一个伪随机位序列。

接下来是具体实现：

1）定义两个大素数，并求出乘积作为模数 n，在生成随机种子，并保证其与 n 互质：

```
p = 1000003
q = 2001911
n = p * q  # 计算两个大素数的乘积，作为模数 n
```

```
seed = randint(1, 10)  # 随机选择一个种子值

# 确保选定的随机种子与 n 互质（最大公约数为 1）
while gcd(seed, n) != 1:
    seed = randint(1, 10)
```

2）进入迭代产生随机位，这里用模 2 来取得 0 或 1 的 bits：

```
bits = str(seed % 2)  # 计算初始种子的最低有效位，并转换为字符串
for _ in range(1, 10000):
    seed = (seed * seed) % n  # 使用平方取模算法更新种子
    bit = seed % 2  # 计算新种子的最低有效位
    bits += str(bit)  # 将新计算的位连接到位串上
```

3）至此生成部分已经全部做完，但是毕竟是随机数生成，必须要有个方式去评判是否做到了随机，下面执行了两种统计分析。首先是计算长度为 1000 的子序列中 0 的平均数量，若平均值接近 500（即 1000 的一半）通常表明较好的随机性。以及统计长度为 4 的子序列出现的频率，长度为 4 的子序列的出现频率有助于了解短序列模式在整个序列中的分布情况。在一个理想的随机数序列中，所有可能的子序列应该有大致相同的出现频率，显著的偏差可能表明序列的随机性不足：

```
# 1) 计算长度为 1000 的子序列中 0 的平均数量
summation = 0
count = 0
for i in range(len(bits) - 1000):
    for j in range(i, i + 999):
        summation += bits[j].count("0")  # 统计子序列中 0 的数量
    count += 1
average = summation / count
print("The average number of zeros per subsequence: ", average)
print()
# 计算长度为 4 的所有可能子序列
subseqFour = [''.join(nums) for nums in product('01', repeat=4)]
freq = {}  # 存储长度为 4 的子序列频率的字典
for subseq in subseqFour:
    freq[subseq] = 0
```

```
# 2) 统计长度为 4 的子序列出现的频率
for i in range(len(bits) - 4):
    freq[bits[i:i + 4]] += 1
print("Frequency of length 4 subsequences:")
print("{:<12} {:<8}".format('Subsequence', 'Count'))
for subseq in freq:
    print("{:<12} {:<8}".format(subseq, freq[subseq]))
```

方法三：使用分组密码 CTR 和 OFB 加密模式生成伪随机数

这两个方法在之前都实现过，这里就不过多赘述，代码已经开源至 sterzhang/Cryptology: Jianshu Zhang's cryptological experiments (github.com)并且里面同时配有实验报告。在这就说一下为什么能用这两种方式生成伪随机数。

CTR（计数器模式）和 OFB（输出反馈模式）可以实现伪随机数生成器（PRNG）的原因有以下几点。

首先是块加密算法的伪随机特性，块加密算法 AES 在给定的密钥下会对输入的明文块产生一个看似随机的输出密文块。对于不同的输入明文块，在不更改密钥的情况下，块加密算法会产生不同的密文块。这种输出的高度不可预测性和唯一性使得块加密算法可以用作 PRNG。

再者就是两种加密模式的特性。CTR 模式使用一个计数器作为明文输入到块加密算法中。计数器每次迭代增加，保证了每次加密的输入都是唯一的。因此，输出密文块（伪随机数）在每次迭代中都是不同的，形成了一个伪随机序列。这种序列可以直接用其中的一部分用作伪随机数。OFB 模式将前一次的输出作为下一次的输入，形成一个反馈循环。这意味着只要初始向量（IV）或初始块是随机的，整个输出序列都将是随机的。OFB 模式产生的是一个不断更新的伪随机状态，这个状态的每次输出都可以用作伪随机数。

还有个重要的特点是它们都符合无错误传播，因为每个块的加密是独立的，所以错误不会在输出序列中传播。另外如果块加密算法本身是安全的，那么只要密钥和初始向量（或计数器）保持秘密，生成的伪随机数序列就是安全的。

在生成完随机数后，可以用以下两种方式检验素数（分为确定性和非确定性两种）。

确定性素数生成算法：AKS 素数检验算法

1. 首先给一个正整数 n>1

2. 如果 n 是某个数的整数次幂，那么直接返回“合数”

```
    def __is_integer__(n):
        i = gmpy2.mpz(n)
        f = n - i
        return not f
```

```
    # 1. 确定 n 是否是纯次幂
```

```
    for b in range(2, gmpy2.mpz(gmpy2.floor(gmpy2.log2(n)))+1):
        a = n**(1/b)
        if __is_integer__(a):
            return False
```

3. 找到最小的 r 使得 ord_r(n)> log^2(n)

```
    # 2. 找到一个最小的 r，符合 o_r(n) > (logn)^2
    maxk = gmpy2.mpz(gmpy2.floor(gmpy2.log2(n)**2))
    maxr = max(3, gmpy2.mpz(gmpy2.ceil(gmpy2.log2(n)**5)))
    nextR = True
    r = 0
    for r in range(2, maxr):
        if nextR == False:
            break
        nextR = False
        for k in range(1, maxk+1):
            if nextR == True:
                break
            nextR = (gmpy2.mpz(n**k % r) == 0) or (gmpy2.mpz(n**k % r) ==
1)
    r = r - 1 # 循环多增加了一层
    print("r = %d" % r)
```

4. 如果存在 a<=r 使得 gcd(a,n) != 1 or n，那么返回“合数”

```
    # 计算 a,b 的最大公约数
    def __gcd__(a, b):
        if b == 0:
            return a
        return __gcd__(gmpy2.mpz(b), gmpy2.mpz(a) % gmpy2.mpz(b))
```

```
    # 3. 如果存在 a<=r 使得 gcd(a,n) != 1 or n，那么返回“合数”
    for a in range(r, 1, -1):
        g = __gcd__(a, n)
        if g > 1 and g < n:
            return False
```

5. 如果 n<=r 那么返回“质数”

```
    # 4. 如果 n<= r，输出素数
    if n <= r:
        return True
```

6. 对于 a ∈[1，根号下 ϕ(r)乘以 log n]，判断(X+a)^n = (X^n +a)(mod X^r - 1 , n)是否恒成立。如果不是，那么返回“合数”，这里为了看整个计算的过程，将每一步运算都打印了下来：

```
    # 5. 对于 a ∈[1，根号下ϕ(r)乘以 log n]，判断(X+a)^n = (X^n +a)(mod X^r - 1 , n)是否恒成立。如果不是，那么返回“合数”

    # 构造 P = (X+a)^n mod (X^r-1)
    # 构造多项式(X+a)^n,并且进行二项式展开
    X = multi_ysymbols('X')
    a = multi_ysymbols('a')
    X_a_n_expand = binomial_expand(ypolynomial1(X, a), n)
    print(X_a_n_expand)
    X.pow(r)
    reduce_poly = ypolynomial1(X, ysymbol(value=-1.0))
    print("构造消减多项式 %s" % reduce_poly)
    print("进行运算 (X+a)^%d mod (X^%d-1)" % (n, r))
    r_equ = ypolynomial_mod(X_a_n_expand, reduce_poly)
    print("得到余式: %s" % r_equ)
    print("进行运算'余式' mod %d 得到式(A)" % n)
    A = ypolynomial_reduce(r_equ, n)
    print("A = %s" % A)
    print("B = x^%d+a mod x^%d-1" % (n, r))
    B = ypolynomial1(multi_ysymbols('X', power=31), a)
    B = ypolynomial_mod(B, reduce_poly)
    print("B = %s" % B)
    C = ypolynomial_sub(A, B)
    print("C = A - B = %s" % C)
    maxa = math.floor(math.sqrt(__phi__(r)) * math.log2(n))
    print("遍历 a = 1 to %d" % maxa)
    print("检查每个'%s = 0 (mod %d)'" % (C, n))
    for a in range(1, maxa+1):
        print("检查 a = %d" % a)
        C.set_variables_value(a=a)
        v = C.eval()
        if v % n != 0:
            return False
```

7.返回“质数”

```
    # 6. 返回“质数”
    return True
```

非确定性素数生成算法：Miller-Rabin 素性测试

1. 首先排除简单情况，如果 n 是 2，则它是素数；如果 n 是偶数或小于 2，它不是素数

```
    # 1. 首先排除简单情况：如果 n 是 2，则它是素数；如果 n 是偶数或小于 2，它不是素数
    if n == 2:
        return True
    if n & 1 == 0 or n < 2:
        return False
```

2. 将 n-1 分解为 2^r * d 的形式

```
    # 2. 将 n-1 分解为 2^r * d 的形式
    m, p = n - 1, 0
    while m & 1 == 0:
        m = m >> 1
        p += 1
```

3. 进行 s 次测试，多次随机选择一个数 a，并计算 a^m mod n。如果这个数不是 1 或 n-1，我们继续检查它的平方是否能得到 n-1。如果在任何点上我们得到了 n-1，那么 n 可能是一个素数；如果我们没有，那么我们可以几乎肯定 n 不是素数

```
    # 3. 进行 s 次测试，多次随机选择一个数 a，并计算 a^m mod n。如果这个数不是 1 或 n-1，我们继续检查它的平方是否能得到 n-1。如果在任何点上我们得到了 n-1，那么 n 可能是一个素数；如果我们没有，那么我们可以几乎肯定 n 不是素数。
    for _ in range(s):
        # 随机选择一个数 a
        a = fast_power(get_random(2, n - 1), m, n)
        if a == 1 or a == n - 1:
            continue
        for __ in range(p - 1):
            a = fast_power(a, 2, n)
            if a == n - 1:
                break
        else:
            # 如果没有找到 n-1，则 n 不是素数
            return False
```

```
    # 所有测试都没有证明 n 是合数，则返回 True（n 很可能是素数）;这个检验会进行 s 次，每次选择不同的随机数 a。如果在所有这些测试中，我们没有发现 n 是合数的证据，那么我们可以宣称 n 是一个素数，尽管如此，这是一个概率性的结论，而不是绝对的。
    return True
```

main 代码如下，这里我还对结果进行的验证，并通过打印错误类型来分析此算法的一个特点（素数一定能被判别出来）：

```
if __name__ == '__main__':
    num = 10000
    s = 3
    # 生成小于 num 的素数列表
    prime = [x for x in range(2, num) if not [y for y in range(2, int(np.sqrt(x) + 1)) if x % y == 0]]
    result = []
    for i in range(num):
        flag = Miller_Rabin(i, s)
        # 验证米勒-拉宾测试结果与已知素数列表是否一致;对生成的素数列表使用米勒-拉宾检验。

        #如果 Miller_Rabin 函数认为 i 是素数（flag 为真），但 i 实际上不在通过传统方法得到的素数列表 prime 中，说明 Miller_Rabin 函数错误地将一个合数判断为素数。这种错误称为"假阳性"，并打印出错误信息和相应的数 i。
        if flag and i not in prime:
            print('错误地将一个合数判断为素数: %d' % i)
        #如果 Miller_Rabin 函数认为 i 不是素数（flag 为假），但 i 实际上存在于通过传统方法得到的素数列表 prime 中，这意味着 Miller_Rabin 函数错误地将一个素数判断为合数。这种错误称为"假阴性"，同样会打印出错误信息和相应的数 i
        elif not flag and i in prime:
            print('错误地将一个素数判断为合数: %d' % i)
```

四、分析与讨论

在上述的素数筛选的实验过程中我主要是侧重将算法的每一步对应实现出来，并没有像生成随机数那部分过多的去分析具体的细节，现在我将开始探讨在素性筛选的实验过程中我对一些问题的理解和思考。

一，Miller-Rabin 素性测试和 AKS 素数检测算法在实现原理、效率、准确性和应用场景上的差异有什么？

首先，**Miller-Rabin 素性测试**是一种**概率性**算法，它的核心思想**基于费马小定理的扩展**。这个算法对于给定的数 n，通过随机选取多个小于 n 的数 a，检验 a^(n-1)是否等于 1（在模 n 的情况下）。如果某个 a 不满足这一条件，n 就被判定为非素数。然而，即使所有选定的 a 都满足条件，n 也只能被视为“可能是素数”。这种方法的优势在于它的**高效性**，特别是对于大数的素性测试，如之前做的加密算法中的素数生成的场景。另外，它的概率性虽然意味着一定的错误概率，但是通过增加测试的轮数可以显著降低这种风险。

而 **AKS 素数检测算法**是一种**确定性**算法，它提供了无误差的精确判断。该算法基于**费马小定理**的一个推广和**多项式同余**的概念。它通过验证对于所有小于等于 n 的整数 a，是否(X+a)^ n 等于 X^n + a（模 X^r - 1, n）来确定 n 是否为素数。这里的 r 是一个通过特定条件选定的整数。AKS 算法的重要性主要是在于它是第一个被证明可以在多项式时间内完成素数检测的算法。但是在处理大数时，AKS 算法的效率远低于 Miller-Rabin 测试，在本次实验中数字只要一大，程序执行会变得非常非常久。

二，为什么 Miller-Rabin 素性测试只会产生单向的错误？

在本次我的 Miller-Rabin 素性测试代码实现中，我特意在最后去做了一个验证性的脚本，可以看出只要产生错误就一定是合数被判定为“可能是素数”的情况，而绝对不会出现素数被错误判定成合数的情况。这个问题很有意思，难道是素数比较珍贵，宁肯错判，也不愿意漏掉吗。

抱着这个想法，我去搜索了相关的资料发现在费马小定理中指出，如果 n 是一个素数，那么对于任意整数 a（1 < a < n），a^(n-1) mod n 等于 1。Miller-Rabin 测试对此定理进行了一种特定形式的检验：它通过随机选择多个 a 值，并验证 a^(n-1)是否等于 1（mod n）来检测 n 是否为素数。如果对于所有选定的 a，该等式都成立，那么 n 被认为“可能是素数”。这里的关键在于，**对于一个实际上是合数的 n，可能仍然存在某些 a 使得 a^(n-1) mod n 等于 1，即使 n 不是素数**。换句话说，即使一个合数 n 可能偶然通过了这一检验，也不意味着所有的 a 都能使 n 通过测试。因此，当一个数通过了 Miller-Rabin 测试，我们只能说它“可能是素数”，而不能确定它一定是素数。

然而，反过来就不成立了。如果 n 是素数，那么对于所有 1＜a＜n，a^(n-1) mod n 总是等于 1。这意味着如果一个数是素数，它必定会通过 Miller-Rabin

测试。因此，Miller-Rabin 测试不会将一个真正的素数错误地判定为合数。

三，Miller-Rabin 素性测试和 AKS 素数检测算法时间复杂度分析

对于 Miller-Rabin 素性测试而言，时间复杂度主要取决于两个因素：**进行测试的轮数和被测试数的大小**。对于一个给定的整数 n，每一轮测试的时间复杂度大致为 O(k log^3 n)。所以想要获得较为精确的结果就必须牺牲时间来增加测试的轮数。相比之下，AKS 算法的时间复杂度是多项式的，具体为 O((log n)^{12})。这意味着算法的运行时间与输入数字的大小成多项式关系，随着输入数字的大小增加，所需的时间急剧增加。这使得 AKS 算法在处理大型数字时效率较低（但后面在查阅资料的时候发现它居然是第一个在多项式时间内确定性地解决素数检验问题的算法）。

六、教师评语	成绩
签名： 日期：	