TRABAJO DE INVESTIGACIÓN E IMPLEMENTACIÓN DE ALGORITMOS QUE PERMITAN GENERAR NÚMEROS ALEATORIOS GRANDES Y ALGORITMOS QUE PRUEBEN LA PRIMALIDAD DE UN NÚMERO GRANDES..

Integrantes:

Xiomara Puma Torres

- Generador de congruencia lineal(LCG)
- Test de Primalidad Miller Rabin

Santiago San Roman Olazo

- Generador congruente permutable (PCG)

Algoritmos

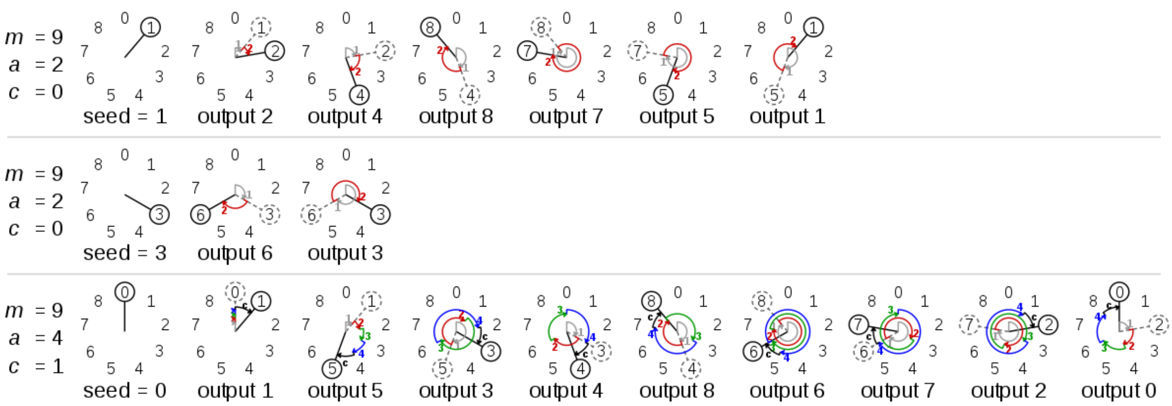
1. Generador de congruencia lineal (LCG)

A. Definición

Es un algoritmo que produce una secuencia de números pseudoaleatorios calculados calculados con una función lineal definida a trozos discontinua Un método congruencial comienza con un valor inicial (semilla) x_0, y los sucesivos valores x_n, n ≥ 1 se obtienen recursivamente con la siguiente fórmula

$$X_{I+1} = (ax_i + c) \bmod (m)$$

donde a, m y b son enteros positivos que se denominan, respectivamente, el multiplicador, el módulo y el incremento. Si b = 0, el generador se denomina multiplicativo; en caso contrario se llama mixto. La sucesión de números pseudo-aleatorios u_n, n ≥ 1 se obtiene haciendo $u_i = \frac{x_i}{m}$.



B. Seguimiento

X_{I+1}= (ax_i+c) mod (m) r_i= x_i/ m-1

- x_i : nùmeros enteros
- a , c , m son nùmeros enteros positivos

Semilla	k= nùmero entero	c= nùmero impar	g= nùmero entero
x0 = 6	a = 1 + 4k k = 8 a = 1+ 4(8) = 33	c = 5	m = 2^g g=2 m = 2^2= 4

x_1= (33 * 6 + 5) mod(4) = 203 mod 4 = 3 r_1= 3 / (4-1) =1
x_2= (33 * 3 + 5) mod(4) = 104 mod 4 = 0 r_1= 0 / (4-1) =0
x_3= (33 * 0 + 5) mod(4) = 5 mod 4 = 1 r_1= 1 / (4-1) =0.33
x_4= (33 * 1 + 5) mod(4) = 38 mod 4 = 2 r_1= 2 / (4-1) =0.66
x_5= (33 * 2 + 5) mod(4) = 71 mod 4 = 3 r_1= 3 / (4-1) =1

2. Generador congruente permutable (PCG)

A. Definición

Utiliza funciones de permutación en tuplas y una base de generador de congruencia lineal para determinar un número pseudoaleatorio de la forma más aleatoria posible. PCG es además una familia de generadores permutables a la que pertenecen los complejos Xorshift64+ Xorshift128+ y el sus sucesores Xoroshiro128+ utilizado en navegadores como Chrome, Safari, Firefox, etc. y Xoroshiro256+ (su version mas reciente).

B. Seguimiento

Solo agrega una permutación al LCG ya que se basa en este pero permitiendo cambios con relación a la permutación misma.

C. Implementación:

Por la complejidad de los códigos decidimos solo comparar dos de los miembros de la familia para explicar la diferencia entre ellos.

Xorshift 32 y Xorshift 64
https://github.com/BrutPitt/fastRandomGenerator

Xoroshiro128.hpp

```cpp
#ifndef AJC_HGUARD_XOROSHIRO128PLUS
#define AJC_HGUARD_XOROSHIRO128PLUS

#include <random>
#include <array>

/* C++11 UniformRandomBitGenerator wrapper around xoroshift128+.
 * Original C code from: http://xoroshiro.di.unimi.it/xoroshiro128plus.c */

struct xoroshiro128plus_engine {
private:
  uint64_t state[2];

  static inline uint64_t rotl(const uint64_t x, int k) {
    return (x << k) | (x >> (64 - k));
  }

public:
  using result_type = uint64_t;
  constexpr static result_type min() { return 0; }
  constexpr static result_type max() { return -1; }

  result_type operator()();
  void seed(std::function<uint32_t(void)>);
  void seed(const std::array<uint32_t, 4> &);
};

#endif
```

Xoroshiro128.cpp

```cpp
#include "xoroshiro.hpp"

/* C++11 UniformRandomBitGenerator wrapper around xoroshift128+.
 * Original C code from: http://xoroshiro.di.unimi.it/xoroshiro128plus.c */

xoroshiro128plus_engine::result_type
xoroshiro128plus_engine::operator()() {
  const uint64_t s0 = state[0];
  uint64_t s1 = state[1];
  const uint64_t result = s0 + s1;

  s1 ^= s0;
  state[0] = rotl(s0, 55) ^ s1 ^ (s1 << 14); // a, b
  state[1] = rotl(s1, 36); // c

  return result;
}

void xoroshiro128plus_engine::seed(std::function<uint32_t(void)> f) {
  uint64_t x_0 = f();
  uint64_t x_1 = f();
  state[0] = (x_0 << 32) | x_1;
  x_0 = f();
  x_1 = f();
  state[1] = (x_0 << 32) | x_1;
}

void xoroshiro128plus_engine::seed(const std::array<uint32_t, 4> &a) {
  state[0] = ((uint64_t)a[0] << 32) | (uint64_t)a[1];
  state[1] = ((uint64_t)a[2] << 32) | (uint64_t)a[3];
}
```

repo: https://github.com/alexcoplan/xoroshiro128plus-cpp

3. Test de Primalidad Miller Rabin

A. Definición

Es un algoritmo para determinar si un número dado es primo, similar al test de primalidad de Fermat.

Hecho : Sea n un primo impar y n - 1≡ 2^sr donde r es impar. Sea a cualquier número entero tal que mcd (a, n) = 1. Entonces $a^r \equiv 1$ (mod n) o $a^{2jr} \equiv -1$ (mod n) para algunos j, 0 ≤ j ≤ (s - 1)

Sea n un entero compuesto impar y sea n - 1 = 2^sr donde r es impar. Deja un ser un número entero en el intervalo [1, n - 1].

- ➔ Si $a^{r} \neq 1$ (mod n) y si $a^{2^{j}r} \neq -1$ (mod n) para todo j, 0 ≤ j ≤ s - 1, entonces a es llamado strong witness (to compositeness) para n.
- ➔ De lo contrario, es decir, si ar ≡ 1 (mod n) o a2j r ≡ −1 (mod n) para algún j, 0 ≤ j ≤ s - 1, entonces se dice que n es un pseudoprime fuerte a la base a. (Es decir, n actos como un primo en el sentido de que satisface para la base particular

B. Seguimiento

Pasos

1. Encontrar n -1 = 2^{k}.m
2. Escoger a : 1 < a < n-1
3. Compute $b_0 = a^{m}$mod n , $b_i = b_{i-1}^2$

Ejemplo :

¿ 53 es primo ?

Paso (1)

n-1 = 2^{k}m   n=53

53-1 = 2^{k}m   k,m son nùmeros entero

$\frac{52}{2}$ = 26	$\frac{52}{2^2}$ = 13	$\frac{52}{2^3}$= 6. 5 x

52 = 2^{2}. 13   k=2 , m =13

Paso(2)

1 < a < n - 1   a = 2

1 < a < 52

Paso (3)

$b_0 = a^{m}$. mod n

$b_0 = 2^{13}$. mod 53 = 30 mod 53

b_0= +1 o -1 -> n es primo (probablemente)

b_0 = 30

$b_1 = 30^{2}$.mod 53 = -1 mod 53

Si b_1= +1 -> compuesto

Si b_1= -1 -> primo (probablemente)

Por lo tanto 53 es primo (probablemente).

C. Pseudo Algoritmo

Algorithm Miller-Rabin probabilistic primality test

MILLER-RABIN(n,t)
INPUT: an odd integer $n \geq 3$ and security parameter $t \geq 1$.
OUTPUT: an answer "prime" or "composite" to the question: "Is n prime?"

1. Write $n - 1 = 2^s r$ such that r is odd.
2. For i from 1 to t do the following:
   - 2.1 Choose a random integer a, $2 \leq a \leq n - 2$.
   - 2.2 Compute $y = a^r \bmod n$ using Algorithm 2.143.
   - 2.3 If $y \neq 1$ and $y \neq n - 1$ then do the following:
     - $j \leftarrow 1$.
     - While $j \leq s - 1$ and $y \neq n - 1$ do the following:
       - Compute $y \leftarrow y^2 \bmod n$.
       - If $y = 1$ then return("composite").
       - $j \leftarrow j + 1$.
     - If $y \neq n - 1$ then return ("composite").
3. Return("prime").

D. Implementaciòn c++

```cpp
bool testPrimalidadMillerRabin(ZZ n,int loops)
{
    ZZ a;
    a = n - to_ZZ("1");
    vector<ZZ> values;
    int i=0;
    while(a != 0)
    {
        i++;
        a /= to_ZZ("2");
        if(a>>1<<1 != a)
        {
            values.push_back(to_ZZ(i));
            values.push_back(a);
        }
    }
    ZZ s = values[0];
    ZZ r = values[1];
    ZZ j;
    for(j =0;j<loops;j++)
    {
        ZZ rndom = modulo(aleatorio(),n-2)+2;
        ZZ y = expo_modular_rapida(rndom,r,n);
        if(y != 1 && y!= n - to_ZZ("1"))
        {
            ZZ j = to_ZZ("1");
            while(j<=s-to_ZZ("1") && y!=n-to_ZZ("1"))
            {
                y = expo_modular_rapida(y,to_ZZ("2"),n);
                if(y == 1)
                {
                    return 0;  // es falso
                }
                j++;
            }
            if(y != n-to_ZZ("1"))
                return false;
        }
    }
    return true;
}
```

Link Github :
https://github.com/xiop-torres/XiomaraPumaAA/tree/main/Miller-RabinTest

Bibliografia

[1] Manuel A. Pulido Cayuela "Generación de números aleatorios" Available: https://webs.um.es/mpulido/miwiki/lib/exe/fetch.php?media=wiki:simt1b.pdf

[2] Thompsom, Sam " Random number generators for C++ performance tested", March 14, 2019. Available: https://thompsonsed.co.uk/random-number-generators-for-c-performance-tested

[3] Img. 1, Wikipedia: https://es.wikipedia.org/wiki/Generador_lineal_congruencial

[4] A. Menezes, P. van Oorschot, and S. Vanstone "Handbook of Applied Cryptography" Chapter 4, CRC Press, 1996. Available: http://cacr.uwaterloo.ca/hac/about/chap4.pdf

[5] Michele Morrone, 2020. https://github.com/BrutPitt/fastRandomGenerator

[6] Alex Coplan, 2016. https://github.com/alexcoplan/xoroshiro128plus-cpp

[7] Melissa O'Neill "PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation", 2018. Available: https://www.pcg-random.org/paper.html