



ugr

Universidad  
de Granada

TRABAJO FIN DE GRADO  
INGENIERÍA EN INFORMÁTICA

# Secuencias pseudoaleatorias

---

Análisis estadístico de secuencias pseudoaleatorias

**Autor**

Manuel Gutiérrez Delgado

**Director**

Jesús García Miranda



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE  
TELECOMUNICACIÓN

Granada, Noviembre de 2018







# Secuencias pseudoaleatorias

---

Análisis estadístico de secuencias pseudoaleatorias.

**Autor**

Manuel Gutiérrez Delgado

**Director**

Jesús García Miranda



# Secuencias pseudoaleatorias: Análisis estadístico de secuencias pseudoaleatorias

Manuel Gutiérrez Delgado

**Palabras clave:** LFSR, test estadísticos, complejidad lineal, secuencias pseudoaleatorias, aleatoriedad.

## Resumen

En este proyecto se pretende desarrollar un programa que genere secuencias de bits de tamaño grande y que puedan ser consideradas como aleatorias (secuencias pseudoaleatorias).

Para esto, generaremos tales secuencias, analizaremos su aleatoriedad a partir de diversos test estadísticos y trataremos de mejorar esta aleatoriedad combinando distintas secuencias.

Para generar dichas secuencias usaremos los Registro de Desplazamiento con Retroalimentación Lineal (LFSR). Estos generadores, bajo ciertas condiciones, producen buenas secuencias pseudoaleatorias, pero tienen un gran inconveniente y es su baja complejidad lineal, lo que hace que sean altamente predecibles y una pequeña porción de la secuencia permite reproducirla entera. Por tanto, veremos cómo la combinación de distintas secuencias mediante operaciones a nivel de bits mantienen la "pseudoaleatoriedad" mejoran sustancialmente la complejidad lineal.





# Pseudorandom sequences: Statistical analysis of pseudorandom sequences

Manuel Gutiérrez Delgado

**Keywords:** LFSR, statistical tests, linear complexity, pseudo-random sequences, randomness.

## Abstract

The aim of this project is to develop a program to generate sequences of large bits that can be considered as random (pseudorandom sequences).

For this, we will generate such sequences, analyze their randomness from various statistical tests and try to improve this randomness by combining different sequences.

To generate these sequences we will use the Displacement Register with Linear Feedback (LFSR). These generators, under certain conditions, produce good pseudorandom sequences, but they have a great disadvantage and its low linear complexity, which makes them highly predictable and a small portion of the sequence allows to reproduce it entirely. Therefore, we will see how the combination of different sequences through operations at the bit level maintains the "pseudo-randomness" and substantially improves the linear complexity.



---

Yo, **Manuel Gutiérrez Delgado**, alumno de la titulación Grado en Ingeniería Informática de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 75929835G, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Manuel Gutiérrez Delgado

Granada a 15 de Noviembre de 2018.



---

D. **Jesús García Miranda**, Profesor del Departamento de Álgebra de la Universidad de Granada.

**Informa:**

Que el presente trabajo, titulado ***Secuencias pseudoaleatorias, análisis estadístico de secuencias pseudoaleatorias***, ha sido realizado bajo su supervisión por **Manuel Gutiérrez Delgado**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 15 de Noviembre de 2018.

**El director:**

**Jesús García Miranda**



# Agradecimientos

Muchas gracias a mis padres y mis hermanos por el apoyo todos estos años de grado, especialmente en los años más duros, a mi compañero, que ya es amigo, por el apoyo cada vez que lo he necesitado y a mis amigos/as que se han preocupado por mí y me han ayudado. Por último gracias a profesores y sobre todo a mi tutor de TFG que ha quedado para una tutoría siempre que hacía falta sin poner ninguna pega, hasta en el mes de agosto que la escuela está cerrada.





# Índice general

<b>1. Introducción</b>	<b>17</b>
1.1. Secuencias pseudoaleatorias	18
1.2. Generadores de secuencias pseudo-aleatorias	18
<b>2. Objetivos y Motivación</b>	<b>21</b>
2.1. Objetivos	21
2.2. Motivación	21
<b>3. Planificación</b>	<b>25</b>
<b>4. Diseño y uso</b>	<b>27</b>
4.1. Diseño	27
4.2. Uso	31
<b>5. Implementación</b>	<b>43</b>
5.1. LFSR (Linear Feedback Shift Register)	43
5.2. Algoritmo de Berlekamp-Massey	46
5.3. Test de Frecuencia (Monobit)	47
5.4. Test de Frecuencia por Bloques	48
5.5. Test de Rachas	48
5.6. Test de Complejidad Lineal	49
<b>6. Pruebas</b>	<b>51</b>
6.1. Prueba1	51
6.1.1. Polinomio irreducible	51
6.1.2. Polinomio irreducible y primitivo	52
6.1.3. Polinomio reducible	53
6.2. Prueba2	54
6.3. Prueba3	55
6.4. Prueba4	59
6.5. Prueba5	63

6.6. Prueba6 . . . . .	68
7. Conclusiones	73
Bibliografía	75

# Índice de figuras

3.1. Planificación del proyecto . . . . .	26
4.1. Diagrama de flujo, menú principal . . . . .	28
4.2. Diagrama de flujo, opción 1 (introducir secuencia) . . . . .	29
4.3. Diagrama de flujo, opción 2 (analizar secuencia) . . . . .	30
4.4. Menú principal. . . . .	31
4.5. Generar una secuencia con un LFSR. . . . .	32
4.6. Combinar con el operador AND dos secuencias. . . . .	33
4.7. Leer secuencia de un fichero. . . . .	34
4.8. Elegir los dos 'XOR' de la función mayoría. . . . .	35
4.9. Realizar primer 'AND' de la función mayoría . . . . .	36
4.10. Realizar segundo 'AND' y por tanto segundo 'XOR' de la función mayoría . . . . .	37
4.11. Solución final tras realizar el segundo 'AND' y por tanto primer 'XOR' de la función mayoría. . . . .	38
4.12. Berlekamp-Massey. . . . .	39
4.13. Análisis test de frecuencia de complejidad lineal combinando dos secuencias. . . . .	40
4.14. Contar rachas . . . . .	41
4.15. Salir del menú . . . . .	42



# Capítulo 1

## Introducción

Actualmente, los métodos criptográficos se dividen en dos grandes apartados: cifrado en bloque y cifrado en flujo. Estos se basan en el cifrado de *Vernam* [5], que consiste en generar una secuencia aleatoria de bits y hacer un “exclusivo” bit a bit con los bits del mensaje a cifrar. ¿Qué significa secuencia aleatoria?

Aleatorio, según la RAE es, *‘que depende del azar o no sigue una pauta definida’*.

El azar según la RAE es, *‘casualidad, caso fortuito’*.

Por tanto, una secuencia aleatoria, puede ser una sucesión de valores obtenidos al azar. Esto supone que el conocimiento de unos términos de la secuencia no nos dan ninguna información sobre el siguiente término. Algunos métodos para generar secuencias aleatorias son: lanzar una moneda al aire varias veces, lanzar un dado, girar una ruleta etc.

Nosotros vamos a centrarnos en secuencias donde cada término puede ser elegido entre dos posibles valores (que representamos como 0 o 1) y la probabilidad de que aparezca uno de ellos es igual a la probabilidad de que aparezca el otro. El lanzamiento de una moneda es un experimento para generar una secuencia aleatoria pero es muy lento, por lo que necesitamos otros más rápidos, como por ejemplo la lectura del reloj o los movimientos del ratón del ordenador.

Pero estos generadores tienen un inconveniente. No se pueden reproducir de esta forma, una vez cifrado un mensaje sería imposible de descifrar, pues no se podría conseguir la secuencia clave a menos que la enviemos por un canal seguro. Pero en tal caso, ¿de qué nos sirve cifrarla? Dicho de otra forma, el sistema de cifrado propuesto por *Vernam* es imposible llevarlo a la práctica, por lo que se intenta buscar aproximaciones lo más fiables posibles

a este cifrado.

## 1.1. Secuencias pseudoaleatorias

Surgen así las secuencias pseudoaleatorias. Son secuencias que aparentan ser aleatorias, pero están generadas por un proceso determinista, lo cual permite reproducirlas. ¿Qué significa ser “aparentemente aleatorio”?

A lo largo de la historia se han dado múltiples criterios para ver si una secuencia satisface los criterios de aleatoriedad.

Uno muy sencillo fue propuestos por Golomb [3], que son 3 postulados para que una secuencia periódica sea considerada aleatoria. Estos postulados son:

1. *‘En todo período la diferencia entre el número de unos y el número de ceros debe ser a lo sumo uno’.*
2. *‘En todo período existen grupos de caracteres (unos o ceros) repetidos, de forma que la mitad de los grupos son de un solo elemento, la cuarta parte de dos elementos, la octava parte de tres elementos y así sucesivamente, es decir, el número de rachas de un tamaño tiene que ser la mitad del número de rachas del tamaño anterior. Además, para cada uno de los grupos de una longitud determinada hay el mismo número de ceros y de unos’.*
3. *‘La distancia de Hamming entre cualesquiera dos secuencias diferentes obtenidas mediante desplazamientos circulares de otra generada por un LFSR es constante, es decir, las secuencias generadas por un LFSR son autocorreladas’.*

Estos postulados vienen a recoger la idea de que dado un segmento de la secuencia, la posibilidad de que el siguiente bit sea 0 es aproximadamente la misma de que sea 1.

Sin embargo, estos postulados son muy rígidos, lo que hace que las secuencias que los satisfacen sean a veces muy predecibles. Por tanto, lo que vamos a hacer es rebajar estas condiciones (buscando secuencias que “mas o menos” cumplan estos postulados).

## 1.2. Generadores de secuencias pseudo-aleatorias

Un generador de secuencias pseudoaleatorias es un algoritmo que produce un resultado muy parecido a una secuencia aleatoria.

Los algoritmos mas usados suelen ser generadores lineales congruentes, generadores de registro de desplazamiento con retroalimentación lineal (LFSR) y los generadores de desplazamiento con retroalimentación no lineal (NLFSR).

Los generadores lineales congruentes generan una sucesión  $x_0, x_1, x_n$  donde  $x_{n+1} = (ax_n + c)(\text{mod } m)$ , dependen de una semilla  $x_0$ , una constante multiplicativa  $a$ , una constante aditiva  $c$  y el módulo de la variable  $m$ , donde  $x_0, a, c < m$  y sean enteros positivos. Para obtener una secuencia de bits se toma  $m = 2^k$ , y entonces se van añadiendo los  $k$  bits de la expresión binaria de  $x_n$ .

Estos generadores debido a que el resultado es un sencillo cifrado aún no deben utilizarse para aplicaciones criptográficas.

Los NLFSR no los vamos a utilizar ya que son no lineales.

Por los motivos anteriores, hemos decidido utilizar el generador de secuencias pseudoaleatorias de registro de desplazamiento con retroalimentación lineal (LFSR), ya que tiene buenas características de aleatoriedad y generan secuencias con bastante rapidez y de forma sencilla.

Antes de pasar a detalles tan técnicos, en los capítulos previos se explicarán otros aspectos del trabajo como son:

- En el *capítulo 2 (Objetivos y Motivación)*, explicamos que es lo que pretendemos obtener con este trabajo y porque pensamos que eso puede ser interesante.
- En el *capítulo 3 (Planificación)*, damos con más detalle como hemos planificado las distintas etapas en las que se ha desarrollado el proyecto para que se haya podido llegar a tiempo a la fecha de entrega.
- En el *capítulo 4 (Diseño y uso)*, vemos los distintos módulos en los que se ha desarrollado el proyecto y explicamos como puede ser usado nuestro programa, para facilitar su uso.

Realizados los apartados anteriores faltaría el *capítulo 5 (Implementación)*, que explicamos como hemos implementado los distintos algoritmos, funciones y test que se van a utilizar.

Una vez que se han implementado todos los algoritmos, vamos a hacer las pruebas, que algunos de los resultados que hemos considerado más relevantes y los hemos explicado en el *capítulo 6 (Pruebas)*.

Ya solo quedaría tras realizar las pruebas, sacar las conclusiones que van en el *capítulo 7 (Conclusiones)*, en el que se expondrán los conocimientos que se han sacado durante el desarrollo del trabajo.





## Capítulo 2

# Objetivos y Motivación

### 2.1. Objetivos

El objetivo de este trabajo es la generación de secuencias mediante LFSR, analizarlas y basándonos en ellas corregir las posibles diferencias que encontramos.

Esto lo hacemos en 3 etapas:

1. Realizamos secuencias pseudoaleatorias.
2. Solucionar los problemas de complejidad lineal de las secuencias generadas para que no sean fácilmente predecibles.
3. Implementar los diferentes test para analizar dichas secuencias, test de frecuencia (monobit), test de frecuencia por bloque, test de rachas y test de complejidad lineal.

### 2.2. Motivación

El principal inconveniente que presentan las secuencias generadas por LFSR es su baja complejidad lineal. Vamos a ver que significa esto. Para esto, tomamos la secuencia, '1011101101000110000100111111101110001111000000111011011000101000100110010000011010010011110111110001010101101000010101000000101101101111001111000100011111101100011101011010100001100110110000011000000001101101101011101011110000101010010000101100100110000010001001000000100000000010010010011010011010111110011000111110010001110111111000011100000001111111111000111000100111011001010110111101010001111010010001111010010101000001011111111010101010111101000011

10100100011001011010110011110101100011001111110010101', a la que un análisis a simple vista nos dice que podría parecer aleatoria. Imaginemos que cogemos 20 bits de esta secuencia '11010010101000001011'. A partir de aquí planteamos un sistema de ecuaciones con estos datos, cuando se hable de los LFSRs se entenderá.

$$\begin{aligned}
 1 &= a_2 + a_4 + a_7 + a_9 + a_{10} \\
 0 &= a_1 + a_3 + a_5 + a_8 + a_{10} \\
 0 &= a_2 + a_4 + a_6 + a_9 \\
 0 &= a_3 + a_5 + a_7 + a_{10} \\
 0 &= a_4 + a_6 + a_8 \\
 0 &= a_5 + a_7 + a_9 \\
 1 &= a_6 + a_8 + a_{10} \\
 0 &= a_1 + a_7 + a_9 \\
 1 &= a_2 + a_8 + a_{10} \\
 1 &= a_1 + a_3 + a_9
 \end{aligned}$$

Vamos a ver un ejemplo de como hemos sacado las ecuaciones anteriores.

La primera ecuación  $1 = a_2 + a_4 + a_7 + a_9 + a_{10}$  se ha generado de la siguiente forma: el 1 que es el bit número 11 de la sucesión que hemos interceptado y los coeficientes  $a_2 + a_4 + a_7 + a_9 + a_{10}$  viene de que los bits que están 2 a la izquierda, 4 a la izquierda, 7 a la izquierda, 9 a la izquierda y 10 a la izquierda de este son unos. La segunda ecuación  $0 = a_1 + a_3 + a_5 + a_8 + a_{10}$  se ha generado de la siguiente forma: el 0 que es el bit número 12 de la sucesión que hemos interceptado y los coeficientes  $a_1 + a_3 + a_5 + a_8 + a_{10}$  viene de que los bits que están 1 a la izquierda, 3 a la izquierda, 5 a la izquierda, 8 a la izquierda y 10 a la izquierda de este son unos. Y así el resto de ecuaciones.

La solución es:

$$\begin{aligned}
 'a_0 = 1', 'a_1 = 0', 'a_2 = 0', 'a_3 = 1', 'a_4 = 0', 'a_5 = 0', 'a_6 = 0', 'a_7 = 0', \\
 'a_8 = 0', 'a_9 = 0', 'a_{10} = 1'
 \end{aligned}$$

Con esta solución podríamos generar toda la secuencia de la siguiente forma:  $S_{j+10} = S_{j+7} + S_j$

Esto significa que la complejidad lineal de esta sucesión es muy baja y con esto podemos decir que el gran enemigo en la criptografía de las secuencias pseudoaleatorias es la complejidad lineal.

Voy a coger otros 10 bits al azar para ver que con la fórmula anterior se puede sacar la secuencia, voy a sacar 3 bits como ejemplo.

De esta parte de la secuencia 1001111010110 (13 bits), imaginemos que solo tengo los 10 primeros bits, por tanto, tengo que sacar los últimos 3 bits que son **110**.

$$S_{j+7} + S_j = 0 + 1 = \mathbf{1}, \text{ vemos que es correcto.}$$

$S_{j+7} + S_j = 1 + 0 = \mathbf{1}$ , vemos que es correcto.

$S_{j+7} + S_j = 0 + 0 = \mathbf{0}$ , vemos que es correcto.

Yo pienso que este trabajo es una buena herramienta para los estudiantes por si quieren estudiar cuando una secuencia es aleatoria o no y quieren generar secuencias aleatorias. Para dar más posibilidades de generar secuencias hemos introducido un segundo LFSR que permite generar secuencias que son periódicas pero a partir de un cierto número, por tanto, se pueden considerar que estas secuencias no son periódicas. La función lineal de esta secuencia se calcula de la siguiente forma:  $S_{j+d} = a_{d-1}S_{j+d-1} + \dots + a_d S_j$

Un ejemplo de este LFSR es: **101**1110100111010011101001110100111010011101001110100111010011101001110100111010011101001110100111010

Si nos fijamos podemos ver que la parte de la secuencia que está en negrita no se repite. Entonces si cogemos y calculamos el periodo sin los 3 primeros bits, vemos que la secuencia tiene periodo 7, por tanto, a partir del bit 4 (incluido el 4) la secuencia es periódica.

Para que no tenga periodo, el término independiente del polinomio que utilizamos para generar la secuencia tiene que ser 0.



## Capítulo 3

# Planificación

El trabajo tuve que comenzar en Marzo por diversas circunstancias. Para poder realizar este trabajo fue necesario una primera fase de estudio e investigación, que aunque posteriormente no ha parado, se ha desarrollado con más intensidad el primer mes. En esta fase tuve una primera toma de contacto con la aleatoriedad y los distintos generadores aleatorios para llevar a cabo la implementación. Una segunda fase en la que se estudia la estructura del código y la aplicación. Una vez planificada la estructura, vemos la fase más larga, la de implementación de las diferentes funciones y algoritmos. Mientras implementaba los algoritmos iba realizando las diferentes pruebas de aleatoriedad para así poder ver su correcto funcionamiento y poder continuar con la implementación. Después realicé numerosas pruebas que algunas de ellas están incluidas en la memoria. Al final, para acabar solo quedaba realizar la memoria con toda la información que hemos considerado relevante, para esto he elegido hacerla en latex [2] lo cual me llevó un tiempo de aprendizaje del funcionamiento.

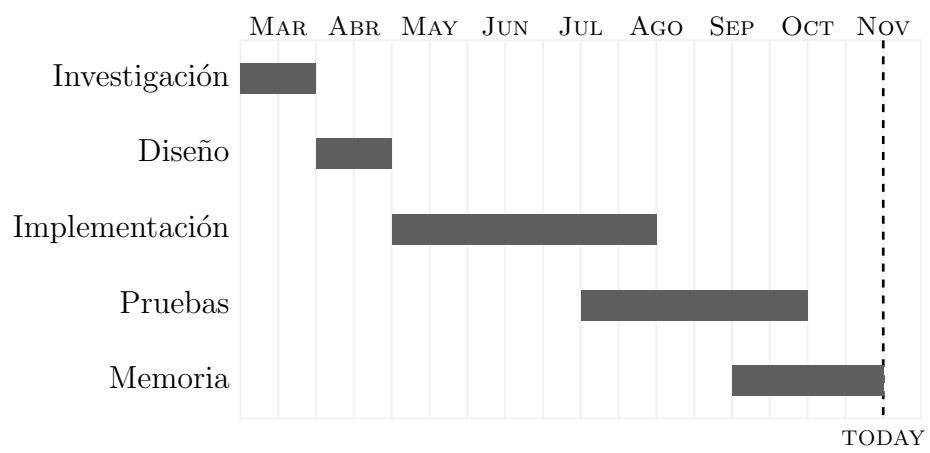


Figura 3.1: Planificación del proyecto

# Capítulo 4

## Diseño y uso

### 4.1. Diseño

He procurado que el programa que he generado sea fácil de usar, que permita todas las posibilidades, una vez que entras, de generar diferentes secuencias sin tener que guardarlas y nos permita guardar dichas secuencias si queremos, para después volver a usarlas, analizarlas y combinarlas sin tener que hacer una a una a parte entrando y saliendo del programa.

El programa se divide en 7 módulos distintos. Los módulos son: `main.py`, `menus.py`, `lfsr.py`, `BM.py`, `test.py`, `operadores.py`, `guardarYleer.py`.

Dicho esto, el `main.py` llama a `menus.py` que a su vez llama a los archivos `lfsr.py`, `BM.py`, `test.py`, `operadores.py` y el archivo `test.py` llama al `BM.py`.

Para diseño del programa, he realizado el diagrama de flujo o de actividades, que lo he separado en tres partes para que sea más legible. Las partes son:

1. El menú principal y sus tres opciones posteriores, generar, analizar y salir.
2. La opción 1 del menú principal: generar, introducir por teclado o leer de archivo, combinar secuencias y guardarlas si queremos para seguir realizando combinaciones o test con las mismas.
3. La opción 2 del menú principal: analizar secuencias, ya sea generándolas, introduciéndolas por teclado o leerlas de un archivo, combinándolas y guardarlas si queremos para seguir realizando combinaciones o test con las mismas.

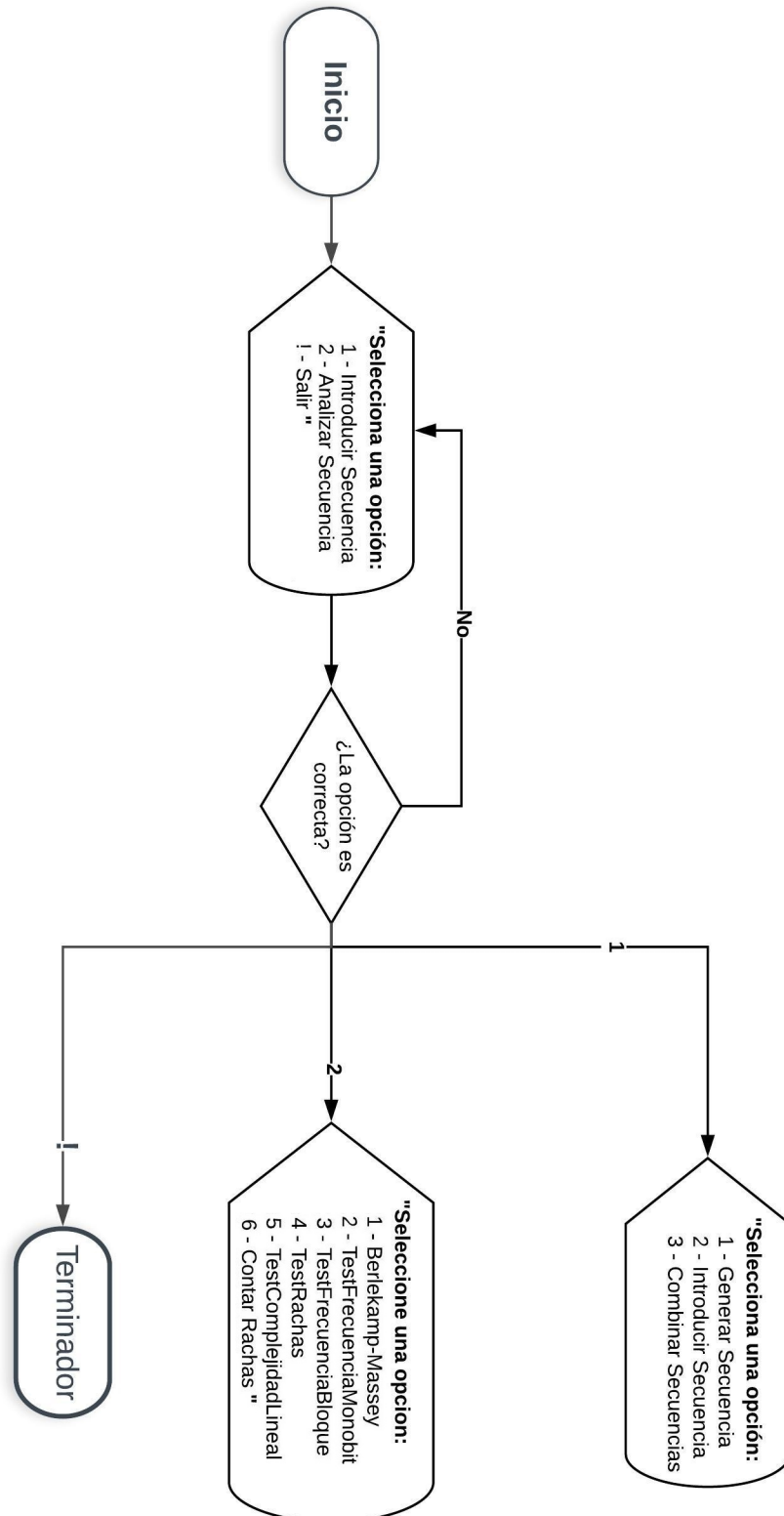


Figura 4.1: Diagrama de flujo, menú principal



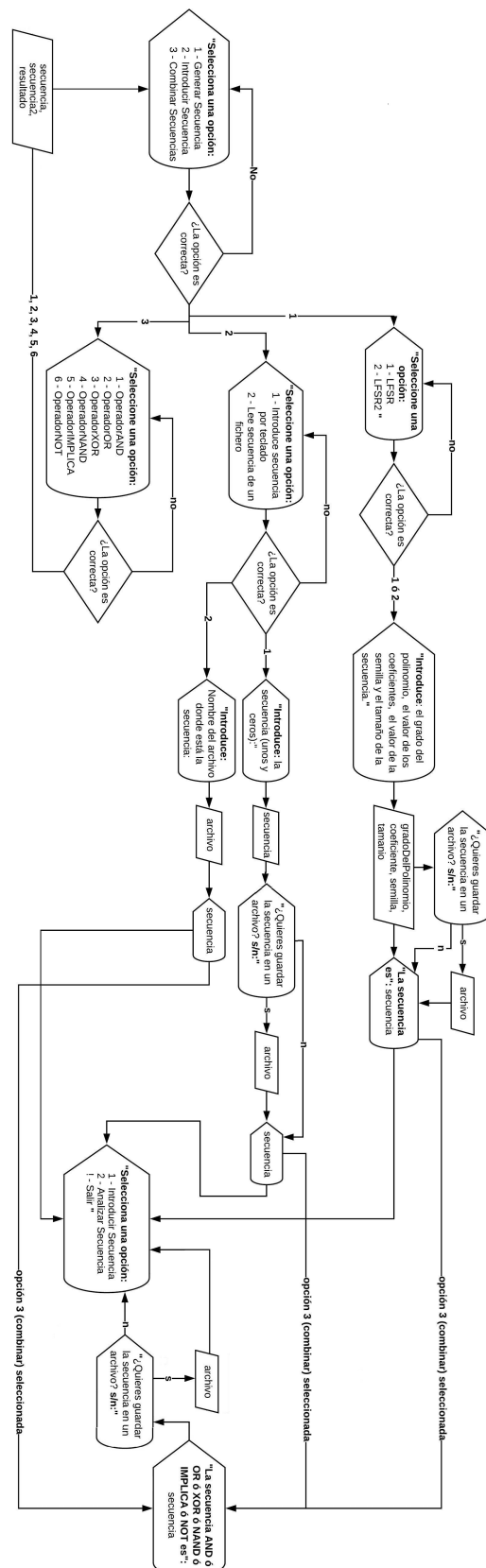
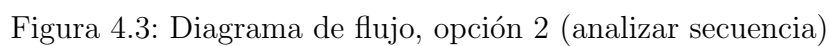


Figura 4.2: Diagrama de flujo, opción 1 (introducir secuencia)

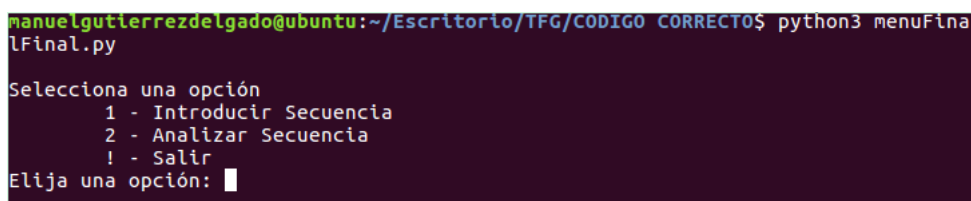


## 4.2. Uso

Para ejecutar el código hay que tener **"Python3"** instalado y hay que escribimos en la línea de comandos **"python3 main.py"** .

En este apartado voy a explicar el funcionamiento. Es un menú con diferentes opciones para que podamos realizar secuencias pseudoaleatorias, combinarlas con distintos operadores y hacer los test a las secuencias generadas y combinadas para ver su aleatoriedad. También nos permite guardar todas las secuencias que generemos, combinemos o introduzcamos por teclado, para así poder utilizarlas más veces.

- En la primera imagen vemos el menú principal (*función menu()*), donde tenemos 3 opciones para elegir (introducir secuencia, analizar secuencia y salir), donde introducir secuencia nos permite generar, introducir por teclado o desde archivo y combinar secuencias y analizar nos permite analizar cualquier secuencia con todos los test estadísticos, el algoritmo de Berlekamp-Massey y contar las rachas de las secuencias.



```
manuelgutierrezdelgado@ubuntu:~/Escritorio/TFG/CODIGO CORRECTO$ python3 menuFinal.py
Selecciona una opción
    1 - Introducir Secuencia
    2 - Analizar Secuencia
    ! - Salir
Elija una opción: █
```

Figura 4.4: Menú principal.

- A continuación, vamos a ver como se genera una secuencia pseudoaleatoria con un LFSR, es igual para el LFSR y el LFSR2 (*función LFSR()*). Como vemos en la imagen, elegimos las opciones en el orden siguiente:
  1. Introducir secuencia.
  2. Generar secuencia.
  3. LFSR ó LFSR2.
  4. Realizados los pasos anteriores, introducimos el grado del polinomio, el valor de cada coeficiente, la semilla y el tamaño que queramos que tenga la secuencia. Pulsamos *'Enter'* y se genera la secuencia (lo vemos en la imagen siguiente).

```

Selecciona una opción
    1 - Introducir Secuencia
    2 - Analizar Secuencia
    ! - Salir
Elija una opción: 1

Selecciona una opción
    1 - Generar Secuencia
    2 - Introducir Secuencia
    3 - Combinar Secuencias
Elija una opción: 1

Selezionare una opción:
    1 - LFSR
    2 - LFSR2
Elija una opción: 1
Introduce el grado del polinomio: 5
Introduce el valor del coeficiente (uno ó cero): 1
(1)x^5
(5,)
Introduce el valor del coeficiente (uno ó cero): 1
(1)x^4
(5, 4)
Introduce el valor del coeficiente (uno ó cero): 0
(0)x^3
Introduce el valor del coeficiente (uno ó cero): 0
(0)x^2
Introduce el valor del coeficiente (uno ó cero): 1
(1)x^1
(5, 4, 1)
Introduce el valor del coeficiente (uno ó cero): 1
(1)x^0
(5, 4, 1, 0)
Introduce el valor de la semilla (unos y ceros): 10111
Introduce el tamaño de la secuencia: 1000000
Counter({'1': 750000, '0': 250000})
Quieres guardar la secuencia en un archivo? (s, n):

```

Figura 4.5: Generar una secuencia con un LFSR.

- Ahora, vamos a ver como combinar secuencias (*función menuCombinar()*). Seguimos este orden:
  1. Introducir secuencia.
  2. Combinar secuencia.
  3. Elegimos el operador que queramos utilizar, en este caso he elegido el operador AND (*función operadorAND()*).
  4. Elegimos la opción que queramos para introducir la secuencia (*función menuIntroducirSecuencia()*), para este ejemplo he elegido introducir la secuencia por teclado, pero las opciones son:
    - a) Generar secuencia (*función menuGenerar()*).
    - b) Introducir secuencia, que puede ser por teclado o leerla de un archivo *función menuIntroduceSecuencia()*.

5. En el siguiente paso podemos elegir entre introducir otra secuencia, para hacer ya la operación o combinar otra secuencia, para así combinar todas las secuencias que queramos. Si elegimos combinar secuencia otra vez, volveríamos a hacer el paso anterior, por el contrario, si elegimos la opción de introducir la secuencia ya realizaría la operación y nos mostraría el resultado como vemos en la imagen siguiente.

```
Selecciona una opción
  1 - Introducir Secuencia
  2 - Analizar Secuencia
  ! - Salir
Elija una opción: 1

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 3

Selecione una opción:
  1 - OperadorAND
  2 - OperadorOR
  3 - OperadorXOR
  4 - OperadorNAND
  5 - OperadorIMPLICA
  6 - OperadorNOT
Elija una opción: 1

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 2

Selecione una opción:
  1 - Introduce la secuencia por teclado
  2 - Lee secuencia de un fichero
Elija una opción: 1
Introduce la secuencia (unos y ceros): 10111011
Quieres guardar la secuencia en un archivo? (s, n):
n

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 2

Selecione una opción:
  1 - Introduce la secuencia por teclado
  2 - Lee secuencia de un fichero
Elija una opción: 1
Introduce la secuencia (unos y ceros): 10000000
Quieres guardar la secuencia en un archivo? (s, n):
n
La secuencia AND es: 10000000
```

Figura 4.6: Combinar con el operador AND dos secuencias.

- Visto ya como combinar dos secuencias, vamos a hacer la función mayoría que está formada por tres 'AND' y dos 'XOR' y se utilizan tres secuencias. La función mayoría elige el bit más común de cada posición de las secuencias. La combinación es:

$(x \text{ AND } y) \text{ XOR } (y \text{ AND } z) \text{ XOR } (z \text{ AND } x).$

Antes de poner los pasos para la función mayoría, vamos a ver como se puede leer una secuencia de un archivo.

1. Introducir secuencia.
2. Introducir secuencia.
3. Lee secuencia de un fichero y introducimos el nombre del fichero.

```
manuelgutierrezdelgado@ubuntu:~/Escritorio/TFG/CODIGO
lFinal.py

Selecciona una opción
    1 - Introducir Secuencia
    2 - Analizar Secuencia
    ! - Salir
Elija una opción: 1

Selecciona una opción
    1 - Generar Secuencia
    2 - Introducir Secuencia
    3 - Combinar Secuencias
Elija una opción: 2

Seleccione una opción:
    1 - Introduce la secuencia por teclado
    2 - Lee secuencia de un fichero
Elija una opción: 2
Nombre del archivo donde está la secuencia:
secuencia1.txt
```

Figura 4.7: Leer secuencia de un fichero.

- Visto esto, ya podemos realizar la función mayoría:  

$$[(1100101001 \text{ AND } 1010111011) \text{ XOR } (1010111011 \text{ AND } 1011000011) \text{ XOR } (1011000011 \text{ AND } 1100101001)] = 1010101011$$

1. Elegimos los dos 'XOR' realizando los siguiente pasos.

- a) Introducir secuencia.
- b) Combinar secuencia.
- c) Operador XOR.
- d) Combinar secuencia.
- e) Operador XOR.

```
Selecciona una opción
  1 - Introducir Secuencia
  2 - Analizar Secuencia
  ! - Salir
Elija una opción: 1

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 3

Seleccione una opcion:
  1 - OperadorAND
  2 - OperadorOR
  3 - OperadorXOR
  4 - OperadorNAND
  5 - OperadorIMPLICA
  6 - OperadorNOT
Elija una opción: 3

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 3

Seleccione una opcion:
  1 - OperadorAND
  2 - OperadorOR
  3 - OperadorXOR
  4 - OperadorNAND
  5 - OperadorIMPLICA
  6 - OperadorNOT
Elija una opción: 3

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 
```

Figura 4.8: Elegir los dos 'XOR' de la función mayoría.

2. Ahora vamos a realizar la primera secuencia del segundo 'XOR' con el primer 'AND', que sería la primera parte de la operación y introducimos las dos secuencias para realizar la operación, sería esta (1100101001 AND 1010111011). Podemos ver la solución del mismo.

```

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 3

Seleccione una opción:
  1 - OperadorAND
  2 - OperadorOR
  3 - OperadorXOR
  4 - OperadorNAND
  5 - OperadorIMPLICA
  6 - OperadorNOT
Elija una opción: 1

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 2

Seleccione una opción:
  1 - Introduce la secuencia por teclado
  2 - Lee secuencia de un fichero
Elija una opción: 1
Introduce la secuencia (unos y ceros): 1100101001
Quieres guardar la secuencia en un archivo? (s, n):
n

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 2

Seleccione una opción:
  1 - Introduce la secuencia por teclado
  2 - Lee secuencia de un fichero
Elija una opción: 1
Introduce la secuencia (unos y ceros): 1010111011
Quieres guardar la secuencia en un archivo? (s, n):
n

La secuencia AND es: 1000101001
Quieres guardar la secuencia en un archivo? (s, n):
n

```

Figura 4.9: Realizar primer 'AND' de la función mayoría

- Después realizamos el segundo 'AND', que sería la segunda parte del segundo 'XOR', introducimos las dos secuencias para realizar la operación. Podemos ver la solución del 'AND' y del segundo 'XOR', que sería esta operación  $((1100101001 \text{ AND } 1010111011) \text{ XOR } (1010111011 \text{ AND } 1011000011))$ , el primer 'AND' es el que calculamos en el primer paso, el segundo 'AND' y el 'XOR' lo calculamos a continuación.



```
Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 3

Selecione una opción:
  1 - OperadorAND
  2 - OperadorOR
  3 - OperadorXOR
  4 - OperadorNAND
  5 - OperadorIMPLICA
  6 - OperadorNOT
Elija una opción: 1

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 2

Selecione una opción:
  1 - Introduce la secuencia por teclado
  2 - Lee secuencia de un fichero
Elija una opción: 1
Introduce la secuencia (unos y ceros): 1010111011
Quieres guardar la secuencia en un archivo? (s, n):
n

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 2

Selecione una opción:
  1 - Introduce la secuencia por teclado
  2 - Lee secuencia de un fichero
Elija una opción: 1
Introduce la secuencia (unos y ceros): 1011000011
Quieres guardar la secuencia en un archivo? (s, n):
n

La secuencia AND es: 1010000011
Quieres guardar la secuencia en un archivo? (s, n):
n

La secuencia XOR es: 0010101010
Quieres guardar la secuencia en un archivo? (s, n):
```

Figura 4.10: Realizar segundo 'AND' y por tanto segundo 'XOR' de la función mayoría

4. Por último, ya solo nos quedaría realizar el tercer 'AND' para tener la secuencia de la segunda parte del primer 'XOR', ya que la secuencia de la primera parte del 'XOR' es la que hemos realizado en el paso 2. Podemos ver la solución del 'AND', por tanto, ya tendríamos la solución del primer 'XOR' que sería la solución final de la función mayoría.

```

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 3

Seleccione una opción:
  1 - OperadorAND
  2 - OperadorOR
  3 - OperadorXOR
  4 - OperadorNAND
  5 - OperadorIMPLICA
  6 - OperadorNOT
Elija una opción: 1

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 2

Seleccione una opción:
  1 - Introduce la secuencia por teclado
  2 - Lee secuencia de un fichero
Elija una opción: 1
Introduce la secuencia (unos y ceros): 1011000011
Quieres guardar la secuencia en un archivo? (s, n):
n

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 2

Seleccione una opción:
  1 - Introduce la secuencia por teclado
  2 - Lee secuencia de un fichero
Elija una opción: 1
Introduce la secuencia (unos y ceros): 1100101001
Quieres guardar la secuencia en un archivo? (s, n):
n

La secuencia AND es: 1000000001
Quieres guardar la secuencia en un archivo? (s, n):
n

La secuencia XOR es: 1010101011
Quieres guardar la secuencia en un archivo? (s, n):

```

Figura 4.11: Solución final tras realizar el segundo 'AND' y por tanto primer 'XOR' de la función mayoría.

A continuación, vamos a ver la parte de analizar secuencias (*función `menuAnalizarSecuencia()`*), voy a mostrar ejemplos del Berlekamp-Massey, test de complejidad lineal (de una secuencia y de la combinación de dos secuencias) y por último contar rachas, ya que todos se hacen igual.

- El primero es el Berlekamp-Massey (*función `berlekampMassey()`*), como vemos en la siguiente imagen los pasos a seguir son:
  1. Analizar secuencia.
  2. Berlekamp-Massey.
  3. Elegimos la opción que queramos para generar la secuencia, ya sea con los LFSRs, introduciendo la secuencia por teclado o leerla

de un archivo o combinar varias secuencias. En la imagen vemos que generamos la secuencia con un LFSR. Por tanto, introducimos todos los datos para poder generarla y vemos que el resultado es el correcto ya que el polinomio  $x_7 + x_6 + x_3 + 1$  es reducible y se puede expresar como  $(x + 1)(x_6 + x_2 + x + 1)$ .

```

Selecciona una opción
  1 - Introducir Secuencia
  2 - Analizar Secuencia
  ! - Salir
Elija una opción: 2

Seleccione una opción:
  1 - Berlekamp-Massey
  2 - TestFrecuenciaMonobit
  3 - TestFrecuenciaBloque
  4 - TestRachas
  5 - TestComplejidadLineal
  6 - ContarRachas
Elija una opción: 1

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 1

Seleccione una opción:
  1 - LFSR
  2 - LFSR2
Elija una opción: 1
Introduce el grado del polinomio: 10
Introduce el valor del coeficiente (uno ó cero): 1
(1)x^10
(10,)
Introduce el valor del coeficiente (uno ó cero): 0
(0)x^9
Introduce el valor del coeficiente (uno ó cero): 0
(0)x^8
Introduce el valor del coeficiente (uno ó cero): 1
(1)x^7
(10, 7)
Introduce el valor del coeficiente (uno ó cero): 1
(1)x^6
(10, 7, 6)
Introduce el valor del coeficiente (uno ó cero): 0
(0)x^5
Introduce el valor del coeficiente (uno ó cero): 1
(1)x^4
(10, 7, 6, 4)
Introduce el valor del coeficiente (uno ó cero): 0
(0)x^3
Introduce el valor del coeficiente (uno ó cero): 1
(1)x^2
(10, 7, 6, 4, 2)
Introduce el valor del coeficiente (uno ó cero): 1
(1)x^1
(10, 7, 6, 4, 2, 1)
Introduce el valor del coeficiente (uno ó cero): 1
(1)x^0
(10, 7, 6, 4, 2, 1, 0)
Introduce el valor de la semilla (unos y ceros): 1001110101
Introduce el tamaño de la secuencia: 50000
Counter({'1': 25020, '0': 24980})
Quieres guardar la secuencia en un archivo? (s, n):

1 +
1 x^( 1 )+
1 x^( 2 )+
1 x^( 4 )+
1 x^( 6 )+
1 x^( 7 )+
1 x^( 10 )+

La longitud mínima es: 10

```

Figura 4.12: Berlakamp-Massey.

- El segundo es el test de frecuencia de complejidad lineal (*función linear\_complexity()*).
  1. Vamos a ver el test de frecuencia de complejidad lineal combinando dos secuencias con el operador 'XOR'. Los pasos seguir los vemos en la imagen

```

Seleccione una opcion:
  1 - Berlekamp-Massey
  2 - FrecuencyTest
  3 - FrecuencyTesttBlock
  4 - RunsTest
  5 - LinearComplexityTest
  6 - TestRachas
Elija una opción: 5

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 3

Seleccione una opcion:
  1 - OperadorAND
  2 - OperadorOR
  3 - OperadorXOR
  4 - OperadorNAND
  5 - OperadorIMPLICA
  6 - OperadorNOT
Elija una opción: 3

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 2

Seleccione una opción:
  1 - Introduce la secuencia por teclado
  2 - Lee secuencia de un fichero
Elija una opción: 2
Nombre del archivo donde está la secuencia:
secuencia5.txt

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 2

Seleccione una opción:
  1 - Introduce la secuencia por teclado
  2 - Lee secuencia de un fichero
Elija una opción: 2
Nombre del archivo donde está la secuencia:
secuencia6.txt
P-value = 0.531580983704
La secuencia es aceptada como aleatoria

```

Figura 4.13: Análisis test de frecuencia de complejidad lineal combinando dos secuencias.

- El último de los análisis que vamos a ver es contar las rachas de una secuencia (*función testRachas()*). Los pasos son similares a los anteriores por lo que no voy a entrar en detalle ya que lo único que hay

que hacer distinto es elegir la opción contar rachas como vemos en la imagen. Para generar la secuencia he elegido la opción de generar la secuencia con un LFSR.

```

Selecciona una opción
  1 - Introducir Secuencia
  2 - Analizar Secuencia
  ! - Salir
Elija una opción: 2

Seleccione una opción:
  1 - Berlekamp-Massey
  2 - TestFrecuenciaMonobit
  3 - TestFrecuenciaBloque
  4 - TestRachas
  5 - TestComplejidadLineal
  6 - ContarRachas
Elija una opción: 6

Selecciona una opción
  1 - Generar Secuencia
  2 - Introducir Secuencia
  3 - Combinar Secuencias
Elija una opción: 1

Seleccione una opción:
  1 - LFSR
  2 - LFSR2
Elija una opción: 1
Introduce el grado del polinomio: 12
Introduce el valor del coeficiente (uno ó cero): 1
(1)x^12
(12,)
Introduce el valor del coeficiente (uno ó cero): 0
(0)x^11
Introduce el valor del coeficiente (uno ó cero): 0
(0)x^10
Introduce el valor del coeficiente (uno ó cero): 0
(0)x^9
Introduce el valor del coeficiente (uno ó cero): 0
(0)x^8
Introduce el valor del coeficiente (uno ó cero): 0
(0)x^7
Introduce el valor del coeficiente (uno ó cero): 0
(0)x^6
Introduce el valor del coeficiente (uno ó cero): 0
(0)x^5
Introduce el valor del coeficiente (uno ó cero): 1
(1)x^4
(12, 4)
Introduce el valor del coeficiente (uno ó cero): 0
(0)x^3
Introduce el valor del coeficiente (uno ó cero): 1
(1)x^2
(12, 4, 2)
Introduce el valor del coeficiente (uno ó cero): 1
(1)x^1
(12, 4, 2, 1)
Introduce el valor del coeficiente (uno ó cero): 1
(1)x^0
(12, 4, 2, 1, 0)
Introduce el valor de la semilla (unos y ceros): 10111011011
Introduce el tamaño de la secuencia: 100000
Counter({'1': 50793, '0': 49207})
¿Quieres guardar la secuencia en un archivo? (s, n):
█
La secuencia es: 1011101101111011110000100001000110010110100110111100011111011
010011110000011000011010011010011001001000101101001011011011010100110000011
100000000100011010001011110001100111100100101010111110001010101011101101110
101100111100010001001101011101001100111100111000010001001011111100110000100101
0111000111010
El período es: 315
Rachas de 1: [(1, 40), (2, 20), (3, 8), (4, 7), (5, 3), (6, 1), (7, 1)]
Rachas de 0: [(1, 40), (2, 20), (3, 12), (4, 5), (5, 2), (9, 1)]

```

Figura 4.14: Contar rachas

- Por último, para salir del menú hay que elegir la opción salir, que es introduciendo el carácter '!'.

```
Selecciona una opción
    1 - Introducir Secuencia
    2 - Analizar Secuencia
    ! - Salir
Elija una opción: !
manuelgutierrezdelgado@ubuntu:~/Escritorio/TFG/CODIGO CORRECTOS$
```

Figura 4.15: Salir del menú

No hemos realizado todas las opciones de generar, analizar y combinar secuencias, ya que siempre hay que seguir los mismos pasos pero eligiendo la opción que se desee. Por tanto, sabiendo hacer lo anterior ya sabremos hacer todo sin problema.

# Capítulo 5

## Implementación

A continuación vamos a ver como hemos ido implementando los distintos algoritmos que constituyen el proyecto.

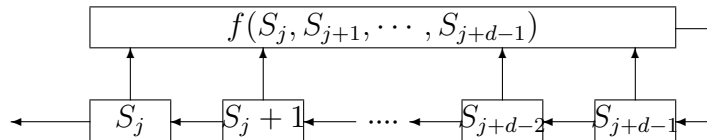
### 5.1. LFSR (Linear Feedback Shift Register)

LFSR (Linear Feedback Shift Register), que significa registro de desplazamiento con retroalimentación lineal [5].

Es un dispositivo constituido por  $d$  celdas, cada una de las cuales guarda un bit. En cada pulso del reloj, se desplaza el contenido de las celdas circularmente, pero la última se sustituye por una función lineal de estas celdas. A su vez, esta última celda se va añadiendo a la secuencia de salida.

Inicialmente, el contenido de las celdas es  $S_0, S_1, \dots, S_{d-1}$ , que es lo que se denomina semilla. Si en un instante el contenido es desde  $S_j, \dots, S_{j+d-1}$ , el contenido de la última celda se calcula a la vez que se desplaza como hemos dicho anteriormente mediante la función  $S_{j+d} = a_1 S_{j+d-1} + \dots + a_d S_j$ , donde los elementos  $a_i$  son 0 ó 1.

Vamos a centrarnos en los registros de desplazamiento retroalimentados. Como hemos dicho, están constituidos por  $d$  celdas  $S_j, S_{j+1}, \dots, S_{j+d-1}$ , y una función que realimenta la última celda, a la vez que las otras se desplazan, mientras que devuelve el bit que se encuentra en la primera celda. El esquema de funcionamiento podría ser:



Tras el pulso de reloj, el generador devuelve el valor almacenado en  $S_j$ , a la vez que desplaza a la izquierda los almacenados en  $S_{j+1} \cdots, S_{j+d-2}, S_{j+d-1}$ . En la última celda se almacena el valor resultado de aplicar la función  $f$  a los valores  $S_j, S_{j+1}, \cdots, S_{j+d-2}, S_{j+d-1}$ .

Asociado a un LFSR tenemos el polinomio  $p(x) = 1 + a_1x + \cdots + a_dx^d$ . Puesto que el número de posibles estados de un LFSR es finito ( $2^d$ ), ya que son  $d$  celdas, y si se repite un estado, la secuencia se repite a partir de ese estado, las secuencias generadas por un LFSR son periódicas. Al ser lineal, de darse el estado  $000 \cdots 0$ , a partir de ahí entonces todo sería  $000 \cdots 0$ , por tanto el primero de los posibles estados máximos es  $2^d - 1$ . Dependiendo del polinomio  $p(x)$ , la secuencia tendrá diferentes propiedades:

- Si  $p(x)$  es irreducible [1], el periodo es siempre el mismo (independientemente de la semilla) y es un divisor del periodo máximo.
- Si  $p(x)$  es irreducible y primitivo, el periodo es  $2^d - 1$ , independientemente de la semilla.
- Si  $p(x)$  es reducible, el periodo es menor que el periodo máximo y depende de la semilla.

Podemos ver algunos ejemplos del periodo según el polinomio que elijamos en el *capítulo 6: Prueba1*.

El periodo generado por un LFSR con polinomio primitivo, además de ser máximo, sustituye los 3 postulados de *Golomb*. Por tanto, va a pasar todos los test de aleatoriedad (a excepción del test de complejidad lineal). Podemos ver algunos ejemplos de esto en el *capítulo 6: Prueba2*.

Por este motivo, para evitar dicho problema sin perder las buenas características de aleatoriedad, vamos a realizar diferentes combinaciones para aumentar la complejidad lineal de las secuencias pseudoaleatorias generadas por LFSRs.

La complejidad lineal [4] [5] es el grado del menor polinomio que genera la secuencia. Por ejemplo, la secuencia 100001100001100001 se puede generar con un polinomio de grado 5 y con otro de grado 6, por tanto la complejidad lineal sería 5.

Vamos a ver que significa que un polinomio sea primitivo. Sea  $p(x)$  un polinomio con coeficientes en  $Z_2$ , ya que nosotros vamos a trabajar con números binarios, consideramos  $A = Z_2[x_p(x)]$ . Si  $p(x)$  es irreducible, este conjunto es



un cuerpo. Por ejemplo:  $Z_2[x]_{x^4+x+1}, Z_2[x]_{x^4+x^3+x^2+x+1}$ . Estos cuerpos tienen 16 elementos que se pueden representar:  $\{0, 1, x, x+1, \dots, x^3+x^2+x+1\}$ . Si eliminamos  $\{0\}$ , tenemos un grupo multiplicativo, por tanto, puede ocurrir que  $x$  sea un generador o no de este grupo. Que  $x$  sea un generador de este grupo quiere decir que si calculamos todas las potencias de  $x$  salen todos los elementos del cuerpo. Vamos a ver dos ejemplos:

1.  $Z_2[x]_{x^4+x+1}$ . Sabemos que  $x^4 + x + 1 = 0$ , es decir,  $x^4 = x + 1$ .

$$\begin{aligned}
 x^0 &= 1 \\
 x^1 &= x \\
 x^2 &= x^2 \\
 x^3 &= x^3 \\
 x^4 &= x + 1 \\
 x^5 &= x^2 + x \\
 x^6 &= x^3 + x^2 \\
 x^7 &= x^3 + x + 1 \\
 x^8 &= x^2 + 1 \\
 x^9 &= x^3 + x \\
 x^{10} &= x^2 + x + 1 \\
 x^{11} &= x^3 + x^2 + x \\
 x^{12} &= x^3 + x^2 + x + 1 \\
 x^{13} &= x^3 + x^2 + 1 \\
 x^{14} &= x^3 + 1 \\
 x^{15} &= 1
 \end{aligned}$$

Tenemos 15, todos menos el  $\{0\}$ .

2.  $Z_2[x]_{x^4+x^3+x^2+x+1}$ , de este sabemos que  $x^4 = x^3 + x^2 + x + 1$ .

$$\begin{aligned}
 x^0 &= 1 \\
 x^1 &= x \\
 x^2 &= x^2 \\
 x^3 &= x^3 \\
 x^4 &= x^3 + x^2 + x + 1 \\
 x^5 &= 1
 \end{aligned}$$

Tenemos 5, todos menos el  $\{0\}$ .

Por tanto, si hacemos un LFSR con el primer polinomio, la secuencia va a tener periodo 15 y si lo hacemos con el segundo polinomio la secuencia va a tener periodo 5.

## 5.2. Algoritmo de Berlekamp-Massey

Este algoritmo de Berlekamp-Massey dada una secuencia de bits calcula el polinomio del LFSR de tamaño mínimo que lo genera. El grado de este polinomio es lo que se llama la complejidad lineal.

El algoritmo lo he encontrado en [4] [5], y la implementación que yo he realizado del algoritmo es la siguiente.

```

1 def berlekampMassey(secuencia):
2     n = len(secuencia)
3     T = []
4
5     c = [1]
6     b = [1]
7     for i in range(0, n-1):
8         c.append(0)
9         b.append(0)
10        i+=1
11
12    L = 0
13    r = 0
14    m = -1
15
16    while r <= n-1:
17        d = 0
18
19        for i in range(0, r+1):
20            d = (d ^ (int(c[i]) and int(secuencia[r-i])))
21
22        if d == 1:
23            T = copy.copy(c)
24            for j in range(0, ((n-r) + (m-1))):
25                c[r-m+j] = c[r-m+j] ^ b[j]
26
27            if L <= r/2:
28                L = r + 1 - L
29                m = r
30                b = T
31
32            r = r + 1
33
34        j=0
35        for i in c:
36            if i == 1:
37                if j != 0:
38                    print(i,'x^(',j,')+')
39                else:
40                    print(i, '+')
41            j += 1
42        i += 1
43
44    return L

```

Por ejemplo, el algoritmo nos devuelve como resultado de la secuencia:  
1001101101010100010010011001111000111011101011110100101100101001110  
010001100010111000010000110100000111110110000001010110111111001101

```

1010101000100100110011110001110111010111101001011001010011100100011
0001011100001000011010000011111011000000101011011111110011011010101
0001001001100111100011101110101111010010110010100111001000110001011
1000010000110100000111110110000001010110111111100110110101010001001
0011001111000111011101011110100101100101001110010001100010111000010
0001101000001111101100000010101.

```

El polinomio que la genera,  $x_7 + x_6 + x_5 + x_2 + 1$  y la longitud mínima  $L = 7$ .

A continuación, vamos a describir los diferentes test que vamos a aplicar [6].

Una secuencia si es aleatoria se espera que se comporte de una determinada forma, por tanto, vamos a pasar una serie de test que tienen el estándar normal y el chi-cuadrado como distribución de referencia. En esta serie de test lo que vamos a estudiar es si las secuencias se pueden considerar aleatorias o no. Para esto tenemos una distribución de referencia que nos indica dependiendo de donde caiga el cálculo estadístico si las secuencias se pueden considerar aleatorias o no. Si el cálculo estadístico está dentro se la distribución de referencia se considera aleatoria y si está fuera no se considera aleatoria. El intervalo de confianza que tenemos es el 95 %, es decir, 0.05, si cae dentro del intervalo se considera aleatoria y si cae fuera no se considera aleatoria. El cálculo estadístico tiene que estar dentro del intervalo de confianza y el p-value nos indica la probabilidad de que dicho cálculo esté dentro o fuera, es decir, si se aproxima a 1 tiene mejor aleatoriedad que si se aproxima a 0

Por tanto, los test miden de una secuencia pseudoaleatoria cuanto tendría que valer el cálculo estadístico para considerarlo como aleatorio dependiendo del intervalo de confianza que tengamos y cuanto se aleja del mismo.

### 5.3. Test de Frecuencia (Monobit)

El propósito de este test es determinar si el número de unos y el número de ceros en una secuencia son aproximadamente los mismos que se esperaría para una secuencia verdaderamente aleatoria, es decir, el número de unos y ceros en una secuencia debe ser aproximadamente igual. Todos los test posteriores dependen de que este test sea aprobado.

Por ejemplo, una secuencia con 15449 unos y 15000 ceros la secuencia se considera aleatoria. Si tiene 15500 unos y 15000 ceros ya la secuencia no se considera aleatoria. También una secuencia con 3202 unos y 3000 ceros

la secuencia se considera aleatoria. Si tiene 3203 unos y 3000 ceros ya la secuencia no se considera aleatoria.

En estos ejemplos vemos el rango que tiene el test para decidir si considera la secuencia aleatoria o no.

La diferencia de este test con el primer postulado de *Golomb* es que no necesariamente tiene que ser la diferencia entre el número de unos y el número de ceros a lo sumo uno, sino que nos permite tener más margen dependiendo del tamaño de la secuencia.

## 5.4. Test de Frecuencia por Bloques

El enfoque de este test es la proporción de unos dentro de los bloques de  $M$  bits. El propósito es determinar si la frecuencia de unos en un bloque de  $M$  bits es aproximadamente  $M/2$ , como se esperaría bajo un supuesto de aleatoriedad.

## 5.5. Test de Rachas

El enfoque de este test es el número total de ejecuciones en la secuencia, donde una ejecución es una secuencia ininterrumpida de bits idénticos. Una ejecución de longitud  $k$  consta exactamente de  $k$  bits idénticos y se delimita antes y después con un bit del valor opuesto. El propósito es determinar si el número de ejecuciones de unos y ceros de varias longitudes es el esperado para una secuencia aleatoria. En particular, esta prueba determina si la oscilación entre tales ceros y unos es demasiado rápida o demasiado lenta.

El test de rachas calcula si la secuencia tiene periodo o no e indica el tamaño (el primer valor del conjunto) y la cantidad (el segundo valor del conjunto) de cada racha de unos y ceros de una secuencia. Consideramos que tiene periodo si se repite una vez como mínimo, en caso de que no se repita consideramos que la secuencia no tiene periodo. Por ejemplo:

Si introducimos la secuencia '1101011011001001000111000010111110010101110011010001001111000101000011000001000000111111101010100110011101101001011000110111101101011011001001000111000010111100101011101101000100111100010100001100000100000011111110101010011001110111010110001101111011011011001001000111000010111110010101110011' que tiene periodo, el test nos muestra:

El periodo es: 127

La secuencia periódica es: 110101101100100100011100001011  
 111001010111001101000100111100010100001100000100000011111  
 1101010100110011101110100101100011011110

Rachas de 1: [(1, 16), (2, 8), (3, 4), (4, 2), (5, 1), (7, 1)]

Rachas de 0: [(1, 16), (2, 8), (3, 4), (4, 2), (5, 1), (6, 1)]

Si introducimos la secuencia '10100010011110001010000110000010000001  
 11111101010100110011101110100101100011011110110101101100100100' que  
 no tiene periodo, el test nos muestra:

La secuencia no tiene periodo

La secuencia final es: 1010001001111000101000011000001000  
 000111111101010100110011101110100101100011011110110101101  
 100100100

Rachas de 1: [(1, 14), (2, 7), (3, 2), (4, 2), (7, 1)]

Rachas de 0: [(1, 13), (2, 7), (3, 3), (4, 1), (5, 1), (6, 1)]

Como estas secuencias están generadas con un polinomio irreducible y primitivo podemos ver que el número de rachas de un tamaño es la mitad del número de rachas del tamaño anterior.

La diferencia que puede ocurrir con algunas secuencias con el segundo postulado de *Golomb* es que no necesariamente el número de rachas de un tamaño tiene que ser la mitad del número de rachas del tamaño anterior, las rachas son aproximadas.

## 5.6. Test de Complejidad Lineal

El enfoque de este test es la longitud de un LFSR. El propósito es determinar si la secuencia es o no lo suficientemente compleja como para ser considerada aleatoria. Un LFSR que es demasiado corto implica que la secuencia no es aleatoria.



# Capítulo 6

## Pruebas

En el capítulo anterior se ha descrito como se ha realizado la implementación del proyecto, por tanto, ahora hay que probar que todo funciona perfectamente con algunos test y analizaremos los resultados.

Las pruebas van a consistir en generar secuencias, analizar dichas secuencias y combinarlas. Al combinar las secuencias veremos si cambia o no la aleatoriedad con cada test.

Vamos a comenzar con las distintas pruebas.

### 6.1. Prueba1

#### 6.1.1. Polinomio irreducible

Vamos a ver el ejemplo de dos secuencias con el mismo polinomio pero con distinta semilla, no debería de cambiar el periodo.

`secuenciaIrreducible1: Generada por un LFSR con los siguientes datos.`

`Polinomio: (4, 3, 2, 1, 0), que es irreducible no primitivo.  
Introduce el valor de la semilla (unos y ceros): 1011  
Introduce el tamaño de la secuencia: 100`

`La secuencia es: 10111101111011110111101111011110111101111011  
11011110111101111011110111101111011110111101111011110111`

`El periodo es: 5. Como podemos ver es un divisor del periodo máximo (15:5 = 3).`

Rachas de 1: [(1, 1), (3, 1)]

Rachas de 0: [(1, 1)]

secuenciaIrreducible2: Generada por un LFSR con los siguientes datos.

Polinomio: (4, 3, 2, 1, 0), que es irreducible no primitivo.

Introduce el valor de la semilla (unos y ceros): 1101

Introduce el tamaño de la secuencia: 100

La secuencia es: 1101111011110111101111011110111101111011110111  
10111101111011110111101111011110111101111011110111101111011

El periodo es: 5. Como podemos ver es un divisor del periodo máximo ( $15:5 = 3$ ).

Rachas de 1: [(2, 2)]

Rachas de 0: [(1, 1)]

Tal y como hemos dicho el periodo no cambia aunque la secuencia se realice con el mismo polinomio pero con distinta semilla.

### 6.1.2. Polinomio irreducible y primitivo

Vamos a ver que las secuencias generadas con polinomios irreducibles y primitivos tienen periodo máximo.

secuenciaIrreduciblePrimitiva: Generada por un LFSR con los siguientes datos.

Polinomio: (6, 4, 3, 2, 1, 0), que es irreducible y primitivo.

Introduce el valor de la semilla (unos y ceros): 100111

Introduce el tamaño de la secuencia: 150

La secuencia es: 10011110000011011100110001110101111101101000  
1000010110010101001001111000001101110011000111010111110110100  
0100001011001010100100111100000110111001100

El periodo es: 63. Como vemos es el periodo máximo.



### 6.1.3. Polinomio reducible

Vamos a ver el ejemplo de dos secuencias con el mismo polinomio pero con distinta semilla, en este caso debería de cambiar el periodo.

secuenciaReducible1: Generada por un LFSR con los siguientes datos.

Polinomio: (5, 2, 1, 0), que es reducible.

Introduce el valor de la semilla (unos y ceros): 10111

Introduce el tamaño de la secuencia: 100

La secuencia es: 1011110010000110111100100001101111001000011011  
110010000110111100100001101111001000011011110010000110

El periodo es: 14. Como vemos el periodo es menor que el periodo máximo.

Rachas de 1: [(1, 3), (4, 1)]

Rachas de 0: [(1, 1), (2, 1), (4, 1)]

secuenciaReducible2: Generada por un LFSR con los siguientes datos.

Polinomio: (5, 2, 1, 0), que es reducible.

Introduce el valor de la semilla (unos y ceros): 10001

Introduce el tamaño de la secuencia: 100

La secuencia es: 100010110001011000101100010110001011000101100010  
1100010110001011000101100010110001011000101100010110

El periodo es: 7. Como vemos es el periodo es menor que el periodo máximo.

Rachas de 1: [(1, 3)]

Rachas de 0: [(1, 1), (3, 1)]

Podemos ver que generando las secuencia con el mismo polinomio reducible y distinta semilla el periodo es distinto.

## 6.2. Prueba2

Vamos a ver que al generar una secuencia con un polinomio irreducible y primitivo, la secuencia pasa todos los test menos el de complejidad lineal.

secuencia1: Está generada por un LFSR con los siguientes datos.

Polinomio: (10, 9, 7, 5, 4, 2, 0), que es irreducible y primitivo.

Introduce el valor de la semilla (unos y ceros): 1011011001

Introduce el tamaño de la secuencia: 10000

2 - Test de frecuencia (monobit).

P-value = 0.729034489341

La secuencia es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.946516392226

La secuencia es aceptada como aleatoria.

4 - Test rachas.

P-value = 0.912818675493

La secuencia es aceptada como aleatoria.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia es aceptada como aleatoria.

6 - Contar rachas.

El periodo es: 1023

Rachas de 1: [(1, 128), (2, 64), (3, 32), (4, 16), (5, 8), (6, 4), (7, 2), (8, 1), (10, 1)]

Rachas de 0: [(1, 128), (2, 64), (3, 32), (4, 16), (5, 8), (6, 4), (7, 2), (8, 1), (9, 1)]

Sabiendo que una secuencia generada por un polinomio reducible no es buena del todo, vamos a hacer pruebas con polinomios irreducibles e irreducibles primitivos para analizar la aleatoriedad.

### 6.3. Prueba3

secuencia3: Está generada por un LFSR con los siguientes datos.

Polinomio: (11, 8, 6, 5, 4, 1, 0), que es irreducible y primitivo.

Introduce el valor de la semilla (unos y ceros): 10111101001

Introduce el tamaño de la secuencia: 850500

Número de unos y ceros: {'1': 425466, '0': 425034}

2 - Test de frecuencia (monobit).

P-value = 0.639475663388

La secuencia es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 1.0

La secuencia es aceptada como aleatoria.

4 - Test rachas.

P-value = 0.672201820148

La secuencia es aceptada como aleatoria.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

6 - Contar rachas.

El periodo es: 2047

Rachas de 1: [(1, 256), (2, 128), (3, 64), (4, 32), (5, 16), (6, 8), (7, 4), (8, 2), (9, 1), (11, 1)]

Rachas de 0: [(1, 256), (2, 128), (3, 64), (4, 32), (5, 16), (6, 8), (7, 4), (8, 2), (9, 1), (10, 1)]

Podemos apreciar que esta secuencia tiene muy buena aleatoriedad en casi todos los test, menos en el test de complejidad lineal, esto quiere decir que la secuencia es predecible, por tanto, tendremos que aumentar la complejidad lineal para que la secuencia pueda considerarse aleatoria. También podemos observar que la secuencia tiene periodo máximo y el número de rachas de un tamaño va siendo la mitad del tamaño anterior, por tanto, esta secuencia satisface el segundo postulado de Golomb.

secuencia4: Está generada por un LFSR con los siguientes datos.

Polinomio: (15, 14, 13, 6, 5, 4, 3, 1, 0), que es irreducible y primitivo.

Introduce el valor de la semilla (unos y ceros):110100010000101

Introduce el tamaño de la secuencia: 850500

Número de unos y ceros: {'1': 425266, '0': 425234}

2 - Test de frecuencia (monobit).

P-value = 0.972320023129

La secuencia es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.976379716581

La secuencia es aceptada como aleatoria.

4 - Test rachas.

P-value = 0.993079745301

La secuencia es aceptada como aleatoria.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

6 - Contar rachas.

El periodo es: 32767

Rachas de 1: [(1, 4097), (2, 2049), (3, 1023), (4, 512),  
(5, 256), (6, 128), (7, 64), (8, 32), (9, 16), (10, 8),  
(11, 4), (12, 2), (13, 1), (15, 1)]

Rachas de 0: [(1, 4096), (2, 2048), (3, 1024), (4, 512),  
(5, 256), (6, 128), (7, 64), (8, 32), (9, 16), (10, 8),  
(11, 4), (12, 2), (13, 1), (14, 1)]

En la 'secuencia4' vemos que la aleatoriedad es mejor que en la 'secuencia3', ya que el p-value se aproxima a 1 en casi todos los test. En el test de complejidad lineal pasa lo mismo que en las anteriores. También vemos que esta secuencia tiene periodo máximo y cumple el segundo postulado de Golomb.

A continuación vamos a combinar las dos secuencias anteriores, la 'secuencia3' y la 'secuencia4' para ver cómo cambia la aleatoriedad.

AND:

2 - Test de frecuencia (monobit).

P-value =  $-1.10866944695e-08$

La secuencia no es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

4 - Test rachas.

No se puede realizar el test.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

OR:

2 - Test de frecuencia (monobit).

P-value =  $-1.10989772825e-08$

La secuencia no es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

4 - Test rachas.

No se puede realizar el test.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

Las secuencias al combinarlas con 'AND' y 'OR' no se consideran aleatorias, esto ocurre porque al hacer la operación suelen salir bastantes mas unos que ceros o bastantes mas ceros que unos.

XOR:

2 - Test de frecuencia (monobit).

P-value = 0.501401184167

La secuencia es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.783648606092

La secuencia es aceptada como aleatoria.

4 - Test rachas.

P-value = 0.174439492972

La secuencia es aceptada como aleatoria.

5 - Test de complejidad lineal.

P-value = 0.570485905552

La secuencia es aceptada como aleatoria.

En el 'XOR' podemos observar que la secuencia sigue cumpliendo los test, pero lo mas importante es que el test de complejidad lineal a aumentado, por lo que la secuencia puede considerarse aleatoria y por tanto mucho menos predecible.

NAND:

2 - Test de frecuencia (monobit).

P-value = -1.10958006929e-08

La secuencia no es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

4 - Test rachas.

No se puede realizar el test.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

NOT (secuencia3):

2 - Test de frecuencia (monobit).

P-value = 0.639475663388

La secuencia es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 1.0

La secuencia es aceptada como aleatoria.

4 - Test rachas.

P-value = 0.993079745301

La secuencia es aceptada como aleatoria

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia es aceptada como aleatoria.

6 - Contar rachas.

El periodo es: 2047

Rachas de 1: [(1, 256), (2, 128), (3, 64), (4, 32), (5, 16), (6, 8), (7, 4), (8, 2), (9, 1), (10, 1)]

Rachas de 0: [(1, 256), (2, 128), (3, 64), (4, 32), (5, 16), (6, 8), (7, 4), (8, 2), (9, 1), (11, 1)]

En el operador 'NAND' pasa lo mismo que en el 'AND' y en el 'OR'. Y en el operador 'NOT' de la 'secuencia3' vemos que la aleatoriedad es la misma que sin aplicarle el operador.

## 6.4. Prueba4

secuencia5: Está generada por un LFSR con los siguientes datos.

Polinomio: (11, 7, 6, 1, 0), que es irreducible.

Introduce el valor de la semilla (unos y ceros):

11100100001

Introduce el tamaño de la secuencia: 45000

Número de unos y ceros: {'1': 24267, '0': 20733}

2 - Test de frecuencia (monobit).

P-value = -2.19771782037e-09

La secuencia no es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 1.14174318948e-54

La secuencia no es aceptada como aleatoria.

4 - Test rachas.

P-value = 0

La secuencia no es aceptada como aleatoria.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

6 - Contar rachas.

El periodo es: 89

Rachas de 1: [(3, 7), (1, 6), (2, 6), (4, 1), (5, 1)]

Rachas de 0: [(1, 10), (2, 5), (3, 2), (4, 1), (5, 1), (6, 1)]

En esta prueba tenemos una secuencia generada por un polinomio irreducible pero no primitivo. Podemos ver que la aleatoriedad no es buena, es decir, la secuencia no se puede considerar aleatoria y que el periodo es un divisor del periodo máximo.

secuencia6: Está generada por un LFSR con los siguientes datos.

Polinomio: (11, 10, 9, 6, 5, 4, 3, 2, 0), que es irreducible y primitivo.

Introduce el valor de la semilla (unos y ceros):

11100100001

Introduce el tamaño de la secuencia: 45000

Número de unos y ceros: {'1': 22515, '0': 22485}

2 - Test de frecuencia (monobit).

P-value = 0.887537083784

La secuencia es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.999518139291

La secuencia es aceptada como aleatoria.

4 - Test rachas.

P-value = 0.894914999444

La secuencia es aceptada como aleatoria.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia no es aceptada como aleatoria.



6 - Contar rachas.

El periodo es: 2047

Rachas de 1: [(1, 256), (2, 128), (3, 64), (4, 32), (5, 16), (6, 8), (7, 4), (8, 2), (9, 1), (11, 1)]

Rachas de 0: [(1, 256), (2, 128), (3, 64), (4, 32), (5, 16), (6, 8), (7, 4), (8, 2), (9, 1), (10, 1)]

En este caso la 'secuencia6' la hemos generado con la misma semilla y el mismo grado del polinomio, pero la diferencia es que el polinomio ahora es irreducible y primitivo. Entonces podemos ver que con un polinomio irreducible y primitivo la aleatoriedad aumenta bastante, menos en el test de complejidad lineal que sigue siendo no aleatoria y la secuencia tiene periodo máximo.

A continuación vamos a combinar las dos secuencias anteriores, la 'secuencia5' y la 'secuencia6' para ver cómo cambia la aleatoriedad y comentaré los cambios al final del ejemplo.

AND:

2 - Test de frecuencia (monobit).

P-value = -8.93362902024e-09

La secuencia no es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

4 - Test rachas.

No se puede realizar el test.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

OR:

2 - Test de frecuencia (monobit).

P-value = -1.06072756681e-08

La secuencia no es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

4 - Test rachas.

No se puede realizar el test.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

XOR:

2 - Test de frecuencia (monobit).

P-value = 0.0011816901243

La secuencia no es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

4 - Test rachas.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

5 - Test de complejidad lineal.

P-value = 1.24686877231e-19

La secuencia no es aceptada como aleatoria.

NAND:

2 - Test de frecuencia (monobit).

P-value = -8.93362902024e-09

La secuencia no es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

4 - Test rachas.

No se puede realizar el test.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

NOT (secuencia6):

2 - Test de frecuencia (monobit).

P-value = 0.887537083784

La secuencia es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.999518139291

La secuencia es aceptada como aleatoria.

4 - Test rachas.

P-value = 0.894914999444

La secuencia es aceptada como aleatoria.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

6 - Contar rachas.

El periodo es: 2047

Rachas de 1: [(1, 256), (2, 128), (3, 64), (4, 32), (5, 16), (6, 8), (7, 4), (8, 2), (9, 1), (10, 1)]

Rachas de 0: [(1, 256), (2, 128), (3, 64), (4, 32), (5, 16), (6, 8), (7, 4), (8, 2), (9, 1), (11, 1)]

Podemos ver que en el 'AND', el 'OR' y el 'NAND' la secuencia no se considera aleatoria por los motivos anteriores. Sin embargo, vemos que en el 'XOR' que la secuencia no puede considerarse aleatoria, esto es porque la 'secuencia5' tiene muy malas características de aleatoriedad y por tanto al combinarla con la 'secuencia6' mejora su aleatoriedad, pero aunque haya sido mejora la secuencia no puede considerarse aleatoria.

## 6.5. Prueba5

secuencia7: Está generada por un LFSR con los siguientes datos.

Polinomio: (13, 12, 11, 9, 8, 6, 5, 1, 0), que es irreducible y primitivo.

Introduce el valor de la semilla (unos y ceros):

1011000011101

Introduce el tamaño de la secuencia: 30000

Número de unos y ceros: {'1': 15036, '0': 14964}

2 - Test de frecuencia (monobit).

P-value = 0.677635253935

La secuencia es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.99516448632

La secuencia es aceptada como aleatoria.

4 - Test rachas.

P-value = 0.661537748587

La secuencia es aceptada como aleatoria.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

6 - Contar rachas.

El periodo es: 8191

Rachas de 1: [(1, 1025), (2, 512), (3, 257), (4, 127), (5, 64), (6, 32), (7, 16), (8, 8), (9, 4), (10, 2), (11, 1), (13, 1)]

Rachas de 0: [(1, 1024), (2, 512), (3, 256), (4, 128), (5, 64), (6, 32), (7, 16), (8, 8), (9, 4), (10, 2), (11, 1), (12, 1)]

secuencia8: Está generada por un LFSR con los siguientes datos.

Polinomio: (9, 7, 6, 4, 0), que es irreducible y primitivo.

Introduce el valor de la semilla (unos y ceros):

100000001

Introduce el tamaño de la secuencia: 30000

Número de unos y ceros: {'1': 15021, '0': 14979}

2 - Test de frecuencia (monobit).

P-value = 0.808402741034

La secuencia es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.98136743416

La secuencia es aceptada como aleatoria.

4 - Test rachas.

P-value = 0.694364952946

La secuencia es aceptada como aleatoria.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

6 - Contar rachas.

El periodo es: 511

Rachas de 1: [(1, 65), (2, 33), (3, 15), (4, 8), (5, 4),  
(6, 2), (7, 1), (9, 1)]

Rachas de 0: [(1, 64), (2, 32), (3, 16), (4, 8), (5, 4),  
(6, 2), (7, 1), (8, 1)]

Antes de combinarlas, podemos ver que las dos secuencias están generadas con un polinomio irreducible y primitivo lo que indica que dichas secuencias satisfacen todos los test de aleatoriedad, exceptuando el test de complejidad lineal. Estas secuencias las hemos generado con un tamaño más pequeño para ver si cambia su aleatoriedad a mejor o peor al combinarlas.

A continuación vamos a combinar las dos secuencias anteriores, la 'secuencia7' y la 'secuencia8' para ver cómo cambia la aleatoriedad.

AND:

2 - Test de frecuencia (monobit).

P-value = -7.98505392535e-09

La secuencia no es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

4 - Test rachas.

No se puede realizar el test.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

OR:

2 - Test de frecuencia (monobit).

P-value = -8.0506196035e-09

La secuencia no es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

4 - Test rachas.

No se puede realizar el test.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

XOR:

2 - Test de frecuencia (monobit).

P-value = 0.619526711396

La secuencia es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.716601682076

La secuencia es aceptada como aleatoria.

4 - Test rachas.

P-value = 0.041110544017

La secuencia no es aceptada como aleatoria.

5 - Test de complejidad lineal.

P-value = 4.84922199218e-16

La secuencia no es aceptada como aleatoria.

NAND:

2 - Test de frecuencia (monobit).

P-value = -7.98505392535e-09

La secuencia no es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

4 - Test rachas.

No se puede realizar el test.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

NOT (secuencia7):

2 - Test de frecuencia (monobit).

P-value = 0.677635253935

La secuencia es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.99516448632

La secuencia es aceptada como aleatoria.

4 - Test rachas.

P-value = 0.661537748587

La secuencia es aceptada como aleatoria.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

6 - Contar rachas.

El periodo es: 8191

Rachas de 1: [(1, 1024), (2, 512), (3, 256), (4, 128), (5, 64), (6, 32), (7, 16), (8, 8), (9, 4), (10, 2), (11, 1), (12, 1)]

Rachas de 0: [(1, 1025), (2, 512), (3, 257), (4, 127), (5, 64), (6, 32), (7, 16), (8, 8), (9, 4), (10, 2), (11, 1), (13, 1)]

Si nos fijamos bien en el operador 'OR' podemos ver que al combinar las dos secuencias en casi todos los test la aleatoriedad es menor, menos en el de complejidad lineal que la aleatoriedad ha aumentado un poco. Por tanto, la secuencia cumple los test de aleatoriedad pero es predecible ya que no cumple el de complejidad lineal. Esto ocurre porque el tamaño de las secuencias es menor.

## 6.6. Prueba6

Ahora vamos a hacer los test a la función mayoría para ver que sucede al combinar más de dos secuencias con distintos operadores. La función mayoría se realiza de la siguiente forma:

```
{(x AND y) XOR (y AND z) XOR (z AND x)}
```

Está formada por tres 'AND' y dos 'XOR'. El menú implementado nos permite realizar estas combinaciones.

```
Polinomio: (13, 12, 11, 1, 0), que es irreducible y primitivo.
Introduce el valor de la semilla (unos y ceros): 1101000011011
Introduce el tamaño de la secuencia: 900000
Número de unos y ceros: {'1': 450059, '0': 449941}
```

```
secuencia9:
```

```
2 - Test de frecuencia (monobit).
P-value = 0.901012094963
La secuencia es aceptada como aleatoria.
```

```
3 - Test de frecuencia por bloques.
P-value = 1.0
La secuencia es aceptada como aleatoria.
```

```
4 - Test rachas.
P-value = 0.887660825312
La secuencia es aceptada como aleatoria.
```

```
5 - Test de complejidad lineal.
P-value = 0.999995158064
La secuencia es aceptada como aleatoria.
```

```
6 - Contar rachas.
El periodo es: 8191
Rachas de 1: [(1, 1024), (2, 512), (3, 256), (4, 128), (5, 64),
(6, 32), (7, 16), (8, 8), (9, 4), (10, 2), (11, 1), (13, 1)]
Rachas de 0: [(1, 1024), (2, 512), (3, 256), (4, 128), (5, 64),
(6, 32), (7, 16), (8, 8), (9, 4), (10, 2), (11, 1), (12, 1)]
```



Polinomio: (15, 1, 0), irreducible y primitivo.  
Introduce el valor de la semilla (unos y ceros): 110100010000101  
Introduce el tamaño de la secuencia: 900000  
Número de unos y ceros: {'0': 450039, '1': 449961}

secuencia10:

2 - Test de frecuencia (monobit).  
P-value = 0.934472390306  
La secuencia es aceptada como aleatoria.

3 - Test de frecuencia por bloques.  
P-value = 0.928385013624  
La secuencia es aceptada como aleatoria.

4 - Test rachas.  
P-value = 0.931125820296  
La secuencia es aceptada como aleatoria.

5 - Test de complejidad lineal.  
P-value = 0.0  
La secuencia no es aceptada como aleatoria.

6 - Contar rachas.  
El periodo es: 32767  
Rachas de 1: [(1, 4096), (2, 2049), (3, 1025), (4, 512),  
(5, 255),  
(6, 128), (7, 64), (8, 32), (9, 16), (10, 8), (11, 4), (12, 2),  
(13, 1), (15, 1)]  
Rachas de 0: [(1, 4096), (2, 2048), (3, 1024), (4, 512),  
(5, 256),  
(6, 128), (7, 64), (8, 32), (9, 16), (10, 8), (11, 4), (12, 2),  
(13, 1), (14, 1)]

Polinomio: (10, 3, 0), que es irreducible y primitivo.  
Introduce el valor de la semilla (unos y ceros): 1101100101  
Introduce el tamaño de la secuencia: 900000  
Número de unos y ceros: {'1': 450447, '0': 449553}

secuencia11:

2 - Test de frecuencia (monobit).

P-value = 0.346009002313

La secuencia es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 1.0

La secuencia es aceptada como aleatoria.

4 - Test rachas.

P-value = 0.334856637516

La secuencia es aceptada como aleatoria.

5 - Test de complejidad lineal.

P-value = 0.0

La secuencia no es aceptada como aleatoria.

6 - Contar rachas.

El periodo es: 1023

Rachas de 1: [(1, 129), (2, 65), (3, 31), (4, 16), (5, 8),  
(6, 4), (7, 2), (8, 1), (10, 1)]

Rachas de 0: [(1, 128), (2, 64), (3, 32), (4, 16), (5, 8),  
(6, 4), (7, 2), (8, 1), (9, 1)]

Función mayoría: (secuencia9 *AND* secuencia10) *XOR* (secuencia10 *AND* secuencia11) *XOR* (secuencia11 *AND* secuencia9):

2 - Test de frecuencia (monobit).

P-value = 0.346009002313

La secuencia es aceptada como aleatoria.

3 - Test de frecuencia por bloques.

P-value = 0.999991567258

La secuencia es aceptada como aleatoria.

4 - Test rachas.

P-value = 0.316186656304

La secuencia es aceptada como aleatoria

5 - Test de complejidad lineal.

P-value = 0.600258139808

La secuencia es aceptada como aleatoria.

6 - Contar rachas.

Rachas de 1: [(1, 112264), (2, 56506), (3, 28461), (4, 14098), (5, 6950), (6, 3490), (7, 1740), (8, 869), (9, 381), (10, 275), (11, 118), (12, 45), (13, 19), (14, 8), (15, 10), (16, 3)]

Rachas de 0: [(1, 112656), (2, 56394), (3, 28198), (4, 14115), (5, 6970), (6, 3490), (7, 1691), (8, 850), (9, 516), (10, 161), (11, 85), (12, 62), (13, 23), (14, 11), (15, 6), (16, 4), (17, 2), (18, 3)]

La función mayoría, como vemos cumple todos los test de aleatoriedad a pesar de realizarse con tres 'AND' (que como hemos visto anteriormente al hacer la combinación con el operador 'AND' las secuencias no podían considerarse aleatorias), esto sucede porque combinamos las secuencias realizadas con el operador 'XOR'. Podemos ver que aumenta la aleatoriedad el test de complejidad lineal y todos los demás siguen siendo aleatorios.



# Capítulo 7

## Conclusiones

Después de realizar este trabajo la principal conclusión a la que he llegado es que una secuencia pseudoaleatoria puede tener muy buenas características de aleatoriedad pero a su vez ser predecible, es decir, que no debemos creer que porque tenga buenas características de aleatoriedad es imposible que alguien pueda romperla, porque como hemos visto en el capítulo 2, una secuencia que tenga la complejidad lineal baja se puede romper y sacar toda la secuencia cogiendo una pequeña muestra de la secuencia.

Nosotros podemos aumentar la complejidad lineal con un polinomio de mayor grado, pero esto requiere de mucha memoria porque tiene que tener muchas celdas a la vez, entonces, esto sería muy costoso y la solución pasa por generar LFSR pequeños que tengan buenas propiedades de aleatoriedad, a excepción de la complejidad lineal y combinarlos de forma apropiada para mantener la aleatoriedad que ya teníamos y aumentar la complejidad lineal que es el gran problema de las secuencias pseudoaleatorias para sus aplicaciones criptográficas. Aquí hemos visto un ejemplo de una función llamada mayoría que la mejora y la combinación con el operador 'XOR' también la mejora.

Otra conclusión que he sacado es que al hacer el test de complejidad lineal a una secuencia que le hemos realizado el .º exclusivo.<sup>el</sup> tamaño de bloque es importante, ya que por ejemplo, si el periodo es pequeño y el tamaño de bloque es grande siempre saldría la misma complejidad lineal en cada bloque, por tanto, la secuencias analizada con el test de complejidad lineal no se consideraría aleatoria.

En cuanto al lenguaje de programación elegido (Python), he podido utilizar varias librerías (numpy, scipy, etc) para el cálculo de los test estadísticos que me han ayudado, también es un lenguaje bueno para operar bit a bit con secuencias 'str' y para manipular dichas secuencias.

Este trabajo se podría continuar analizando otro tipo de combinaciones distintas a las que ya hay para mejorar la complejidad lineal y mantener la aleatoriedad del resto de los test, ya que el tema de las secuencias pseudo-aleatorias está presente en la actualidad.

# Bibliografía

- [1] POLINOMIOS IRREDUCIBLES <https://www.wolframalpha.com/>
- [2] MANUAL DE LATEX <https://www.ugr.es/~mmartins/material/latex-basico.pdf>
- [3] GOLOMB, S. W. *Shift Register Sequences*. San Francisco, Holden Day 1967.
- [4] Peralta F. I., Duchén G. I. y Vázquez R. *Información Tecnológica*. Vol. 17 N°3-2006, pág.: 167-178
- [5] Pastor Franco, J., Sarasa López, M. A., Salazar Riaño, J. L. *CRIPTOGRAFÍA DIGITAL, fundamentos y aplicaciones*, Zaragoza, España: Prensas Universitarias de Zaragoza (2001).
- [6] Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., Vo, S. (2010). *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*.





