Review: Principles and Applications of Pseudorandom Number Generators

一、引入 Introduction

随机数在日常生活和科学研究中扮演着重要角色。它们在密码学、游戏开发和数据分析等多个领域至关重要。尽管我们经常需要随机数，但我们通常生成的是伪随机数。这些数字乍看起来似乎是随机的，但实际上它们是由算法确定的，因此具有一定的模式，理论上是可以被破解的。

在计算机科学中，使用伪随机数生成器（PRNG）来生成随机数序列是执行各种任务的重要步骤。然而，由于现代计算机主要基于冯·诺依曼架构设计，这些计算机无法生成真正的随机数。这是因为它们只能根据预定义的程序指令有序地执行任务。

本文的主要内容是介绍各种伪随机和真随机数生成器的原理，讨论多种编程语言中随机数API的实现方法，并设计相关实验来验证NIST测试。

二、基本概念 Basic Concept

根据密码学的原理，随机数的随机性可以通过三个标准来测试：

1. 统计伪随机性。这指的是在给定的随机比特流样本中，1的数量大致等于0的数量。同样，'10'、'01'、'00'和'11'的数量也大致相等。这种标准也被称为统计随机性。满足这些要求的数字乍看起来对人眼来说似乎是随机的。
2. 密码学伪随机性。它被定义为在给定随机样本的一部分和随机算法的情况下，无法有效地推断出随机样本的剩余部分。
3. 真随机性。它被定义为随机样本的不可重现性。实际上，只要给定边界条件，真正的随机数就不存在。然而，如果生成真随机数样本的边界条件非常复杂且难以捕捉（例如局部背景辐射波动值），则可以认为使用这种方法生成的是真随机数。但实际上，这只是非常接近真随机数的伪随机数。一般来说，基于经典力学生成的任何随机数都只是伪随机数。

相应地，随机数被分为三类：

1. **弱伪随机数**：满足第一个条件的随机数。
2. **密码学安全伪随机数**：满足前两个条件的随机数。它们可以通过密码学安全伪随机数生成器计算得出。
3. **真随机数**：满足所有三个条件的随机数。

随机数在密码学中非常重要。广泛用于保密通信的会话密钥的生成需要真随机数的参与。如果随机数生成算法存在缺陷，会话密钥可以直接被推断出来。如果发生这样的事故，任何加密算法都将失去意义。

因此，评估各种生成随机序列的方法是必要的，以确保它们满足相应的安全标准。这包括对伪随机数生成器（PRNGs）和真正随机数生成器（TRNGs）的安全性、随机性和统计属性进行严格的测试和验证。除了需要满足Golomb提出的伪随机性测试条件外，伪随机序列还需要满足周期足够大、产生在计算上容易以及不可由部分信息推断出整体等要求。只有那些通过严格测试并在输出中表现出足够随机性和不可预测性的随机数生成器才能被信任并用于密码学应用。

三、伪随机数生成器 PRNG

1. 同余法

1.1 线性同余发生器（Linear Congruential Generator, LCG）

线性同余发生器是最简单的生成伪随机数的方法。递推公式为:

$$X_{i+1} = (aX_i + c) \mod M$$
$$r_i = \frac{X_i}{M}$$

其中：a 是乘子；c 是增量；M 是模数；X_0 是种子变量

得到的 r_i 为均匀分布的 01 之间的随机数。

当 $c = 0$ 时，这种方法也被称为乘同余法。线性同余法的最大周期是 M，但通常会小于 M。

参数选取要求：为了使周期达到最大，参数 a、c 和 M 应满足以下条件：

1. c 和 M 互质（没有共同的质因子，确保在一次完整的循环中，所有取值都能够被覆盖到）
2. M 的所有质因子的积能整除 $a-1$（确保在一次完整的循环中，所有取值都能够被生成，以避免出现周期缩短的情况）
3. 如果 M 是 4 的倍数，则 $a-1$ 也是 4 的倍数

在查询相关文献时，我了解到了线性同余序列在随机数生成中的一些问题，特别是其**高维不均匀性。**当将线性同余序列中相继的随机数视为多维空间上的一个点的坐标时，这些点会散布在少数几个超平面上，形成稀疏网格结构。这种稀疏性在二维情况下已经可以通过图形直观地观察到。均匀随机数是产生其他分布随机变数的基础，然而，如果利用具有稀疏网格结构的线性同余序列来产生其他分布的随机变数，可能会出现一些问题。例如，当尝试通过变换两个独立的均匀随机数来得到一对独立的正态分布随机变数时，如果这两个均匀随机数来自同一个线性同余序列，那么它们实际上可能并不独立，而是落在一条螺线上。

> 比如工程概率统计上的一道题：
>
> (15 points) Continued with Example 4.5 (3), prove that if U_1~U[0,1] and U_2~U[0,1] are independent, and let
>
> $$\begin{cases} Z_1 = \sqrt{-2\ln(U_1)}\cos(2\pi U_2) \\ Z_2 = \sqrt{-2\ln(U_1)}\sin(2\pi U_2) \end{cases},$$
>
> then Z_1 and Z_2 are a pair of independent standard normal random variables.
> (Hint: Show that $P(Z_1 \le a, Z_2 \le b) = \Phi(a)\Phi(b)$ for all a and b, which requires variable substitution in a double integral.)
>
> Set $X = Z_1^2 + Z_2^2 = -2\ln(U_1), Y = Z_2/Z_1 = \tan(2\pi U_2)$.
>
> Then by inverse function, $U_1 = e^{\frac{-(Z_1^2+Z_2^2)}{2}}, U_2 = \frac{\arctan\frac{Z_2}{Z_1}}{2\pi}$. And the Jacobian yields as
>
> $$\begin{aligned} \frac{\partial(U_1,U_2)}{\partial(Z_1,Z_2)} &= \begin{vmatrix} \frac{\partial U_1}{\partial Z_1} & \frac{\partial U_1}{\partial Z_2} \\ \frac{\partial U_2}{\partial Z_1} & \frac{\partial U_2}{\partial Z_2} \end{vmatrix} \\ &= \left[\frac{1}{\sqrt{2\pi}}e^{-\frac{Z_1^2}{2}}\right]\left[\frac{1}{\sqrt{2\pi}}e^{-\frac{Z_2^2}{2}}\right] \\ &= \phi(Z_1)\phi(Z_2) \end{aligned}$$

$$
\begin{aligned}
f_{Z_1,Z_2}(z_1,z_2) &= f_{U_1,U_2}(u_1(z_1,z_2),u_2(z_1,z_2))\left|\frac{\partial(U_1,U_2)}{\partial(Z_1,Z_2)}\right| \\
&= 1\cdot\left|\frac{\partial(U_1,U_2)}{\partial(Z_1,Z_2)}\right| \\
&= \phi(z_1)\phi(z_2)
\end{aligned}
$$

So Z_1, Z_2 are independent standard normal random random variables.

可以发现 Z_1, Z_2 是独立服从正态分布，U_1, U_2 服从独立均匀分布，但是通过 U_1, U_2 共同表达得到的式子却构成了螺线的方程

自从发现高维稀疏网格现象之后，人们在构造线性同余发生器时开始考虑参数的选择，以使网格尽可能不那么稀疏。这导致了一系列度量和检验网格稀疏程度的准则的出现，如相邻平行超平面之间最大距离、平行超平面的最小数目、点间距等检验方法。这些检验都属于理论检验，可以在给定参数的情况下进行，而无需实际产生具体的序列。其中，基于相邻平行超平面之间最大距离的检验，也称为**谱检验**，是一种重要的检验方法。谱检验的计算归结为求解一个极值问题，该问题的解称为发生器的维精度，而相邻平行超平面之间的最大距离则是评估发生器性能的一个重要指标。

综上所述，线性同余序列在随机数生成中的应用需要谨慎考虑其高维不均匀性，并通过合理的参数选择和检验方法来确保生成的随机数的质量和独立性。

优点:

- 实现简单，计算速度快。
- 参数选择合适时，可以生成具有较长周期的伪随机数序列。

缺点:

- 安全性较弱，容易被人破解

1.2 非线性同余发生器

非线性同余发生器的递推公式为:
$X_{n+1} = f(X_n) \mod m$

其中 $f(X_n)$ 是非线性函数。常见的非线性同余发生器包括:

- 二次同余: $X_{n+1} = (X_n^2 + c) \mod m$
- 立方同余: $X_{n+1} = (X_n^3 + c) \mod m$
- 逆同余发生器是非线性同余类中被研究得最多的一种，其中一种推导公式:

$$
\begin{aligned}
X_{i+1} &= (a\overline{X_i} + c) \mod M \\
r_i &= \frac{X_i}{M}
\end{aligned}
$$

其中 $\overline{X_i}$ 表示 X_i 关于素数模数 M 的乘法逆元。一般使用快速幂和费马小定理、扩展欧几里得算法求解乘法逆。

- 混合同余发生器，递推公式为

$$
\begin{aligned}
Y_{i+1} &= (aX_i + c) \mod M \\
X_{i+1} &= f(X_{i+1}) \\
r_i &= \frac{X_i}{M}
\end{aligned}
$$

优点:

- 由于非线性特性，生成的随机数序列具有更好的随机性，能克服高位网络结构。
- 可以生成更复杂的随机数序列，减少模式的出现。

缺点：

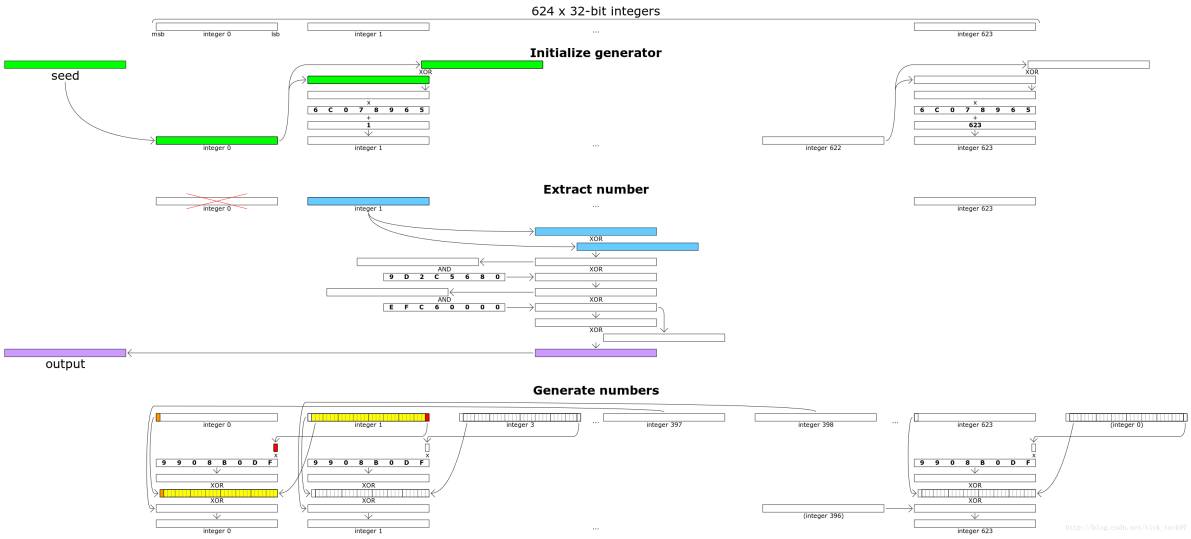
- 逆同余发生器的速度明显慢于线性同余，但是在现在的硬件条件上这点相对不明显了。
- 逆同余发生器的周期不比线性同余发生器的长。

2. Mersenne Twister（MT）

Mersenne Twister是一种高效的伪随机数生成算法，由日本数学家松本真和西村拓士在1997年共同开发。该算法的名称来源于其周期长度通常取Mersenne质数（即形如2^p-1的质数，其中p为整数）这一事实。Mersenne Twister算法基于有限二进制字段上的矩阵线性递归，能够快速产生高质量的伪随机数，修正了古典随机数生成算法的很多缺陷。

整个算法分成了三个阶段：

- 第一阶段：获得基础的梅森旋转链；
- 第二阶段：对于旋转链进行旋转算法；
- 第三阶段：对于旋转算法所得的结果进行处理



对应的伪代码如下：

```
// Create an array of length 624 to store the generator's state
int[0..623] MT
int index = 0

// Initialize the generator, using the seed as the first element
function initialize_generator(int seed) {
    i := 0
    MT[0] := seed
    for i from 1 to 623 { // Traverse the remaining elements
        MT[i] := last 32 bits of(1812433253 * (MT[i-1] xor (right shift by 30
bits(MT[i-1]))) + i) // 1812433253 == 0x6c078965
    }
}

// Extract a tempered pseudorandom number based on the index-th value,
// calling generate_numbers() every 624 numbers
function extract_number() {
```

```
    if index == 0 {
        generate_numbers()
    }

    int y := MT[index]
    y := y xor (right shift by 11 bits(y))
    y := y xor (left shift by 7 bits(y) and (2636928640)) // 2636928640 ==
0x9d2c5680
    y := y xor (left shift by 15 bits(y) and (4022730752)) // 4022730752 ==
0xefc60000
    y := y xor (right shift by 18 bits(y))

    index := (index + 1) mod 624
    return y
}

// Generate an array of 624 untempered numbers
function generate_numbers() {
    for i from 0 to 623 {
        int y := (MT[i] & 0x80000000)                       // bit 31 (32nd bit)
of MT[i]
                       + (MT[(i+1) mod 624] & 0x7fffffff)   // bits 0-30 (first
31 bits) of MT[...]
        MT[i] := MT[(i + 397) mod 624] xor (right shift by 1 bit(y))
        if (y mod 2) != 0 { // y is odd
            MT[i] := MT[i] xor (2567483615) // 2567483615 == 0x9908b0df
        }
    }
}
```

优点:

- 周期极长，适合需要大量随机数的应用。
- 生成的随机数序列具有良好的统计特性，在高维空间中仍然均匀分布
- 内部状态转移函数具有高的线性复杂度，这使得序列难以被预测。

缺点:

- 初始化过程相对复杂，需要较多的计算。
- 由于其内部状态较大，占用内存较多，需要 2.5 KiB的缓存空间

3. Linear Feedback Shift Register（LFSR)

线性反馈移位寄存器是一种特殊的移位寄存器，其特点是在每次时钟脉冲到来时，不仅将寄存器中的各位向右（或向左）移动一位，而且还将某些位的线性组合通过异或门反馈到输入端。这样，寄存器中的状态就会按照一定的规律变化，从而生成一个伪随机序列。

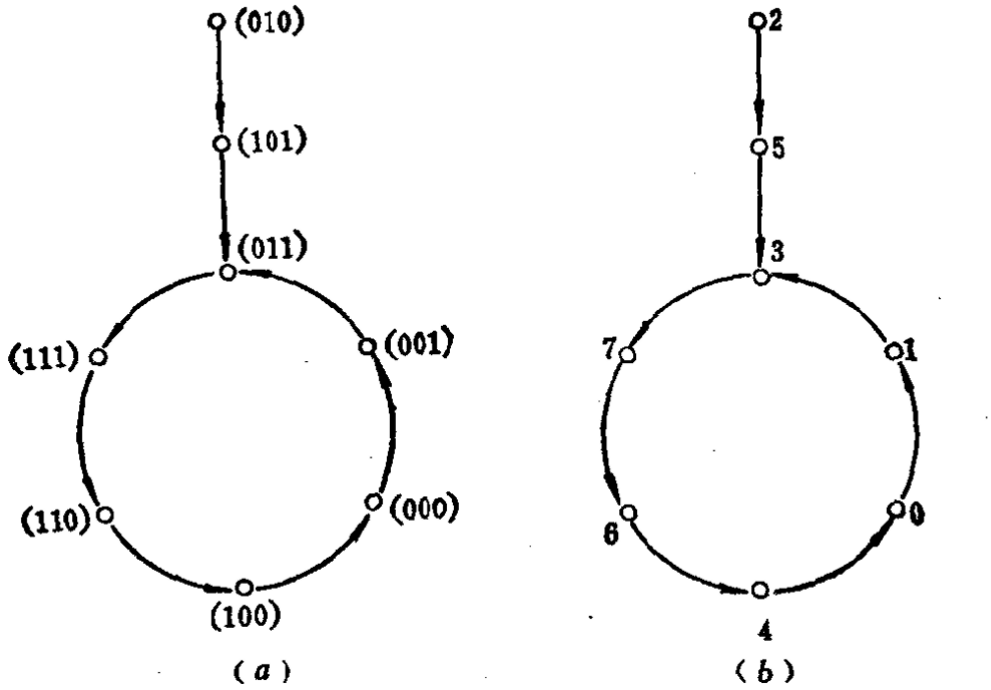
> 这一部分内容和本学期的专业课《数字逻辑》部分内容相互映照。在这个伪随机序列生成器中，移位器中的所有数字共同组成了一个状态，这个状态决定了下一个状态和输出。因此，我们可以使用有向图来表示状态的转移和切换。在这个有向图中，每个节点代表一个状态，每条有向边代表一个状态转移。由于伪随机序列生成器是确定性的，所以每个节点（状态）都有且仅有一条出边（状态转移）。

为了使产生的随机序列周期更长，我们需要设计一个尽可能复杂的状态图。这可以通过增加移位寄存器的位数、选择合适的反馈函数以及调整初始状态来实现。增加移位寄存器的位数可以直接增加状态空间的大小，从而增加序列的周期。选择合适的反馈函数可以使得状态图中的环路更加复杂，进一步增加序列的随机性和不可预测性。调整初始状态则可以改变序列的起始点，使得相同的硬件在不同的初始条件下生成不同的伪随机序列。

线性反馈移位寄存器的递推公式为：

$$X_{n+1} = (X_n << 1) \oplus g(X_n)$$

其中 g 是反馈函数，X_n 表示当前状态，其中包含移位器中 k 个位数。



优点：

- LFSR基于移位和异或操作，这些操作在数字逻辑电路中相对简单且易于实现
- LFSR生成的伪随机序列可以使数据频谱展宽，有助于减小电磁干扰（EMI）噪声，这在通信系统中尤为重要。

缺点：

- 由于LFSR是线性的，其输出序列可能在某些情况下表现出可预测性
- LFSR的状态大小（即寄存器的位数）限制了其周期和随机性

4. 减法进位随机数生成器（Subtract with Carry, SWC）

减法进位随机数生成器是一种高效的伪随机数生成算法，其基本思想是使用两个序列的差值来生成新的随机数，并在需要时进行进位调整。这种算法由George Marsaglia和Arif Zaman在1991年提出。

SWC 的算法过程是：

1. 初始化状态向量，通常是一个长度为 r 的数组，每个元素是 w 位的整数，一般可以使用 `std::random_device{}()` 进行初始化；进位值 c，初始值通常为 0，用来增加算法的复杂度；模数 m 通常是 2^w，即 w 为整数的最大值。
2. 选择一个参数为 步长 s，通常和 r 互素（保证状态向量中每个元素都能均匀地相互影响）
3. 产生随机数列

1. 计算差值 $y = S[i] - S[i-s] - c$，如果结果为负，则 $c = 1, y$ 加上模数 m 保证结果为正；否则 $c = 0$.
2. $S[i+1] = y$，且 y 作为新的随机数输出。

优点：

- 仅涉及减法、模运算，计算速度快
- 周期较长，可以达到 2^{rw}
- 引入进位参数 c 增加了算法的复杂度和状态空间，使得生成的随机序列具有较高的熵

缺点：

- 初始状态敏感，即使使用 `std::random_device{}()` 进行初始化也不一定能得到性质良好的随机序列
- 仍然是线性结构，可能会被破解，特别是状态向量较短的时候。

5. 时滞斐波那契生成器(Lagged Fibonacci generator, LFG)

斐波那契数列和 Tausworthe 序列的递推公式为：

$$X_{n+1} = (X_n * X_{n-k}) \mod m$$

新项由两个老项计算生成。m 通常是2的幂 $(m = 2^M)$, 经常 2^{32} 或 2^{64}。其中 `⋆算符` 表示一般的二元运算符，这可以是加法、减法、乘法或者位运算异或。使用k个状态字的生成器，称作'记住'了过去 k 个值。

参数选取要求：

若是加法（减法）生成器，则前k个初始值中至少包含一个奇数（保证了生成的随机数中包含奇数和偶数）

若是乘法生成器，则前k个初始值要求全部是奇数（如果存在偶数，则在很短的周期内就会产生模数 2^M，之后的序列为全零序列）

优点：

- 生成的随机数序列具有较好的随机性。
- 实现相对简单，计算速度较快。

缺点：

- 周期相对较短，不适合需要大量随机数的应用。
- 对参数选择敏感，不当的参数可能导致周期缩短或随机性下降。
- 对于经典斐波那契生成器($X_{n+1} = (X_n + X_{n-1}) \mod m$)，发现一个**特殊的性质（缺陷）**：不会出现 $X_{n-1} < X_{n+1} < X_n, X_n < X_{n+1} < X_{n-1}$，即第三个数不是大于前两个数就是同时小于前两个数。

6. 其他伪随机数生成器

6.1 均匀长周期线性并 WELL family of generators

WELL（Well-Equidistributed Long-period Linear）生成器是一种改进的线性反馈移位寄存器生成器，具有更好的统计特性和更长的周期。

操作设计类似于 Mersenne Twister，读取状态中最旧的32位，最新的32位，以及中间三个其他字（每个字由32位组成，不足可能会跨越边界），然后通过一系列单字变换（主要包括移位和异或操作）得到两个词作为状态中最新的两个词，其中一个将作为输出

优点：

- 周期极长，适合需要大量随机数的应用
- 生成的随机数序列具有良好的统计特性
- 实现相对简单，计算速度较快

缺点：

- 需要复杂的计算，和大量的内存来存储状态

6.2 Blum-Blum-Shub生成器（BBSG）

Blum Blum Shub (BBS) 算法是一种密码学安全的伪随机数生成器，由 Blum, Blum 和 Shub 三位密码学家在1986年提出。BBS算法的安全性基于大数分解的困难性，与RSA加密算法的安全性基础相似。如果希望具有与RSA相同级别的安全性，选择的质数大小必须与RSA密钥的大小相近。

BBS 算法过程：

1. 选择两个大素数 p，q满足 $p \equiv q \equiv 3(\mod 4)$，令 $n = pq$
2. 选择一个随机数 s 要求 s 与 n 互素，然后计算 $X_0 = s^2 \mod n$
3. 随机序列为 $X_{i+1} = (X_i^2 - 1) \mod n$，$r_i = X_i \mod n$

优点：

- 生成的随机数序列具有较高的安全性，适合加密应用。
- BBS算法的效率相对较高。
- BBS算法能够产生具有较长周期的随机序列

缺点：

- 对素数的选择非常敏感，不当的素数可能导致安全漏洞。

6.3 Micali-Schnorr生成器（MSG）

1. 选择两个大素数 p 和 q，计算 n=pq
2. 选择一个与 ϕ(n)$\phi(n)$ 互素的整数 ee 作为公钥。
3. 选择一个整数 x 作为种子，要求 x 与 n 互素。
4. **生成伪随机数**：
   - 计算 $z_i = x^e \mod n$。
   - 从 z_i 中提取 k 位作为输出，通常取 z_i 的最低 k 位。
   - 更新种子 x 为 z_i。

安全性：MSG的安全性基于 RSA 问题的困难性，即在不知道大素数 p 和 q 的值的情况下，很难从生成的伪随机数序列中恢复出种子 x

优点：

- 生成的随机数序列具有较高的安全性，适合加密应用。

缺点：

- 计算速度相对较慢，计算复杂度较高，不适合需要大量随机数的应用。
- 对基和素数的选择非常敏感，不当的选择可能导致安全漏洞。

四、不同编程语言提供的API、方法

4.1 C/C++

引擎	描述
`std::minstd_rand`	线性同余引擎，产生均匀的伪随机数序列。
`std::mt19937`	梅森旋转算法，适合通用随机数生成。
`std::mt19937_64`	64 位的梅森旋转算法。
`std::ranlux24_base`	简化的减法进位引擎，用于高质量生成。
`std::knuth_b`	Knuth shuffle 随机数生成器。

我们可以在 `random.h` 文件中找到对应的定义、参数和实现方法

1. **minstd_rand0 和 minstd_rand**

都使用了线性同余随机数生成器（LCG），但是参数不一样，形式是：$X_{i+1} = (aX_i + c) \mod M$；其中参数 $c = 0, M = 2147483647, a_1 = 16807, a_2 = 48271$。`minstd_rand0` 是基于 Lewis、Goodman 和 Miller 提出的经典最小标准随机数生成器；`minstd_rand` 是另一种线性同余随机数生成器，通常被称为 Lehmer 生成器。

```
/**
   * The classic Minimum Standard rand0 of Lewis, Goodman, and Miller.
   */
  typedef linear_congruential_engine<uint_fast32_t, 16807UL, 0UL, 2147483647UL>
  minstd_rand0;

  /**
   * An alternative LCR (Lehmer Generator function).
   */
  typedef linear_congruential_engine<uint_fast32_t, 48271UL, 0UL, 2147483647UL>
  minstd_rand;
```

2. **mt19937 和 mt19937_64**

这是两个经典的梅森旋转算法（Mersenne Twister）伪随机数生成器，都使用了 mersenne_twister_engine 模板类，区别在于随机数的位数、状态向量的长度、提取随机数的掩码等相关参数不同。

以 mt19937 为例，生成的是32位的随机数，相关参数含义是：

```
  /**
   * The classic Mersenne Twister.
   *
   * Reference:
   * M. Matsumoto and T. Nishimura, Mersenne Twister: A 623-Dimensionally
   * Equidistributed Uniform Pseudo-Random Number Generator, ACM Transactions
```

```
   * on Modeling and Computer Simulation, Vol. 8, No. 1, January 1998, pp 3-30.
   */

  typedef mersenne_twister_engine<
    uint_fast32_t,  // 表示生成器使用的整数类型
    32,             // 每个随机数的位数
    624,            // 状态向量的长度
    397,            // 状态向量的偏移量
    31,             // 用于提取随机数的掩码
    0x9908b0dfUL,   // 用于混合的常数
    11,             // 右移操作的位数
    0xffffffffUL,   // 用于掩码的常数
    7,              // 左移操作的位数
    0x9d2c5680UL,   // 用于混合的常数
    15,             // 右移操作的位数
    0xefc60000UL,   // 用于混合的常数
    18,             // 右移操作的位数
    1812433253UL    // 用于初始化的常数
> mt19937;

  /**
   * An alternative Mersenne Twister.
   */
  typedef mersenne_twister_engine<
    uint_fast64_t,
    64, 312, 156, 31,
    0xb5026f5aa96619e9ULL, 29,
    0x5555555555555555ULL, 17,
    0x71d67fffeda60000ULL, 37,
    0xfff7eee000000000ULL, 43,
    6364136223846793005ULL> mt19937_64;
```

3. **ranlux24, ranlux48**

ranlux24_base 和 ranlux48_base是基于减法与进位（subtract with carry）算法的随机数生成器，参数分别为 $(24, 10, 24), (48, 5, 12)$.

ranlux24 和 ranlux48 是基于 ranlux24_base 和 ranlux48_base 的**丢弃块引擎（discard block engine）**。

丢弃块引擎在一个简单的随机数生成器的基础上，需要两个参数 $(p, q), p > q$，会随机丢弃前 $p - q$ 个随机数，保留最后 q 个，最终选择一个最为输出，使得生成的随机序列更加难以预测。

```
  typedef subtract_with_carry_engine<uint_fast32_t, 24, 10, 24>
    ranlux24_base;

  typedef subtract_with_carry_engine<uint_fast64_t, 48, 5, 12>
    ranlux48_base;

  typedef discard_block_engine<ranlux24_base, 223, 23> ranlux24;

  typedef discard_block_engine<ranlux48_base, 389, 11> ranlux48;
```

4. **random_device**

这是一个标准接口，用于访问平台特定的非确定性随机数生成器，提供了一种从硬件或操作系统获取真正随机数的方法。一般可以作为种子/初始值使用。

4.2 Java

引擎	描述
`java.util.Random`	基于线性同余法（LCG）的伪随机数生成器，产生均匀的伪随机数序列。
`java.security.SecureRandom`	加密安全的伪随机数生成器，适合需要高随机性保证的场景，如加密。
`java.util.concurrent.ThreadLocalRandom`	为多线程环境优化的随机数生成器，每个线程有独立的随机数生成器，避免锁竞争。
`Math.random()`	生成0.0到1.0之间的随机双精度浮点数，实际上是 `java.util.Random` 的一个封装。

其中**最特殊的就是** `SecureRandom` ，是 `Java` 中用于生成安全随机数的类。 `SecureRandom` 提供了与某个特定算法集合相关的包， 该包可以独立实现。

当我们使用 `SecureRandom.getInstance()` 请求一个 `SecureRandom` 实例时， 可以申请实现某个特定的算法。 如果算法可行， 那么就可以将它作为 `SecureRandom` 的对象使用。 如果算法不可行， 或者我们没有为算法明确特定的实现方法， 那么会由系统选择 `SecureRandom` 的实现方法。

以下为内部实现，默认情况下， `SecureRandom` 会尝试使用系统配置的默认 PRNG 算法，如果失败，则回退到 `SHA1PRNG` 算法。

```
    synchronized public void nextBytes(byte[] bytes) {
        secureRandomSpi.engineNextBytes(bytes);
    }
    public SecureRandom() {
        super(0);
        getDefaultPRNG(false, null);
    }
    private void getDefaultPRNG(boolean setSeed, byte[] seed) {
        String prng = getPrngAlgorithm();
        if (prng == null) {
            // bummer, get the SUN implementation
            prng = "SHA1PRNG";
            this.secureRandomSpi = new sun.security.provider.SecureRandom();
            this.provider = Providers.getSunProvider();
            if (setSeed) {
                this.secureRandomSpi.engineSetSeed(seed);
            }
        } else {
            try {
                SecureRandom random = SecureRandom.getInstance(prng);
                this.secureRandomSpi = random.getSecureRandomSpi();
                this.provider = random.getProvider();
                if (setSeed) {
                    this.secureRandomSpi.engineSetSeed(seed);
                }
```

```
            } catch (NoSuchAlgorithmException nsae) {
                // never happens, because we made sure the algorithm exists
                throw new RuntimeException(nsae);
            }
        }
        if (getClass() == SecureRandom.class) {
            this.algorithm = prng;
        }
    }
```

在 `sun.security.provider.SeedGenerator` 类里，可以看到 seed 是利用 `/dev/random` 或 `/dev/urandom` 来生成的，启动应用程序时可以通过参数 `file:/dev/urandom` 来指定 seed 源。从操作系统的本地随机数生成器中可以得到高质量的熵，因此能提供较高的安全性。

```
    static {
        String var0 = SunEntries.getSeedSource();
        if (!var0.equals("file:/dev/random") &&
!var0.equals("file:/dev/urandom")) {
            if (var0.length() != 0) {
                try {
                    instance = new SeedGenerator.URLSeedGenerator(var0);
                    if (debug != null) {
                        debug.println("Using URL seed generator reading from " +
var0);
                    }
                } catch (IOException var2) {
                    if (debug != null) {
                        debug.println("Failed to create seed generator with " +
var0 + ": " + var2.toString());
                    }
                }
            }
        } else {
            try {
                instance = new NativeSeedGenerator(var0);
                if (debug != null) {
                    debug.println("Using operating system seed generator" +
var0);
                }
            } catch (IOException var3) {
                if (debug != null) {
                    debug.println("Failed to use operating system seed generator:
" + var3.toString());
                }
            }
        }

        if (instance == null) {
            if (debug != null) {
                debug.println("Using default threaded seed generator");
            }

            instance = new SeedGenerator.ThreadedSeedGenerator();
        }
```

```
    }
```

在使用的过程中，也会发现 `java.security.SecureRandom` 产生随机数的速度明显慢于其他随机数生成器方法。

4.3 Python

引擎	描述
`random` 模块	Python 的标准库之一，提供了生成伪随机数的功能。基于 Mersenne Twister 算法，生成均匀分布的随机数。适合一般应用。
`secrets` 模块	用于生成密码安全的随机数。基于操作系统提供的熵源（如 `/dev/urandom`），生成不可预测的随机数。适合密码学应用。
`numpy.random` 模块	`numpy` 库中的随机数生成器，提供了更高效和更丰富的随机数生成功能。基于 Mersenne Twister 算法，适合生成大量随机数。
`numpy.random.Generator`	在 Numpy 1.17 版本中引入的全新随机数生成器，基于 BitGenerator 实例，提供了更好的可重复性和统计特性。

除了Python标准库中的 `secrets` 模块和 `os` 库中的 `urandom` 函数外，还有一些第三方库可以用于生成密码安全的随机数，如 `cryptography` 库和 `pycryptodome` 库等。

相关实现和 `java`，`c/c++` 中的类似。

4.4 Matlab

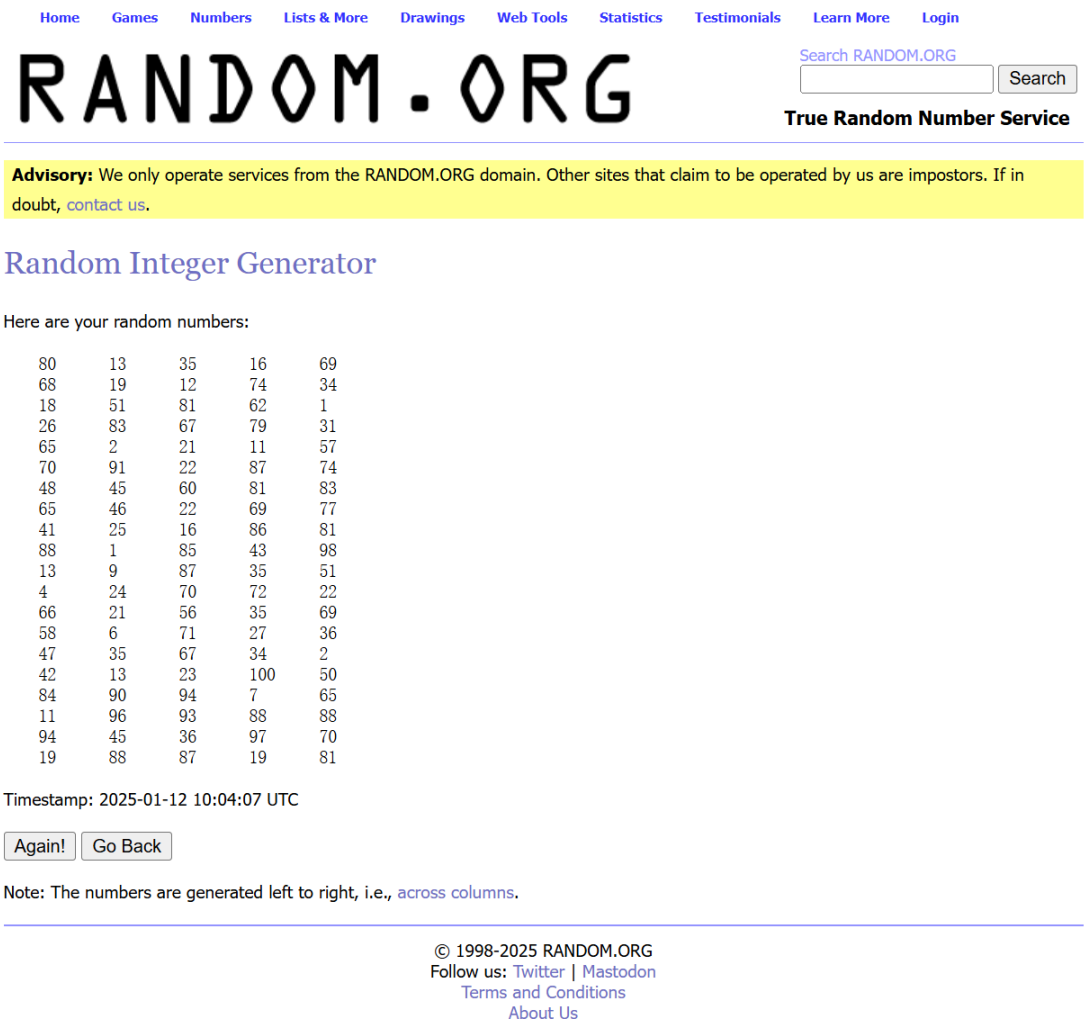
引擎	描述
`rand` 函数	生成 (0, 1) 区间上均匀分布的伪随机数。基于 Mersenne Twister 算法。
`randi` 函数	生成 (0, N] 区间上均匀分布的伪随机整数。基于 Mersenne Twister 算法。
`randn` 函数	生成标准正态分布（均值为 0，方差为 1）的随机数。基于 Box-Muller 变换。
`rng` 函数	控制随机数生成器的种子和算法。支持多种算法，如 `twister`（梅森旋转）、`simdTwister`（面向 SIMD 的快速梅森旋转算法）、`combRecursive`（组合多递归）、`philox`（执行 10 轮的 Philox 4×32 生成器）、`threefry`（执行 20 轮的 Threefry 4×64 生成器）等。

五、真随机数生成器

在现代技术应用中，随机数生成器（RNG）扮演着至关重要的角色，尤其是在加密、模拟、统计分析和游戏开发等领域。随机数生成器主要分为两类：伪随机数生成器（PRNG）和真随机数生成器（TRNG）。伪随机数生成器通过确定性算法生成序列，而真随机数生成器则依赖于物理过程中的不可预测性。本文将介绍几种常见的真随机数生成器及其特点和适用场景。

5.1 Random.org：

Random.org 是一个广泛使用的真正随机数生成器的网站。该网站生成的随机序列是免费分发的，吸引了各种用户。熵是从大气噪声中收集的，该网站的随机数生成器中的无线电设备接收噪声并通过后处理器，然后转换成二进制的 1 和 0 的流。几个第三方已经证明，这个网站上的数字序列通过了行业标准测试套件，使其成为不定期或永久用户获取随机数的免费且可行的选择。



5.2 HotBits：

- **描述**：HotBits 是另一个基于互联网的流行随机数生成器。它基于放射性衰变生成随机数序列，从这个生成器获得的随机数是通过互联网发送的，所以总是有可能第三方知道这个序列。这使得它不适合安全用途，但当需要不容置疑的随机数据时，HotBits 是有用的。
- HotBits 随机数生成器使用说明：New User Guide on the Web – HOTBIT Support Center

六、测试比较

6.1 随机数生成器的评价指标

具体包括了：

1. 精确性：所需要产生的随机数的分布函数是确定的， 那么生成的方法也是确定的。
2. 数学上的有效性：是否满足某些条件。
3. 速度：生成随机数的速度必须是可以接受的。
4. 空间：生成随机数所需的计算机内存。

5. 简单：算法和实现都必须满足此条件。
6. 参数稳定性：对所有的参数而言都是速度均匀的。

在这里，我们主要考虑随机数列各种特征上是否满足，而暂时忽略速度、空间等性能、效率差异。在评估伪随机数生成器和真随机数生成器时，NIST SP 800-22测试标准和PKCS#11标准提供了重要的指导。NIST SP 800-22测试标准由美国国家标准技术研究院（NIST）制定，旨在为 RNG 和 PRNG 提供一套详尽的测试方法，用于检验其生成的随机数是否满足密码学应用的要求。PKCS#11标准则定义了安全硬件（如硬件安全模块HSM ）的接口，其中也涉及到随机数生成器的规范。

6.2 NIST 测试

在NIST SP 800-22中，有以下15个测试方法：

1. **频率（单比特）测试**：检查序列中0和1的个数是否接近，以判断序列是否均匀分布。
2. **频率测试（块内）**：将序列分成多个块，检查每个块中0和1的个数是否接近，以评估序列在局部的均匀性。
3. **累积和（Cusum）测试**：通过计算序列的累积和，检测序列是否存在趋势或周期性变化。
4. **运行测试（基于块）**：检查序列中0和1的连续出现次数（即运行长度）是否符合随机序列的统计特性。
5. **运行测试（重叠）**：与运行测试类似，但采用重叠的方式计算运行次数，以更细致地分析序列的随机性。
6. **二进制矩阵秩测试**：将序列转换为二进制矩阵，计算矩阵的秩，以检测序列中是否存在线性相关性。
7. **离散傅里叶变换测试**：利用离散傅里叶变换分析序列的频谱特性，检测序列是否存在周期性成分。
8. **非重叠模板匹配测试**：检查序列中特定模板（如特定的01序列）的出现次数是否符合预期，以评估序列的随机性。
9. **重叠模板匹配测试**：与非重叠模板匹配测试类似，但采用重叠的方式计算模板的出现次数。
10. **比特序列通用统计测试**：通过计算序列的压缩率，检测序列是否具有可压缩性，从而评估其随机性。
11. **Lempel-Ziv复杂度测试**：利用Lempel-Ziv算法计算序列的复杂度，以判断序列是否具有随机性。
12. **线性复杂度测试**：计算序列的线性复杂度，检测序列是否存在线性规律。
13. **序列测试**：通过分析序列中相邻位之间的关系，检测序列是否存在某种模式或规律。
14. **近似熵测试**：计算序列的近似熵，以评估序列的随机性和不可预测性。
15. **随机游程测试（及其变体）**：将序列视为随机游程过程，分析游程的长度和方向等特性，以检测序列的随机性。

为了进行随机数生成器的测试，我下载了 NIST SP 800-22 测试套件。该套件可以从 NIST 官方网站 获取。下载后，我使用 Windows Subsystem for Linux (WSL, version: Ubuntu 22.04.5 LTS) 在本地环境中安装和配置了测试套件。

我分别使用了 `C/C++`、`Java` 中的 `minstd_rand`,`mt19937`,`ranlux24_base`,`java.security.SecureRandom` 等六个随机数生成器产生的 10 MB的 01随机序列，使用NIST测试套件进行15个指标的测试。

随机数列生成器代码在 src 文件夹中，产生的 01随机数列在 data 文件夹中，测试结果在 result 文件夹中。

以近似熵测试(ApproximateEntropy)为例，测试结果如下：

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1		c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	P-value	partition	STATISTICAL TEST	
2	java.util.Random	0	0	0	0	0	1	0	3	1	0	----	5/5	Frequen	
50	java.util.Random	0	0	2	0	1	0	0	0	1	1	----	5/5	ApproximateEntropy	
58	java.security.SecureRandom	1	0	1	1	0	0	0	1	1	0	----	5/5	ApproximateEntropy	
58	Math_random()	0	0	0	1	0	1	0	2	1	0	----	5/5	ApproximateEntropy	
58	minstd_rand()	1	0	0	1	2	0	0	0	1	0	----	5/5	ApproximateEntropy	
58	mt20095	2	1	0	1	0	1	0	0	0	0	----	5/5	ApproximateEntropy	
58	ranlux182_base	0	0	0	0	0	2	0	1	2	0	----	5/5	ApproximateEntropy	
88															

> **C1 - C10**：这些列代表了不同的P值区间，每个区间覆盖0.1的范围，从0到1。
>
> **P-VALUE**：这是每个测试的P值，表示序列通过测试的概率。P值大于0.01表示序列通过了测试。
>
> **PROPORTION**：这是通过测试的序列比例。例如，5/5表示所有5个序列都通过了测试。
>
> **STATISTICAL TEST**：这是测试的名称，这里是近似熵测试（Approximate Entropy Test）

所有列出的随机数生成器在近似熵测试中都表现良好，通过率达到了100%，但是每一个算法的 P值分布都不相同。

就这次测试的 $5 \times 1MB$ 数据而言，`mt20095` 的P值主要集中在0.00 - 0.20区间，表明该生成器生成的序列在复杂性和不可预测性方面表现较差；`java.util.Random` 的P值比较均匀，表明在复杂性和不可预测性方面表现较好。

又以块内频率测试（BlockFrequency）为例，测试结果如下：

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1		c1	c2	c3	c4	c5	c6	c7	c8	c9	c10	P-value	partition	STATISTICAL TEST	
2	java.util.Random	0	0	0	0	0	1	0	3	1	0	----	5/5	Frequen	
3	java.util.Random	0	0	0	1	0	0	0	1	2	1	----	5/5	BlockFrequency	
201	java.security.SecureRandom	0	0	1	0	1	0	0	1	1	1	----	5/5	BlockFrequency	
401	Math_random()	1	0	1	0	1	1	1	0	0	0	----	5/5	BlockFrequency	
601	minstd_rand()	1	0	1	0	1	0	0	0	1	1	----	4/5	BlockFrequency	
801	mt19938	1	0	0	2	0	0	1	0	1	0	----	5/5	BlockFrequency	
1001	ranlux25_base	1	0	1	0	2	0	1	0	0	0	----	5/5	BlockFrequency	
1188															
1189															

- 发现 `java.util.Random`、`mt19938` 和 `java.security.SecureRandom` 在块内频率测试中表现较好，P值分布较为均匀，通过率较高
- `minstd_rand()` 在块内频率测试中表现较差，通过率略低，P值分布较为分散

6.3 总结

这些测试结果可以帮助我们初步评估不同随机数生成器的随机性质量，识别潜在问题，并指导我们在实际应用中选择合适的生成器。由于本次测试的样本量较小，测试结果不具有准确的参考价值。为了获得更全面和准确的评估，建议在更大样本量和更多测试条件下进行进一步的测试。通过这些严格的测试，可以筛选出高质量的随机数生成器，为保护信息安全提供坚实的基础。

七、总结

本文全面介绍了伪随机数生成器（PRNG）和真随机数生成器（TRNG）的原理、实现方法以及测试标准。

主要包括：

- **线性同余发生器（LCG）**：实现简单，计算速度快，但安全性较弱，容易被破解。
- **非线性同余发生器**：生成的随机数序列具有更好的随机性，能克服高位网络结构，但速度较慢。
- **Mersenne Twister（MT）**：周期极长，生成的随机数序列具有良好的统计特性，但初始化过程复杂，占用内存较多。
- **线性反馈移位寄存器（LFSR）**：基于移位和异或操作，实现简单，但输出序列可能在某些情况下表现出可预测性。
- **减法进位随机数生成器（SWC）**：计算速度快，周期较长，但初始状态敏感，线性结构可能被破解。

- **时滞斐波那契生成器（LFG）**：生成的随机数序列具有较好的随机性，实现简单，但周期相对较短，对参数选择敏感。

还详细讨论了不同编程语言中提供的随机数生成库及其内部实现机制。具体来说，我们分析了C/C++、Java、Python和Matlab等编程语言中的标准随机数生成库，以及它们的内部算法和实现细节，可以帮助我们理解了各种生成器的优缺点，以及它们在不同应用场景中的适用性。

进一步地，我们通过NIST SP 800-22测试套件对这些随机数生成器生成的10 MB 01随机序列进行了全面测试。这些测试包括了15个不同的指标，如频率测试、块内频率测试、累积和测试、运行测试、二进制矩阵秩测试等，旨在评估随机数序列的随机性、均匀性和不可预测性。通过这些测试，我们可以使用这些指标来筛选出高质量的随机数生成器，确保它们在密码学、数据分析、游戏开发等领域的应用中表现出色。

这些测试结果不仅提供了对不同生成器性能的初步评估，还帮助我们识别了某些生成器在特定测试中可能存在的问题。例如，某些生成器在近似熵测试中表现较差，而另一些在块内频率测试中通过率较低。这些发现为我们选择合适的随机数生成器提供了科学依据，确保在实际应用中能够使用到高质量、安全可靠的随机数生成器。

附录

所有测试数据和代码都保存在Git仓库中：github url

1. **定理**：如果 M 是一个质数，那么 a 是模 M 的一个原根当且仅当对于 $M-1$ 的每一个质因数 q，有 $a^{(M-1)/q} \not\equiv 1 \pmod{M}$。

证明：

充分性：假设对于 $M-1$ 的每一个质因数 q，有 $a^{(M-1)/q} \not\equiv 1 \pmod{M}$。设 d 是 a 模 M 的阶，即最小的正整数使得 $a^d \equiv 1 \pmod{M}$。由于 $a^{M-1} \equiv 1 \pmod{M}$（根据费马小定理），$d$ 必须整除 $M-1$。假设 $d \neq M-1$，则 d 必须是 $M-1$ 的一个真除数。设 q 是 $M-1$ 的任意质因数，使得 d 整除 $(M-1)/q$。那么 $a^{(M-1)/q} \equiv 1 \pmod{M}$，这与我们的假设矛盾。因此，$d$ 必须等于 $M-1$，即 a 是模 M 的一个原根。

必要性：假设 a 是模 M 的一个原根。那么 a 的阶是 $M-1$，即 $a^{M-1} \equiv 1 \pmod{M}$，并且 $M-1$ 是满足这个同余式的最小正整数。设 q 是 $M-1$ 的任意质因数。那么 $(M-1)/q$ 是一个正整数，并且 $(M-1)/q < M-1$。假设 $a^{(M-1)/q} \equiv 1 \pmod{M}$，这将意味着 a 的阶不是 $M-1$，而是 $(M-1)/q$ 或 $(M-1)/q$ 的某个除数，这与 a 是原根的假设矛盾。因此，对于 $M-1$ 的每一个质因数 q，有 $a^{(M-1)/q} \not\equiv 1 \pmod{M}$。

■

```
\pagestyle{fancy} % With this command we can customize the header style.

\fancyhf{} % This makes sure we do not have other information in our header or
footer.

\lhead{\footnotesize Discrete Mathematics(H): Project}% \lhead puts text in the
top left corner. \footnotesize sets our font to a smaller size.

%\rhead works just like \lhead (you can also use \chead)
\rhead{\footnotesizprojecte Mengxuan Wu} %<---- Fill in your lastnames.

% Similar commands work for the footer (\lfoot, \cfoot and \rfoot).
% We want to put our page number in the center.
\cfoot{\footnotesize \thepage}
```

测试环境说明

环境	版本
系统	win11家庭中文版
WSL	Ubuntu 22.04.5 LTS
NIST	NIST SP 800-22
C++编译器	8.1.0
Java	1.8.0

相关文件说明：

文件（夹）	描述
reference	部分参考资料
src/generator.cpp	C++ 使用不同函数 产生 01随机序列源码
RandomSequenceGenerator.java	java 产生01 随机序列源码
result/result.xlsx	6种 PRNG 使用 NIST 15个测试结果
result/AlgorithmTesting_java_Random	java中 Random 类的测试结果
data	记录了使用不同 PRNG 产生的 10 MB 01随机序列

reference：

- 综述_产生伪随机数的若干新方法——杨自强 2001
- 对伪随机数生成算法的随机性评价方法的研究——丁豪杰 2020
- 伪随机数生成器-维基百科，自由的百科全书 --- Cryptographically secure pseudorandom number generator - Wikipedia
- BBS(Blum-Blum—Shub)产生器 | quanquan
- xorshift算法生成随机数的原理是什么 – PingCode
- NIST 测试套件使用指南-CSDN博客
- 时滞斐波那契生成器