Penerapan Relasi Rekurens dalam Generator Bilangan Pseudo-Acak berbasis Linear Feedback Shift Register (LFSR)

Muhamad Nazih Najmudin — 13523144[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13523144@std.stei.itb.ac.id, [2]nazihnajmudin@gmail.com

Abstrak—Perkembangan teknologi komputasi yang pesat memungkinkan simulasi dan pemodelan masalah dunia nyata dengan lebih efisien, salah satunya melalui penggunaan bilangan pseudo-acak. Bilangan pseudo-acak banyak digunakan dalam berbagai bidang, termasuk kriptografi, simulasi Monte Carlo, dan pengujian perangkat lunak. Salah satu metode untuk menghasilkan bilangan pseudo-acak adalah *Linear Feedback Shift Register* (LFSR), yang menggunakan relasi rekurens untuk menghasilkan urutan bit acak. Penelitian ini membahas penerapan relasi rekurens dalam LFSR dan implikasinya dalam menghasilkan bilangan pseudo-acak. Hasil implementasi dan pengujian menunjukkan bahwa LFSR berhasil menghasilkan urutan angka biner yang lolos uji *Autocorrelation Test*, namun gagal pada uji *Runs Test* dan *Chi-Square Test*. Hal ini mengindikasikan bahwa bilangan yang dihasilkan bersifat deterministik dan tidak sepenuhnya acak. Meskipun demikian, LFSR tetap memiliki potensi yang signifikan untuk aplikasi yang membutuhkan keacakan, seperti simulasi dan sampling. Penelitian ini juga menunjukkan bahwa meskipun LFSR tidak ideal untuk kriptografi dengan tingkat keamanan tinggi, metode ini dapat diterapkan untuk tujuan lain yang memerlukan keacakan terbatas. Penerapan relasi rekurens dalam LFSR terbukti memiliki manfaat dalam berbagai aplikasi komputasi yang memanfaatkan bilangan pseudo-acak.

Kata kunci—Bilangan pseudo-acak, *Linear Feedback Shift Register* (LFSR), relasi rekurens.

I. PENDAHULUAN

Di era modern ini, teknologi berkembang dengan sangat cepat. Saat ini, teknologi komputasi telah melalui banyak perkembangan. Banyak hal-hal yang dahulu dikerjakan oleh manusia, kini tergantikan dengan komputer. Banyak pula masalah-masalah dalam sains dan matematika yang dahulu tidak bisa diselesaikan oleh manusia, kini dapat dibuat pemodelan masalah dengan sangat mudah oleh komputer. Kemampuan komputer yang meningkat cepat ini dipicu oleh berkembangnya teknologi, algoritma-algoritma komputasi, dan metode pemodelan yang menghasilkan cara efisien untuk dijalankan dan dipahami oleh komputer.

Sebagai alat untuk mempermudah kehidupan manusia, komputer banyak digunakan untuk memodelkan atau mensimulasikan dunia nyata. Akan tetapi, tidak seperti manusia, komputer sangat terpaku pada aturan-aturan logis dan algoritmik yang baku, sehingga komputer memiliki kendala dalam mensimulasikan dunia nyata yang memiliki keacakan tinggi. Maka dari itu, bilangan pseudo-acak dapat menjadi solusi untuk mensimulasikan keacakan dunia nyata, atau memperoleh nilai acak selayaknya dunia nyata untuk digunakan dalam komputasi-komputasi tertentu yang memerlukannya.

Bilangan pseudo-acak digunakan dalam simulasi Monte Carlo berbasis probabilitas dalam bidang fisika, biologi dan teknik, contohnya dalam pemodelan perilaku partikel dalam fisika dan perkiraan harga opsi dalam keuangan. Bilangan pseudo-acak juga dimanfaatkan dalam bidang komputasi lainnya, seperti dalam kriptografi untuk kunci enkripsi, protokol kemanan jaringan untuk nonce atau port, grafika komputer untuk noise generation atau ray tracing, machine learning dan AI untuk memagi dataset menjadi training dan testing secara acak, hingga game development untuk menciptakan elemen acak dalam game seperti lokasi musuh dan drop item. Bilangan pseudo-acak juga sangat berguna untuk dalam pengujian perangkat lunak untuk membuat input uji dan membantu dalam mengidentifikasi bug sistem dengan berbagai skenario. Selain itu, bilangan pseudo-acak juga digunakan untuk pengambilan sampel acak dalam survei atau eksperimen statistik.

Ada banyak cara untuk memperoleh bilangan pseudo-acak. Salsah satunya, adalah dengan metode *Linear Feedback Shift* Register (LFSR). Metode ini merupakan dalah satu metode yang mudah, namun memiliki keacakan yang cukup tinggi untuk digunakan dalam komputasi. Untuk memperoleh bilangan pseudo-acak dengan LFSR, terdapat proses rekursif yang harus dilakukan. Maka dari itu, penelitian ini akan membahas mengenai penerapan rekursifitas tersebut dalam metode LFSR untuk menghasilkan bilangan pseudo-acak.

II. DASAR TEORI

A. Relasi Rekurens

Relasi rekurens adalah sebuah persamaan yang menyatakan deret ke-n, yaitu elemen a_n pada deretan atau *sequence* $\{a_n\}$, yang dinyatakan secara rekursif dalam satu atau lebih term elemen sebelumnya, yakni elemen a_{n-i} dengan $i < n$. Suatu barisan deret dapat dinyatakan sebagai informasi satu atau lebih nilai dan relasi rekurens yang diperlukan untuk menghitung suku-suku selanjtunya dalam barisan deret. Contoh relasi rkeurens adalah sebagai berikut:

$$a_n = 2a_{n-1} + 1$$
$$a_n = a_{n-1} + 2a_{n-2}$$
$$a_n = 2a_{n-1} - a_{n-2}$$

Solusi dari sebuah relasi rekurens adalah persamaan yang menyatakan deret ke-n, yaitu elemen a_n, dengan tidak lagi mengandung terminologi rekursif, tetapi memenuhi relasi rekurensnya. Secara iteratif, solusi relasi rekurens dapat dicari dengan menemukan pola deret melalui substitusi definisi persamaan itu sendiri pada pemanggilan term rekursifnya, seperti ditunjukkan pada gambar brikut:

$a_n = 2a_{n-1} - a_{n-2}$; $a_0 = 0$ dan $a_1 = 3$?

Periksa apakah $a_n = 3n$ merupakan solusi relasi rekurens tersebut.

Penyelesaian: $2a_{n-1} - a_{n-2} = 2[3(n-1)] - 3(n-2)$
$= 6n - 6 - 3n + 6$
$= 3n = a_n$

Jadi, $a_n = 3n$ merupakan solusi dari relasi rekurens tersebut.

Gambar 1. Contoh penyelesaian relasi rekurens secara iteratif.
Sumber: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/11-Deretan,%20rekursi-dan-relasi-rekurens-(Bagian2)-2024.pdf

Solusi relasi rekurens juga dapat dicari dengan pendekatan sistematis untuk relasi rekurens yang berbentuk homogen linear. Relasi rekurens homogen linear memiliki bentuk umum:

$$a_n + c_1 a_{n-1} + c_2 a_{n-2} + \ldots + c_k a_{n-k} = 0$$

dengan c_1, c_2, ..., c_k adalah konstanta, dan a_n adalah urutan yang dicari. Solusi diperoleh dengan mengubah relasi rekurens menjadi bentuk persamaan karakteristik, yaitu dengan mengganti $a_n = r^n$, sehingga menjadi:

$$r^k + c_1 r^{k-1} + c_2 r^{k-2} + \ldots + c_k = 0$$

lalu dicari akar-akarnya. Ada tiga kemungkinan hasil akar yang diperoleh:

1. Akar berbeda
   Jika akar-akar r_1, r_2, ..., r_k semuanya berbeda, solusi umum adalah:

   $$a_n = C_1 r_1^n + C_2 r_2^n + \ldots + C_k r_k^n$$

   dengan C_1, C_2, ..., C_k adalah konstanta yang ditentukan oleh kondisi awal.
2. Akar berulang
   Jika r adalah akar berulang dengan kelipatan m, maka solusi untuk r adalah:

   $$C_1 r^n + C_2 n r^n + C_3 n^2 r^n + \ldots + C_m n^{m-1} r^n$$

   Selanjutnya solusi ini digabung dengan solusi untuk akar lainnya.
3. Akar kompleks
   Jika akar-akar berupa bilangan kompleks $r = \alpha \pm i\beta$, maka solusi dapat ditulis dalam bentuk:

   $$r^n = (\alpha + i\beta)^n + (\alpha - i\beta)^n$$

Kondisi awal a_0, a_1, a_2, ..., a_{k-1} digunakan untuk menentukan nilai konstanta C_1, C_2, ..., C_k dengan menyusun sistem persamaan linear dari solusi umum.

B. Bilangan Pseudo-acak

Bilangan pseudo-acak atau *pseudo-random numbers* adalah deretan angka yang dihasilkan oleh algoritma komputer yang dirancang untuk meniru sifat-sifat bilangan acak yang sebenarnya. Meskipun disebut semu atau pseudo, bilangan pseudo-acak sangat berguna dalam berbagai aplikasi yang memerlukan angka yang tampak acak, karena bilangan ini memiliki tampak urutan yang terlihat acak secara statistik. Bilangan pseudo-acak memiliki beberapa karakteristik utama, yaitu:

1. Deterministik
   Meskipun terlihat acak, bilangan pseudo-random sebenarnya mengikuti aturan matematis yang pasti. Hal ini berarti bahwa jika diberikan nilai awal yang sama sebagai seed, hasil bilangan pseudo-acak memiliki urutan angka yang sama persis, seperti fungsi Hash.
2. Reprodusibel
   Karena memiliki sifat deterministik, bilangan psedo-acak dapat dihasilkan kembali dengan urutan angka yang sama dengan menggunakan nilai awal atau seed yang sama.
3. Periodik
   Setiap bilangan pseudo-acak memiliki periode berulang, yaitu panjang urutan angka yang tampak acak sebelum mengulangi pola yang sama. Namun, algoritma modern seringkali menghasilkan periodic yang sangat penjang, sehingga sifat ini jarang menjadi masalah.
4. Statistik
   Bilangan pseudo-acak dirancang untuk menghasilkan urutan angka yang memenuhi uji statistik untuk menunjukkan sifat keacaka, seperti distribusi yang seragam, independensi antar urutan, dan tidak adanya pola yang jelas.

Bilangan pseudo-acak memiliki banyak kegunaan dalam statsitika dan komputasi. Kegunaan-kegunaan tersebut, pada berbagai bidang, diantaranya adalah:

1. Simulasi
   Bilangan pseudo-acak banyak digunakan dalam simulasi berbagai fenomena acak di dunia nyata, seperti pergerakan partikel, cuaca, atau perilaku pasart keuangan.
2. Kriptografi
   Bilangan pseudo-acak dapat digunakan untuk menghasilkan kunci enkripsi yang kuat dan sulit di tebak, sehingga dapat membantu dalam meningkatkan keamanan siber. Meski demikian, hanya menggunakan pseudo-acak dalam kriptografi tidak disarankan, karena masih memiliki pola yang dapat dipecahkan.
3. Permainan
   Bilangan pseudo-acak dapat digunakan untuk menghasilkan angka acak yang banyak terdapat

dalam permainan, seperti dadu, kartu, drop item, dan lain sebagainya.

4. Sampling
   Bilangan pseudo-acak memiliki peran dalam pengambilan sampel data secara acak untuk keperluan statistik.

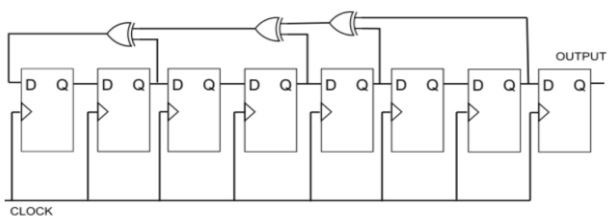
Bilangan pseudo-acak dapat dihasilkan melalui algoritma yang beragam, seperti LCG, LFSR, dna masih banyak lagi. Secara garis besar, bilangan pseudo-acak memiliki kegunaan yang hampir sama dengan bilangan acak, meskipun memiliki karakteristik yang berbeda.

C. Linear Feedback Shift Register (LFSR)

Linear Feedback Shift Register (LFSR) adalah sebuah algoritma rangkaian digital yang digunakan untuk menghasilkan urutan bit (0 atau 1) bilangan pseudo-acak. LFSR merupakan register geser (*shift register*) yang input bit-bitnya bergantung pada fungsi linear dari keadaan sebelumnya. Fungsi linear yang paling umum digunakan dalam LFSR adalah operasi XOR (eksklusif OR).

Register dalam LFSR terdiri dari beberapa blok berisi bit yang disebut *flip-flop*. Setiap *flip-flop* menyimpan satu bit data (0 atau 1). Data dalam setiap *flip-flop* akan bergeser satu posisi ke kanan pada setiap siklus *clock*. Bit input pada *flip-flop* di paling kiri (*most significant bit*) dihitung dengan melakukan operasi (misalnya XOR) beberapa output dari *flip-flop* lainnya yang diperhitungkan, yang disebut sebagai *taps*. Secara umum, algoritma LFSR adalah sebagai berikut:

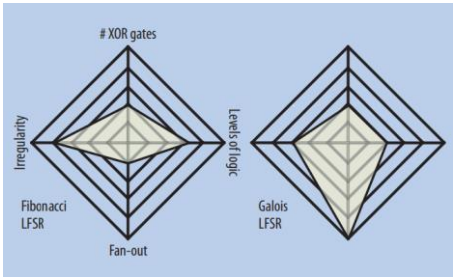
1. Inisialisasi
   LFSR dimulai dengan nilai awal register yang mengisi *flip-flop* yang disebut sebagai *seed*. Sebagai nilai awal, konfigurasi *seed* tidak boleh memiliki nilai 0 untuk setiap *flip-flop*-nya.
2. Pergeseran (*Shift*)
   Pada setiap sikuls, semua bit dalam LFSR bergeser satu posisi ke kanan (>> 1) secara lojik. Artinya, pergeseran tidak mengikuti aturan 2's *complement*, sehingga bit paling kiri pasti menjadi 0.
3. *Feedback*
   Bit yang paling kanan akan menjadi output LFSR, dan bit input baru di paling kiri (*most significant bit*) dihitung dengan melakukan operasi, umumnya XOR, dari bit-bit dalam *taps*.
4. Pengulangan
   Proses ini berulang secara rekursif sebagai definisi konsep LFSR. Urutan output pada setiap siklus menjadi urutan bit yang tampak acak dengan periode pola tertentu.



Gambar 2. LFSR dengan *taps* di bit 6, 4, 3, dan 1
Sumber: https://aircconline.com/ijcsit/V15N2/15223ijcsit03.pdf

Karakteristik LFSR ditentukan oleh panjang register dan *taps*. Kondisi LFSR akan kembali seperti *seed* setelah melewati periode tertentu. Besarnya periode ini ditentukan oleh *taps* yang dapat direpresentasikan dalam polinomial pada bidang Galois GF(2). Polinomial primitif pada GF(2) akan menghasilkan periode maksimum. Periode maksimum pola LFSR dengan panjang register sebanyak n bit, dapat ditentukan dengan rumus $2^n - 1$ untuk semua kemungkinan kombinasi seed.

Terdapat dua jenis LFSR, yaitu Fibonacci LFSR, dan Galois LFSR. Pada Fibonacci LFSR, *feedback* dilakukan dengan output *taps* ke input *flip-flop* yang paling kiri. Sedangkan pada Galois LFSR, *feedback* dilakukan dengan menghubungkan output *taps* ke input *flip-flop* yang berbeda-beda.



Gambar 3. Perbandingan Fibonacci LFSR dan Galois LFSR
Sumber: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5654516

D. Polinomial Primitif

Polinomial primitif adalah polinomial *irreducible* atas suatu medan hingga GF(q) yang akar-akarnya merupakan elemen primitive dari GF(q). Elemen primitif adalah elemen yang dapat menghasilkan semua elemen bukan nol dari GF(q) melalui perkalian berulang. Medan hingga, disebut juga Medan Galois, GF(q) adalah suatu medan dengan tepat q elemen, dengan q adalah bilanagn prima atau pangkat dari bilangan prima. Sebuah polinomial dikatakan tak tereduksi (*irreducible*) jika tidak dapat difaktorkan menjadi perkalian dari dua polinom dengan derajat yang lebih rendah pada medan yang sama.

Polinomial primitif memiliki beberapa karakteristik utama. Akar-akar dari polinomial primitif dapat digunakan untuk menghasilkan semua elemen bukan nol dari medan Galoisnya. Orde polinomial primitif adalah jumlah elemen yang dapat dihasilkan oleh akar-akarnya sebelum pengulangan terjadi. Orde ini sama dengan q – 1, dengan q adalah jumlah elemen dalam medan.

Jumlah polinomial primitif berderajat n pada medan GF(q) dapat ditentukan dengan rumus:

$$a_q(n) = \frac{\phi(q^n - 1)}{n}$$

dengan $\phi(n)$ adalah fungsi totient.

Polinomial primitif sudah dicari oleh peneliti-peneliti sebelumnya, sehingga pada praktiknya, polinomial primitif tidak perlu dicari ulang. Beberapa contoh polinomial primitif tertera dalam tabel berikut:

Tabel 1. Contoh Polinom Primitif

n	primitive polynomials
1	$1 + x$
2	$1 + x + x^2$
3	$1 + x + x^3, 1 + x^2 + x^3$
4	$1 + x + x^4, 1 + x^3 + x^4$
5	$1 + x^2 + x^5, 1 + x + x^2 + x^3 + x^5, 1 + x^3 + x^5, 1 + x + x^3 + x^4 + x^5,$ $1 + x^2 + x^3 + x^4 + x^5, 1 + x + x^2 + x^4 + x^5$

Sumber: https://mathworld.wolfram.com/PrimitivePolynomial.html

III. IMPLEMENTASI

A. Metodologi

Pada penerapan relasi rekurens dalam generator bilangan pseudo-acak berbasis LFSR yang telah dilakukan penulis, dilakukan hal-hal berikut dalam proses implementasinya.

1. Pemuatan tabel polinomial primitif dengan jumlah suku minimum untuk menghasilkan periode maksimum. Tabel polinomial primitif diperoleh dari penelitian yang telah dilakukan oleh Wayne Stahnke, yang dapat diakses pada: https://www.ams.org/journals/mcom/1973-27-124/S0025-5718-1973-0327722-7/S0025-5718-1973-0327722-7.pdf.
2. Penginputan seed atau register awal LFSR, panjang registernya, dan jumlah bit pseudo-acak yang hendak dihasilkan.
3. Penyesuaian seed agar memiliki panjang register yang sama dengan yang diinput.
4. Pemuatan polinom primitif pada derajat yang sama dengan panjang register. Polinom primitif berfungsi sebagai *taps* pada LFSR.
5. Proses eksekusi LFSR, dan ditampilkan di layar.

B. Program Source Code

Program ditulis dalam bahasa Python menggunakan *library* luar sebagai alat untuk visualisasi hasil angka output dari LFSR. Berikut adalah tangkapan *source code* dalam implementasi program.

Fungsi extractPolinom() digunakan untuk memuat data polinom primitif dari file txt. Data yang digunakan dalam penelitian ini terdiri dari polinom berderajat 1 hingga 168. Namun, untuk panjang register yang dapat digunakan adalah antara 2 sampai 168 bit register. Berikut adalah data yang digunakan.

Tabel 2. Polinomial Primitif pada GF(2)

1 0	57 7 0	113 9 0
2 1 0	58 19 0	114 82 81 1 0
3 1 0	59 22 21 1 0	115 15 14 1 0
4 1 0	60 1 0	116 71 70 1 0
5 2 0	61 16 15 1 0	117 20 18 2 0
6 1 0	62 57 56 1 0	118 33 0
7 1 0	63 1	119 8 0
8 6 5 10	64 4 3 1 0	120 118 111 7 0
9 4 0	65 18 0	121 18 0
10 3 0	66 10 9 1 0	122 60 59 1 0
11 2 0	67 10 9 1 0	123 2 0
12 7 4 3 0	68 9 0	124 37 0
13 4 3 1 0	69 29 27 2 0	125 108 107 1 0
14 12 11 1 0	70 16 15 1 0	126 37 36 1 0
15 1 0	71 6 0	127 1 0
16 5 3 2 0	72 53 47 6 0	128 29 27 2 0
17 3 0	73 25 0	129 5 0
16 7 0	74 16 15 1 0	130 3 0
19 6 5 1 0	75 11 10 1 0	131 48 47 1 0
20 3 0	76 36 35 1 0	132 29 0
21 2 0	77 31 30 1 0	133 52 51 1 0
22 1 0	78 20 19 1 0	134 57 0
23 5 0	79 9 0	135 11 0
24 0 3 1 0	80 38 37 0	136 126 125 1 0
25 3 0	81 4 0	137 21 0
26 8 7 1 0	82 38 35 3 0	138 8 7 1 0
27 8 7 1 0	83 46 45 1 0	139 8 5 3 0
28 3 0	84 13 0	140 29 0
29 2 0	85 28 27 1 0	141 32 31 1 0
30 16 15 1 0	86 13 12 1 0	142 21 0
31 3 0	87 13 0	143 21 20 1 0
32 28 27 1 0	88 72 71 1 0	144 70 69 1 0
33 13 0	89 38 0	145 52 0
34 15 14 1 0	90 19 18 1 0	146 60 59 1 0
35 2 0	91 84 83 1 0	147 38 37 1 0
36 11 0	92 13 12 1 0	148 27 0
37 12 10 2 0	93 2 0	149 110 109 1 0
38 6 5 1 0	94 21 0	150 53 0
39 4 0	95 11 0	151 3 0
40 21 19 2 0	96 49 47 2 0	152 66 65 1 0
41 3 0	97 6 0	153 1 0
42 23 22 1 0	98 11 0	154 129 127 2 0
43 6 5 1 0	99 47 45 2 0	155 32 31 1 0
44 27 26 1 0	100 37 0	156 116 115 1 0
45 4 3 1 0	101 7 6 1 0	157 27 26 1 0
46 21 20 1 0	102 77 76 1 0	158 27 26 1 0
47 5 0	103 9 0	159 31 0
48 28 27 0	104 11 10 1 0	160 19 18 1 0
49 9 0	105 16 0	161 18 0
50 27 26 1 0	106 15 0	162 88 87 1 0
51 16 15 1 0	107 65 63 2 0	163 60 59 1 0
52 3 0	108 31 0	164 14 13 1 0
53 16 15 1 0	109 7 6 1 0	165 31 30 1 0
54 37 36 1 0	110 13 12 1 0	166 39 38 1 0
55 24 0	111 10 0	167 6 0
56 22 21 1 0	112 45 43 2 0	168 17 15 2 0

Sumber: https://www.ams.org/journals/mcom/1973-27-124/S0025-5718-1973-0327722-7/S0025-5718-1973-0327722-7.pdf

Angka-angka yang terdapat dalam tabel merupakan pangkat dari suku yang memiliki koefisien 1 pada polinom primitif. Pada proses pengolahannya, angka-angka ini menjadi posisi bit pada *taps* yang bernilai 1, dan menjadi posisi bit pada register LFSR yang diperhitungkan dalam operasi XOR. Berikut adalah tampilan kode untuk fungsi extractPolinom()

```python
# Fungsi untuk mengekstrak data Polinomial Primitif
def extractPolinom(filename:str) -> list[list[int]]:
    '''Ekstrak polinom dari TXT ke bentuk list of list of int'''
    # Extract polinom data
    allPolinom = []
    with open(filename, 'r') as file:
        lines = file.readlines()
        for line in lines:
            line = line.strip()
            dataPolinom = [n for n in line.split(' ') if not n=='']
            dataPolinom = [int(n) for n in dataPolinom]
            allPolinom.append(dataPolinom)
    return allPolinom
```

Gambar 4. Fungsi extractPolinom
Sumber: Dokumen penulis

Setelah diekstrak, polinom primitif dapat dicari dengan derajat berupa panjang register yang diinput di main(). Polinom primitif dicari menggunakan fungsi searchPrimitivePolinom() berikut.

```python
# searchforPolinom
def searchPrimitivePolinom(degree:int, inList:bool, data:list[list[int]]):
    '''Menghasilkan polinom primitif berdasarkan derajatnya'''

    # Search for degree
    allPolinom = data
    index = -1
    for polinom in allPolinom:
        if polinom[0] == degree:
            index = allPolinom.index(polinom)
            break

    # Output
    if not index == -1:
        if inList:
            return allPolinom[index]
        else:
            finalBit = 0
            for bitPosition in polinom:
                bit = 1 << bitPosition
                finalBit = finalBit | bit
                if bitPosition == 0:
                    break
            return finalBit
    else:
        return None # not found
```

Gambar 5. Fungsi searchPrimitivePolinom()
Sumber: Dokumen penulis

Seed yang diinput harus melalui proses penyesuaian terlebih dahulu. Penyesuaian ini adalah pemotongan atau penambahan digit biner di kiri register untuk membuat *flip-flop* dengan panjang register yang diinginkan. Misalnya, jika input seed adalah 1234, dan panjang register adalah 8, angka biner 1234 yang merupakan (0b10011010010) dipotong menjadi 8 bit saja (0b11010010). Proses ini diimplementasi pada fungsi sesuiakanSeeds() berikut.

```python
# Fungsi penyesuaian seeds dengan jumlah register yang diminta
def sesuaikanSeeds(inputSeed:int, N_register:int) -> int:
    '''
    Menyesuaikan seeds dari input menjadi:
        - berbentuk 32 bit (karena maksimum polinom primitifnya derajat 31)
        - menghapus kelebihan bit: mengubah bits yang melebihi panjang register N_Register menjadi 0
    '''

    # Masking menjadi 32 bit
    hasil = inputSeed & 0xFFFFFFFF

    # Ambil sebanyak n bit terakhir
    maskReg = 0
    for i in range(N_register):
        maskReg += (1 << i)

    hasil = hasil & maskReg

    # Output
    return hasil
```

Gambar 6. Fungsi sesuaikanSeeds()
Sumber: Dokumen penulis

Proses dalam satu siklus *clock* LFSR dimulai dengan melakukan operasi XOR pada bit-bit di register LFSR atau *filp-flop* yang seusai dengan konfigurasi *taps*. Konfigurasi *taps* berbentuk bilangan biner yang memiliki panjang yang sama dengan register *flip-flop*. Angka 1 pada *taps* menunjukkan bahwa bit pada posisi tersebut di *flip-flop* diperhitungkan dalam operasi XOR, sedangkan angka 0 pada *taps* berarti tidak diperhitungkan. Impelentasi proses XOR ini dituliskan dalam fungsi xor() berikut ini.

```python
# Fungsi XOR sesuai polinom
def xor(reg:int, polinom:int) -> int:
    '''
    Mencari hasil XOR bit-bit tertentu pada reg yang sesuai dengan polinom
    contoh:
        xor(0b110001, 0b1101) akan melakukan operasi
        0 xor 0 xor 1

    Aturan Rekursif:
        - jika least significant bit polinom adalah 1,
        - maka least significant bit reg terlibat dalam XOR
    '''

    # Basis
    if polinom == 0b0:
        return 0b0

    # Rekurens
    else:
        # LSB polinom = 0
        if (polinom & 0b1) == 0b0:
            return xor(reg>>1, polinom>>1)
        # LSB polinom = 1
        else:
            return (reg & 0b1) ^ xor(reg>>1, polinom>>1)
```

Gambar 7. Fungsi xor()
Sumber: Dokumen penulis

Pada penerapan ini, fungsi XOR diimplementasi secara rekursif untuk melakukan operasi bitwise seperti melakukan pemrosesan list secara rekursif. Hasil operasi XOR dari fungsi tersebut, digunakan sebagai *feedback* untuk siklus berikutnya. Hasil XOR di input ke register di bit paling kiri setelah register digeser ke kanan sebanyak 1 bit. Proses ini diimplementasikan pada fungsi feedback() berikut ini.

```python
# Insert feedback ke depan
def feedback(reg:int, val:int, N_register:int) -> int:
    '''
    Menggeser register ke kanan,
    lalu memasukkan feedback ke most significant bit
    '''
    # Geser ke kanan
    reg = reg >> 1

    # Insert
    val = val << (N_register - 1)
    hasil = val | reg

    # Output
    return hasil
```

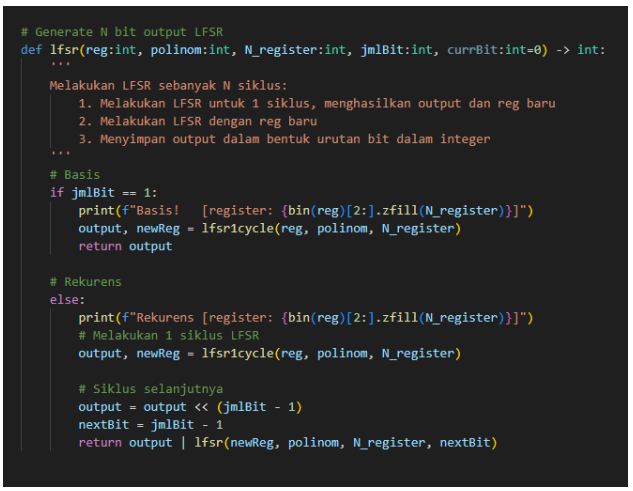
Gambar 8. Fungsi feedback()
Sumber: Dokumen penulis

Satu siklus *clock* LFSR merupakan gabungan operasi XOR dan proses *feedback* dengan menghasilkan output berupa *least significant bit* pada register *flip-flop* sebelum digeser. Proses ini diimplementasikan pada fungsi lfsr1cycle() berikut ini.

```python
# LFSR untuk Satu Siklus
def lfsr1cycle(reg:int, polinom:int, N_register:int) -> tuple[int, int]:
    '''
    Melakukan LFSR untuk satu siklus:
        1. Melakukan xor
        2. Insert feedback
    format return adalah tuple berisi < output, updated_register >
    '''
    # Mengambil Output
    output = reg & 0b1
    # Melakukan XOR
    feedbackValue = xor(reg, polinom)
    # Insert feedback
    updatedRegister = feedback(reg, feedbackValue, N_register)

    # Output
    return (output, updatedRegister)
```

Gambar 9. Fungsi lfsr1cycle()
Sumber: Dokumen penulis

Siklus *clock* LFSR diulang sebanyak jumlah bit yang diinginkan. Proses pengulangan ini melibatkan register baru yang telah dimanipulasi sebelumnya. Sehingga dalam implementasinya, proses ini melibatkan relasi rekurens. Perlu dicatat pula bahwa relasi rekurens juga dapat diubah ke dalam bentuk non-rekursif dengan mencari solusi dari relasi rekurens tersebut. Dalam pemrograman, proses rekursif juga dapat diubah ke bentuk iteratif. Namun, pada penerapan ini, dipilih proses rekursif sebagai penerapan relasi rekurens dalam generator bilangan pseudo-acak berbasis LFSR. Proses ini diimplementasi pada fungsi lfsr() berikut.



```python
# Generate N bit output LFSR
def lfsr(reg:int, polinom:int, N_register:int, jmlBit:int, currBit:int=0) -> int:
    '''
    Melakukan LFSR sebanyak N siklus:
        1. Melakukan LFSR untuk 1 siklus, menghasilkan output dan reg baru
        2. Melakukan LFSR dengan reg baru
        3. Menyimpan output dalam bentuk urutan bit dalam integer
    '''
    # Basis
    if jmlBit == 1:
        print(f"Basis!   [register: {bin(reg)[2:].zfill(N_register)}]")
        output, newReg = lfsr1cycle(reg, polinom, N_register)
        return output

    # Rekurens
    else:
        print(f"Rekurens [register: {bin(reg)[2:].zfill(N_register)}]")
        # Melakukan 1 siklus LFSR
        output, newReg = lfsr1cycle(reg, polinom, N_register)

        # Siklus selanjutnya
        output = output << (jmlBit - 1)
        nextBit = jmlBit - 1
        return output | lfsr(newReg, polinom, N_register, nextBit)
```

Gambar 10. Fungsi lfsr()
Sumber: Dokumen penulis

Semua proses tersebut digabung di fungsi main() sebagai program utama yang dijalankan. Selain dikeluarkan sebagai output urutan bit pseudo-acak, angka-angka ini juga dapat merepresentasikan bilangan integer. Sebagai contoh, apabila output LFSR adalah 11001011, output ini dapat direpresentasikan dalam desimal sebagai angka 203. Angka ini juga merupakan bilangan pseudo-acak, sehingga LFSR dapat tampak seperti menghasilkan urutan angka desimal 2, 0, dan 3 yang merupakan angka pseud-acak pula. Fungsi main(), visualisasi output, dan pengujian tertera dalam gambar berikut.

```python
# Main Program
def main():
    sys.setrecursionlimit(10037)
    tabelPolinom = extractPolinom('polinomPrimitif.txt')

    while True:
        terminalSize = shutil.get_terminal_size().columns

        # Intiate
        os.system("cls")
        print()
        print(f"[{'-'*((terminalSize-18)//2)}[ Program LFSR ]{'-'*((terminalSize-18)//2)}]")
        print()
        print(">> Masukkan Seeds    : ", end=""); inputSeeds = int(input())

        if inputSeeds > 0:
            print(">> Panjang Register  : ", end=""); n_reg = int(input())
            print(">> Jumlah bit Random : ", end=""); n_bit = int(input())
            print()

            # Hitung informasi
            period = countMaxPeriod(n_reg)

            # Hitung LFSR
            seeds = sesuaikanSeeds(inputSeeds, n_reg)
            polinom = searchPrimitivePolinom(n_reg, False, tabelPolinom)

            print("[program] Informasi hasil")
            print(f"\t[1] Seeds (disesuaikan)   : {seeds}")
            print(f"\t[2] Panjang Register      : {n_reg}")
            print(f"\t[3] Jumlah bits Random    : {n_bit}")
            print(f"\t[4] Polinom Primitif      : {bin(polinom)[2:]}")
            print(f"\t[5] Periode Berulang (max): {period}")

            hasil = lfsr(seeds, polinom, n_reg, n_bit)

            # Informasi hasil LFSR
            os.system("cls")
            print()
            print(f"[{'-'*((terminalSize-18)//2)}[ Program LFSR ]{'-'*((terminalSize-18)//2)}]")
            print()
            print("[program] Informasi hasil")
            print(f"    [1] Seeds (disesuaikan)     : {seeds}")
            print(f"    [2] Panjang Register        : {n_reg}")
            print(f"    [3] Jumlah bits Random      : {n_bit}")
            print(f"    [4] Polinom Primitif        : {bin(polinom)[2:]}")
            print(f"    [5] Periode Berulang (max)  : {period}")

            # Hasil LFSR
            print()
            print(f"[program] Hasil LFSR (bits)     : {bin(hasil)[2:].zfill(n_bit)}")
            print(f"[program] Hasil LFSR (int)      : {hasil}")
            print()
            print(f"[{'-'*(terminalSize-2)}]")

        # Exit
        else:
            break

        # Cek keacakan
        digitBin = [int(digit) for digit in str(bin(hasil)[2:])]
        digitDec = [int(digit) for digit in str(hasil)]
        visualizeSequence(digitBin)
        visualizeSequence(digitDec)
        # Menampilkan hasil uji
        print()
        print("\n[program] Binary Random Test")
        randomtestBin = testRandomness(digitBin)
        for test_name, result in randomtestBin.items():
            print(f"    {test_name:20}: {result}")
        print("\n[program] Decimal Random Test")
        randomtestDec = testRandomness(digitDec)
        for test_name, result in randomtestDec.items():
            print(f"    {test_name:20}: {result}")

        input("\n>> Press enter to continue: ")


    # End of Program
    os.system("cls")
    print()
    print(f"[{'-'*((terminalSize-18)//2)}[ Program LFSR ]{'-'*((terminalSize-18)//2)}]")
    print()
    print("[program] Berhasil keluar dari program")
    time.sleep(2)
    os.system("cls")
```

Gambar 11. Fungsi main()
Sumber: Dokumen penulis

```python
def visualizeSequence(sequence):
    '''Menampilkan visualisasi deretan angka dalam bentuk grafik garis'''
    # Panjang deretan angka
    x_values = list(range(len(sequence)))

    # Membuat grafik garis
    plt.figure(figsize=(10, 6))
    plt.plot(x_values, sequence, marker='o', linestyle='-', color='b', label="Deretan Angka")

    # Menambahkan label dan judul
    plt.title("Visualisasi Deretan Angka", fontsize=14)
    plt.xlabel("Indeks", fontsize=12)
    plt.ylabel("Nilai", fontsize=12)

    # Menambahkan grid dan legenda
    plt.grid(alpha=0.5)
    plt.legend()

    # Menampilkan grafik
    plt.show()
```

Gambar 12. Fungsi visualizeSequence()
Sumber: Dokumen penulis

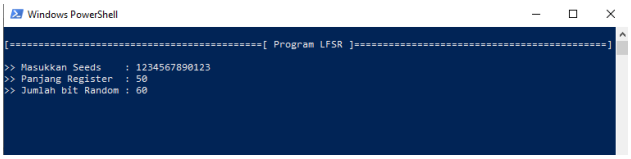
```python
def testRandomness(sequence):
    ''' Menguji keacakan deretan angka dengan beberapa metode '''
    results = {}

    # 1. Uji Mean dan Variance
    mean = np.mean(sequence)
    variance = np.var(sequence)
    results['Mean'] = mean
    results['Variance'] = variance

    # 2. Runs Test (uji pola beruntun)
    runs = 1
    for i in range(1, len(sequence)):
        if sequence[i] != sequence[i-1]:
            runs += 1
    expected_runs = (2 * len(sequence) - 1) / 3
    results['Runs Test'] = {
        'Observed Runs': runs,
        'Expected Runs': expected_runs,
        'Random': abs(runs - expected_runs) < expected_runs * 0.1
    }

    # 3. Autocorrelation Test (Ljung-Box)
    lb_test = acorr_ljungbox(sequence, lags=[len(sequence)-1], return_df=True)
    results['Autocorrelation Test'] = {
        'p-value': lb_test['lb_pvalue'].iloc[0],
        'Random': lb_test['lb_pvalue'].iloc[0] > 0.05
    }

    # 4. Chi-Square Test
    observed, bins = np.histogram(sequence, bins=10)
    expected = np.ones_like(observed) * np.sum(observed) / len(observed)
    chi2_stat, p_value = chisquare(observed, f_exp=expected)
    results['Chi-Square Test'] = {
        'Chi2 Stat': chi2_stat,
        'p-value': p_value,
        'Random': p_value > 0.05
    }

    return results
```

Gambar 13. Fungsi testRandomness()
Smber: Dokumen penulis

C. Tampilan Program dan Pengujain

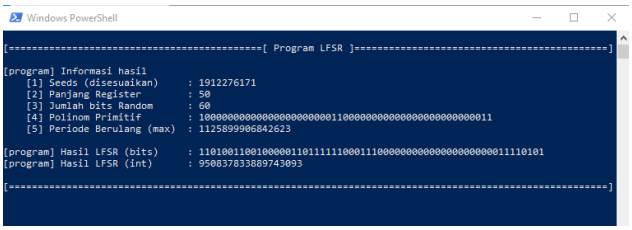
Penggunaan program meliputi penginputan seed, panjang register, dan jumlah bit pseudo-acak yang diinginkan. Hasil LFSR ditampilkan dalam bentuk urutan bit pseudo-acak, informasi mengenai hasil tersebut, dan representasi bilangan dari urutan bit pseudo-acak yang dihasilkan. Hasil ouput LFSR yang ditampilkan, juga disertai dengan hasil pengujian dan visualisasi angka-angka yang dihasilkannya.

Tampilan awal untuk menginput seed, panjang register, dan jumlah bit yang diinginkan, adalah sebagai berikut.



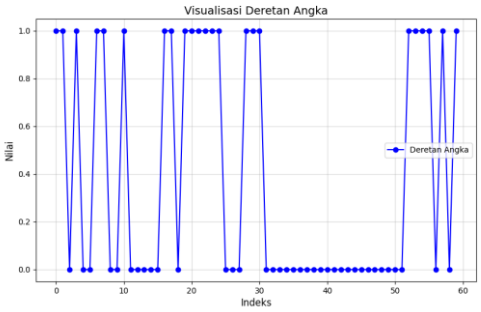
Gambar 14. Tampilan proses input
Sumber: Dokumen penulis

Proses LFSR dilakukan terhadap input tersebut, hingga menghasilkan 60 angka biner pseudo-acak. Hasil output dari program adalah sebagai berikut.

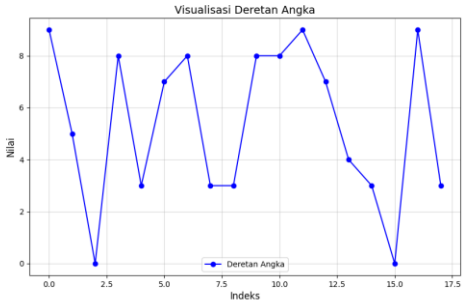


Gambar 15. Tampilan hasil LFSR
Sumber: Dokumen penulis

Selain menampilkan hasil LFSR, program juga menampilkan visualisasi output LFSR dalam bentuk diagram garis. Contoh hasil untuk untuk input tersebut adalah sebagai berikut.



Gambar 16. Visualisasi output biner LFSR
Sumber: Dokumen penulis

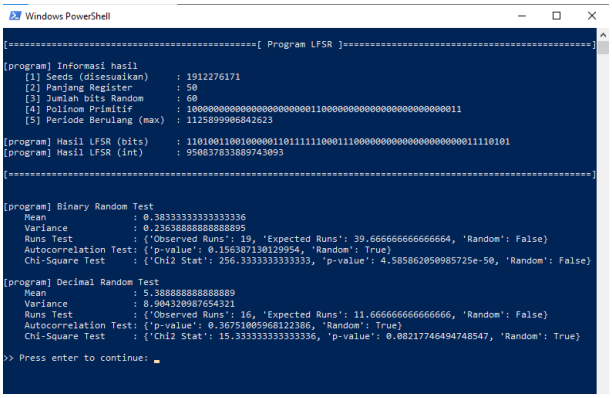


Gambar 17. Visualisasi output decimal LFSR
Sumber: Dokumen penulis

Selain itu, hasil LFSR juga disertai dengan hasil uji keacakan. Ada tiga uji keacakan yang digunakan, yaitu sebagai berikut:

1. *Runs Test*
   Tes ini memeriksa apakah urutan bilangan pseudo-random menunjukkan pola tertentu (*runs*). *Run* adalah urutan bilangan yang semua nilainya meningkat atau menurun secara berurutan. Tujuannya adalah untuk memeriksa independensi dan keacakan dalam urutan bilangan pseudo-acak. Jika urutan terlalu banyak atau terlalu sedikit runs dibandingkan dengan ekspektasi, maka angka yang dihasilkan mungkin tidak acak.
2. *Autocorrelation Test*
   Tes ini mengevaluasi hubungan antara bilangan pseudo-acak yang berjarak tertentu (*lag*). Tujuannya adalah untuk mendeteksi korelasi antara bilangan yang dihasilkan pada *lag* tertentu. Dalam bilangan pseudo-acak yang ideal, nilai R_d harus mendekati nol untuk semua *lag d*.
3. *Chi-Square Test*
   Tes ini mengukur apakah distribusi bilangan pseudo-random mendekati distribusi uniform. Tujuannya adalah untuk untuk mengevaluasi keseragaman bilangan pseudo-acak. Nilai X^2 dibandingkan dengan tabel distribusi chi-square

pada tingkat signifikansi tertentu (α). Jika X^2 terlalu besar, distribusi bilangan tidak uniform.



Gambar 15. Hasil pengujian keacakan
Sumber: Dokumen penulis

IV. PEMBAHASAN

A. Relasi Rekurens dalam LFSR

Berdasarkan terminologinya, generator bilangan pseudo-acak berbasis *Linear Feedback Shift Register* (LFSR) merupakan relasi rekurens yang dituliskan sebagai:

$$x_{n+k} = c_1 x_{n+k-1} \oplus c_2 x_{n+k-2} \oplus \ldots \oplus c_k x_n$$

dengan,

1. x_i: bit ke-i dalam urutan output
2. $c_1, c_2, \ldots, c_k$: koefisien feedback (bernilai 0 atau 1). Nilai ini ditentukan oleh polinomial karakteristik LFSR
3. k: panjang register
4. n: nomor siklus
5. $\oplus$: operasi XOR

Relasi ini memiliki bentuk polinomial karakteristik atau *feedback polynomial* yang berbentuk:

$$P(x) = x^k + c_1 x^{k-1} + c_2 x^{k-2} + \ldots + c_{k-1} x + c_k$$

Polinomial ini menentukan posisi register atau *flip-flop* yang berkontribusi dalam relasi rekurens. Koefisien $c_i = 1$ menunjukkan bahwa bit dari register ke-i (dari kanan ke kiri) digunakan dalam perhitungan *feedback*. Polinomial ini disebut juga dengan istilah *taps* pada pemrogramannya. Jika polinomial P(x) adalah prima/*irreducible* pada medan Galois GF(2), periode pengulangan pola pseudo-acak mencapai periode maksimum (2^k - 1). Polinomial yang memenuhi kondisi ini adalah polinomial primitif.

B. Hasil Implementasi dan Pengujian

Setelah dilakukan implementasi program generator bilangan pseudo-acak berbasis LFSR dan dilakukan pengujian, diperoleh hasil yang cukup sesuai dengan teori. Hasil menunjukkan bahwa output LFSR berupa urutan angka biner berhasil lolos uji *Autocorrelation Test*, tetapi gagal dalam *Runs Test* dan *Chi-Square Test*. Sedangkan, ouput LFSR yang telah diubah ke bentuk integer dan diperiksa berdasarkan urutan digit desimalnya, berhasil lolos uji *Autocorrelation Test* dan *Chi-Suqare Test*, tetapi juga gagal dalam *Runs Test*.

Hasil ini membuktikan bahwa LFSR bersifat deterministik, dan tidak sepenuhnya acak. Angka yang dihasilkan LFSR sangat dipengaruhi oleh *seed*. Dalam uji *runs* misalnya, untuk *taps* yang sama, *seed* tertentu dapat menghasilkan urutan yang cenderung lebih sering 0, atau lebih sering 1, dan ada pula yang dapat menghasilkan terlalu banyak perubahan. Hal seperti ini mempengaruhi hasil uji *runs* yang mengukur seberapa sering hasil berganti antar urutan output. Adapun kegagalan uji *runs* pada tipe desimal juga dapat disebabkan oleh karakteristik angka desimal yang dapat berubah drastis terhadap perubahan binernya, ditunjukkan dengan hsil *runs test* yang melebihi hasil yang diharapkan. *Autocorrelation test* yang berhasil, menunjukkan bahwa LFSR tidak bergantung pada output, tetapi bergantung pada register yang menghasilkan output tersebut.

Secara garis besar, hasil ini menunjukkan bahwa LFSR memang tidak diciptakan untuk kriptografi dengan keamanan tinggi, tetapi untuk kebutuhan simulasi dan *sampling* dalam statistika. Relasi rekurens, polinomial karakteristik, dan *seed* sangat mempengaruhi hasil LFSR. Apabila LFSR hendak digunakan dalam kriptografi, hendaknya dipadukan dengan teknik lain. Dalam penerapan ini, pengubahan barisan bilangan biner menjadi integer besar dapat membantu dalam menyamarkan ketidakacakan hasil output LFSR. Namun, hal ini tidak membuat hasil benar-benar acak.

V. KESIMPULAN

Relasi rekurens dalam *Linear Feedback Shift Register* (LFSR) memiliki peran penting dalam menghasilkan bilangan pseudo-acak untuk berbagai keperluan komputasi, seperti simulasi, sampling, dan membantu dalam kriptografi. Proses rekursif yang digunakan dalam LFSR, dengan bantuan polinomial karakteristik, memungkinkan LFSR untuk menghasilkan urutan bilangan biner yang memiliki keacakan yang cukup memadai untuk simulasi, tetapi kurang acak untuk kriptografi karena tidak sepenuhnya acak. Hasil pengujian menunjukkan bahwa meskipun LFSR dapat lolos uji *Autocorrelation Test*, LFSR pada penerapan ini gagal dalam uji *Runs Test* dan *Chi-Square Test*. Hal ini menandakan bahwa bilangan yang dihasilkan tidak sepenuhnya acak, melainkan bersifat deterministik. Pengubahan urutan bilangan biner menjadi integer besar dapat membantu menyamarkan pola yang dihasilkan oleh LFSR, meskipun hal ini tidak sepenuhnya menghilangkan keteraturan dalam output. Secara keseluruhan, LFSR menunjukkan potensi yang signifikan dalam aplikasi yang membutuhkan keacakan, tetapi memiliki keterbatasan

dalam penggunaan untuk kriptografi yang memerlukan tingkat keamanan tinggi. Selain itu, penelitian ini menunjukkan bahwa relasi rekurens memiliki penerapan yang sangat berguna dalam generator bilangan pseudo-acak berbasis LFSR.

REFERENSI

[1] R. Munir, Deretan, Rekursi, dan Relasi Rekurens Bagian 1. IF1220 Matematika Diskrit - Semester I Tahun 2024/2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/10-Deretan,%20rekursi-dan-relasi-rekurens-(Bagian1)-2024.pdf

[2] R. Munir, Deretan, Rekursi, dan Relasi Rekurens Bagian 2. IF1220 Matematika Diskrit - Semester I Tahun 2024/2025. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2024-2025/11-Deretan,%20rekursi-dan-relasi-rekurens-(Bagian2)-2024.pdf

[3] A. Khalique, A. H. Lone, and S. S. Ashraf, "A Novel Unpredictable Temporal Based Pseudo Random Number Generator," Int. J. Comput. Appl., vol. 117, no. 13, pp. 24, May 2015. [Online]. Available: https://research.ijcaonline.org/volume117/number13/pxc3903301.pdf

[4] K. M. U. Maheswari, R. Kundu, and H. Saxena, "Pseudo Random Number Generators Algorithms and Applications," Int. J. Pure Appl. Math., vol. 118, no. 22, pp. 331–336, 2018. [Online]. Available: https://acadpubl.eu/hub/2018-118-22/articles/22a/48.pdf

[5] N. Mukherjee, J. Rajski, G. Mrugalski, A. Pogiel, and J. Tyszer, "Ring Generator: An Ultimate Linear Feedback Shift Register," IEEE Comput., vol. 44, no. 6, pp. 64–71, Jun. 2011. doi: 10.1109/MC.2010.334. [Online]. Available: https://ieeexplore.ieee.org/document/5654516

[6] F. J. Manuel, M. G. R. Francisco, and M. P. J. Francisco, "Linear Feedback Shift Register Genetically Adjusted for Sequence Copying," Int. J. Comput. Sci. Inf. Technol., vol. 15, no. 2, Apr. 2023. doi: 10.5121/ijcsit.2023.15203. [Online]. Available: https://aircconline.com/ijcsit/V15N2/15223ijcsit03.pdf

[7] T. Hansen and G. L. Mullen, "Primitive Polynomials Over Finite Fields," Math. Comput., vol. 59, no. 200, pp. 639–643, Oct. 1992. [Online]. Available: https://www.ams.org/journals/mcom/1992-59-200/S0025-5718-1992-1134730-7/S0025-5718-1992-1134730-7.pdf

[8] W. Stahnke, "Primitive Binary Polynomials," Math. Comput., vol. 27, no. 124, Oct. 1973. [Online]. Available: https://www.ams.org/journals/mcom/1973-27-124/S0025-5718-1973-0327722-7/S0025-5718-1973-0327722-7.pdf

[9] E. W. Weisstein, "Primitive Polynomial," MathWorld--A Wolfram Web Resource, Jan. 2025. [Online]. Available: https://mathworld.wolfram.com/PrimitivePolynomial.html

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Jatinangor, 8 Januari 2025

Muhamad Nazih Najmudin
NIM: 13523144