Глава 7 (Сазанович Владислав M3339)

In [682]:
```
import numpy as np
import pandas as pd
import math
import matplotlib.pyplot as plt
import scipy
import scipy.stats
import hashlib
import time
import copy

from numpy.linalg import matrix_rank
from tqdm import tqdm
```

In [684]:
```
# generate all sequences of length l
def generate(l):
    res = []

    for i in range(0, 2**l):
        b = bin(i)[2:]
        b = '0' * (l - len(b)) + b
        b = np.array(list(map(lambda x: int(x), b)))
        res.append(b)

    return np.array(res)
```

Processing math: 100%

In [685]:
```
class GaloisField:
    def __init__(self, q):
        self.q = q

    def add(self, x, y):
        return (x + y) % self.q

    def mult(self, x, y):
        return (x * y) % self.q

    def sub(self, x, y):
        return (x - y + self.q) % self.q

    def add_poly(self, x, y):
        l = x
        r = y
        if len(x) > len(y):
            l = y
            r = x

        return self.add(np.append(l, np.zeros(len(r) - len(
l)).astype(int)), r)

    def sub_poly(self, x, y):
        return self.add_poly(x, -y)
```

In [686]:
```
def polynomial_to_string(P):
    p = ''
    for i in range(len(P)):
        if P[i] == 0:
            continue
        if i == 0:
            p = p + '{} + '.format(int(P[i]))
            continue

        if P[i] != 1:
            p = p + '{}*'.format(int(P[i]))
        if i == 1:
            p = p + 'x + '
        else:
            p = p + 'x^{} + '.format(i)

    return p[:-2]

def print_polynomial(P, name = None):
    p = polynomial_to_string(P)

    if name is None:
        print(p)
    else:
        print(name, p)
```

In [687]:
```
print_polynomial([1,2,1,0])
```

```
1 + 2*x + x^2
```

Processing math: 100%

Задание 1

In [688]:

```
# S - синдромный многочлен в GF(q)
def BM_algo(S, q, d):
    GF = GaloisField(q)
    Bs = []
    As = []
    Ls = []
    deltas = []

    L = 0 # Длина регистра
    A = np.array([1]) # Многочлен локаторов
    B = np.array([1]) # Многочлен компенсации невязки

    delta = 0 # невязка
    for r in range(1, d):
        Bs.append(polynomial_to_string(B))
        As.append(polynomial_to_string(A))
        Ls.append(L)
        deltas.append(delta)

        delta = 0
        for j in range(L + 1):
            cur = GF.mult(S[r - j], A[j])
            delta = GF.add(delta, cur)

        B = np.insert(B, 0, 0) # Сдвиг
        if delta != 0:
            T = GF.sub_poly(A, GF.mult(delta, B))
            if 2 * L <= r - 1:
                B = GF.mult(delta, A)
                L = r - L
            A = T


    Bs.append(polynomial_to_string(B))
    As.append(polynomial_to_string(A))
    Ls.append(L)
    deltas.append(delta)

    data = pd.DataFrame({'s': S[:d], 'd': deltas, 'A': As,
'B': Bs, 'L': Ls})
    data = data[['s', 'd', 'B', 'A', 'L']]
    data.index.rename('r', inplace=True)


    if len(A) == L + 1:
        return A, data
    else:
        return None, data
```

Проверим алгоритм на примере из учебника

Processing math: 100%

In [689]:
```
S = np.array([1, 2, 1, 0, 1, 2, 1, 0, 1, 2, 1, 0, 1])
A, D = BM_algo(S, 3, 8)
D
```

Out[689]:

	s	d	B	A	L
r					
0	1	0	1	1	0
1	2	2	2	1 + x	1
2	1	0	2*x	1 + x	1
3	0	1	1 + x	1 + x + x^2	2
4	1	2	x + x^2	1 + 2*x + 2*x^2	2
5	2	1	1 + 2*x + 2*x^2	1 + 2*x + x^2 + 2*x^3	3
6	1	0	x + 2*x^2 + 2*x^3	1 + 2*x + x^2 + 2*x^3	3
7	0	0	x^2 + 2*x^3 + 2*x^4	1 + 2*x + x^2 + 2*x^3	3

Возьмем теперь произвольную последовательность

Processing math: 100%

In [690]:
```
# S = np.array([0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0])
S = np.array(list('110101101101')).astype(int)
A, D = BM_algo(S, 2, 12)
D
```

Out[690]:

	s	d	B	A	L
r					
0	1	0	1	1	0
1	1	1	1	1 + x	1
2	0	1	x	1	1
3	1	1	1	1 + x^2	2
4	0	0	x	1 + x^2	2
5	1	0	x^2	1 + x^2	2
6	1	1	1 + x^2	1 + x^2 + x^3	4
7	0	1	x + x^3	1 + x + x^2	4
8	1	0	x^2 + x^4	1 + x + x^2	4
9	1	0	x^3 + x^5	1 + x + x^2	4
10	0	0	x^4 + x^6	1 + x + x^2	4
11	1	0	x^5 + x^7	1 + x + x^2	4

P.S. Нарисованный ЛРОС находиться на последней странице

Теперь продлим последовательность еще на 10 символов.

In [691]:
```
print('initial: {}\n'.format(S))

extended_S = S
for i in range(10):
    extended_S = np.append(extended_S, [np.sum(A * S[-len(A
):]) % 2])

print('extended: {}\n'.format(extended_S))
```

```
initial: [1 1 0 1 0 1 1 0 1 1 0 1]

extended: [1 1 0 1 0 1 1 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0]
```

Задание 6

Processing math: 100%

Для начала создадим несколько helper методов.

In [692]:
```
# Вычисляет p(alphas[j])
# Предполагается что полином записан в обратном порядке т.е
. [1, 0, 1, 1] = x^3 + x + 1
def p_alpha(p, alphas, j):
    s = 0

    for n in range(1, len(p) + 1):
        if p[-n] == 1:
            s = (s + alphas[(j * (n - 1)) % len(alphas)]) %
2

    return s
```

In [693]:
```
# Определить j : alphas[j] = p
def locate(p, alphas):
    for i in range(len(alphas)):
        if np.all(np.equal(p, alphas[i])):
            return i
```

In [694]:
```
# Коневертация начального числа в последовательность бит
def convert_to_binary_charwise(num, m):
    bn =''
    for c in str(num):
        binary = np.binary_repr(int(c))
        binary = '0' * (3 - len(binary)) + binary
        bn = bn + binary

    bn = np.array(list('0' + bn))
    if len(bn) < 2**m - 1:
        np.append(np.zeros(2**m - 1 - len(bn)), bn)

    return bn[-32:].astype(int)
```

In [695]:
```
# Начальные данные
# p = x^5 + x^3 + 1
p = np.array([1, 0, 1, 0, 0, 1]) # примитивный полином
m = 5 # полиномы из поля GF(2^m)
d = 5 # желаемое расстояние кода (исправление двух ошибок)

v = convert_to_binary_charwise('1674435744', m) # входное с
лово
print(v)
```

```
[0 0 0 1 1 1 0 1 1 1 1 0 0 1 0 0 0 1 1 1 0 1 1 1 1 1 0 0 1
0 0]
```

Processing math: 100%

In [696]:
```
# Подсчет циклотомических классов
def get_cyclic_classes(m, prt=False):
    cyclic = [np.zeros(1).astype(int)]

    taken = set()
    taken.add(0)

    for i in range(2**m - 1):
        if i in taken:
            continue
        c_class = [i]
        current = i
        while True:
            taken.add(current)
            current = (current * 2) % (2**m - 1)
            if current == i or current in taken:
                break
            else:
                c_class.append(current)
        if prt:
            print('C{}: {}'.format(c_class[0], c_class))
        cyclic.append(np.array(c_class))
    return cyclic

cyclic = get_cyclic_classes(m, prt=True)
```

```
C1: [1, 2, 4, 8, 16]
C3: [3, 6, 12, 24, 17]
C5: [5, 10, 20, 9, 18]
C7: [7, 14, 28, 25, 19]
C11: [11, 22, 13, 26, 21]
C15: [15, 30, 29, 27, 23]
```

Проверим что x является примитивным элементом по модулю p

Processing math: 100%

In [697]:
```
# Фунцкия возводящая x в степени 0 .. qm по модулю p
def get_all_pows_of_x(p, m, prt=False):
    unique = set()

    alphas = []

    x = np.array([1]).astype(int) # x^1
    for i in range(0, 2**m - 1):
        poly = (np.polydiv(x, p)[1] + 2) % 2
        poly = np.append(np.zeros(m - len(poly)).astype(int
), poly)
        alphas.append(poly)
        unique.add(tuple(poly))
        if prt:
            print(i, polynomial_to_string(poly[::-1]))
        x = np.append(x, [0])

    if prt:
        print('Unique: {}, All: {}'.format(len(unique), 2**
m-1))

    return alphas
```

Processing math: 100%

In [698]:
```
alphas = get_all_pows_of_x(p, m, prt=True)
```

```
0 1
1 x
2 x^2
3 x^3
4 x^4
5 1 + x^3
6 x + x^4
7 1 + x^2 + x^3
8 x + x^3 + x^4
9 1 + x^2 + x^3 + x^4
10 1 + x + x^4
11 1 + x + x^2 + x^3
12 x + x^2 + x^3 + x^4
13 1 + x^2 + x^4
14 1 + x
15 x + x^2
16 x^2 + x^3
17 x^3 + x^4
18 1 + x^3 + x^4
19 1 + x + x^3 + x^4
20 1 + x + x^2 + x^3 + x^4
21 1 + x + x^2 + x^4
22 1 + x + x^2
23 x + x^2 + x^3
24 x^2 + x^3 + x^4
25 1 + x^4
26 1 + x + x^3
27 x + x^2 + x^4
28 1 + x^2
29 x + x^3
30 x^2 + x^4
Unique: 31, All: 31
```

x действительно является примитивным элементом по модулю $p = x^5 + x^3 + 1$

Посчитаем минимальные полиномы:

Processing math: 100%

In [699]:
```
def get_minimal_Ms(alphas, cyclic, m, prt=False):
    Ms = []
    for i in range(len(cyclic)):
        n = len(cyclic[i])
        poly = [0] * (n + 1)

        for seq in generate(n):
            ones = np.count_nonzero(seq)
            zeros = n - ones

            power = 0
            for j in range(len(seq)):
                if seq[j] == 1:
                    power = (power + (2**m - 1 - cyclic[i][
j])) % (2**m - 1)

            poly[zeros] = (poly[zeros] + alphas[power]) % 2

        poly_converted = []
        for j in range(len(poly)):
#             print(poly[j])
            ones = np.count_nonzero(poly[j])
#             print(poly[j])
            assert ones <= 1
            poly_converted.append(ones)

        if prt:
            print('M{}: {}'.format(cyclic[i][0], polynomial
_to_string(poly_converted[::-1])))
        Ms.append(np.array(poly_converted))
    return Ms
```

In [700]:
```
Ms = get_minimal_Ms(alphas, cyclic, m, prt=True)
```

```
M0: 1 + x
M1: 1 + x^3 + x^5
M3: 1 + x + x^2 + x^3 + x^5
M5: 1 + x + x^3 + x^4 + x^5
M7: 1 + x^2 + x^3 + x^4 + x^5
M11: 1 + x + x^2 + x^4 + x^5
M15: 1 + x^2 + x^5
```

Нам нужно расстояние $d = 5$. Можно взять полиномы M_1, M_3 так как в объединение их циклотомических классов есть последовательность соседних элементов длины 4.

In [701]:
```
g = np.polymul(Ms[1], Ms[2]) % 2
print('g(x) = {}'.format(polynomial_to_string(g[::-1])))
```

```
g(x) = 1 + x + x^2 + x^4 + x^5 + x^7 + x^10
```

Должно быть $g(alpha^i) = 0, i = 1..4$ Тогда расстояние кода будет равно 5. Проверим что это так:

Processing math: 100%

In [702]:
```
start = 1 # начало альф : g(alpha^i) = 0
# g(alpha^i) должен быть = 0 для i = 1..(d-1)
for i in range(start, start + d - 1):
    print(p_alpha(g, alphas, i)) # Вычисление g(alpha^i) с
помощью helper функции
```

```
[ 0.  0.  0.  0.  0.]
[ 0.  0.  0.  0.  0.]
[ 0.  0.  0.  0.  0.]
[ 0.  0.  0.  0.  0.]
```

Напишем helper функции для алгоритма ПГЦ. Научимся считать синдромные компоненты:

In [705]:
```
def get_syndrom_components(v, start, d):
    ss = []
    for j in range(start, start + d - 1):
        ss.append(p_alpha(v, alphas, j)) # v(alpha^j)

    n_ss = []
    for j in range(len(ss)):
        n_ss.append(locate(ss[j], alphas)) # находим n : ss
[j] = alpha^n
    return ss, n_ss
```

Напишем функцию для нахождения корней полинома в поле.

In [706]:
```
def get_roots_2(p, m):
    power = 2 ** m - 1;

    # transform to make first element 0
    add = power - p[0]
    for i in range(len(p)):
        p[i] = (p[i] + add) % power

    for x in range(power):
        for y in range(power):
            if (x + y) % power != p[2]:
                continue

            sum = (alphas[x] + alphas[y]) % 2
            k = locate(sum, alphas)
            if (k != p[1]):
                continue
            return [x, y]
    return None
```

Решение системы в поле $GF(2^m)$:

Processing math: 100%

In [715]:
```
def get_solution_2(matrix, result, m):
    for x in range(2**m):
        for y in range(2**m):

            result_check = []
            check = [x, y]
            for i in range(len(matrix)):
                s = np.zeros(m)
                for j in range(len(matrix[0])):
                    k = (matrix[i][j] + check[j]) % (2**m-1
)
                    s += alphas[k]
                s %= 2
                result_check.append(locate(s, alphas))
            if (np.all(result_check == result)):
                return check

    return [-1, -1]
```

In [716]:
```
# Фунция для нахождения лямбд. Передаем систему - получем м
ассив лямбд
def get_lambdas_2(S):
    return get_solution_2(
        [
            [S[0], S[1]],
            [S[1], S[2]]
        ],
        [S[2], S[3]],
        5
    )
```

Сам алгоритм. Считаем синдромные компоненты, находим лямбды, находим корни полинома локатором ошибок.

Processing math: 100%

In [813]:
```
def solve_pgc(v, start, d):
    print('v = {}'.format(v))
    S, nS = get_syndrom_components(v, start, d)
    print('S = {}'.format(nS))
    lambdas = get_lambdas_2(nS)

    if lambdas is None:
        return []

    print('lambdas = {}'.format(lambdas))

    lambda_polynomial = np.zeros(3)
    lambda_polynomial[-1] = 0
    lambda_polynomial[-2] = lambdas[1]
    lambda_polynomial[-3] = lambdas[0]
    print('lambda poly = {}'.format(lambda_polynomial))

    lambda_roots = get_roots_2(lambda_polynomial, m)
    if lambda_roots is None:
        return []

    lambda_roots = [((2**m - 1) - root) % (2**m - 1) for ro
ot in lambda_roots]
    print('error positions = {}'.format(lambda_roots))
    return lambda_roots
```

Напишем функцию для проверки алгоритма:

In [791]:
```
# invert one bit
def invert(v, i):
    v[i] = (v[i] + 1) % 2

def check_algo(f, g, errors):

    # create code word in the form f * g
    temp = np.polymul(f, g) % 2
    c = np.zeros(2**m - 1)
    for i in range(1, len(temp) + 1):
        c[-i] = temp[-i]

    print('c = {}'.format(c))
    # if 0 => c is code word
    print(np.polydiv(c, g)[1] % 2)

    # add errors
    v = np.array(c)
    for e in errors:
        invert(v, -(e + 1))

    print(solve_pgc(v, start, d))
```

Processing math: 100%

Проверим алгоритм. Должно получаться $errorpositions$ = номерам ошибок которые мы передали (третий аргумент в функции)

In [792]:
```
check_algo([1,0], g, [11, 22])
```

```
c = [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0
.  0.  0.  0.  0.
  0.  1.  0.  0.  1.  0.  1.  1.  0.  1.  1.  1.  0.]
[ 0.]
v = [ 0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0
.  0.  0.  0.  0.
  0.  0.  0.  0.  1.  0.  1.  1.  0.  1.  1.  1.  0.]
S = [3, 6, 30, 12]
lambdas = [2, 3]
lambda poly = [ 2.  3.  0.]
error positions = [22, 11]
[22, 11]
```

In [793]:
```
check_algo([1,1,1], g, [7, 13])
```

```
c = [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0
.  0.  0.  0.  0.
  1.  1.  1.  1.  1.  0.  0.  0.  0.  0.  1.  0.  1.]
[ 0.  0.  0.  0.  0.  0.  0.  0.]
v = [ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0
.  0.  0.  0.  1.
  1.  1.  1.  1.  1.  1.  0.  0.  0.  0.  1.  0.  1.]
S = [17, 3, 7, 6]
lambdas = [20, 17]
lambda poly = [ 20.  17.   0.]
error positions = [13, 7]
[13, 7]
```

Теперь попытаемся декодировать последовательность из примера. Сходы не получилось, поэтому напишем функцию перебора всех возможных комбинаций минимальных полиномов, для которых в объединении соответствующих циклотомических классов есть последовательность из $d-1$ соседних элементов.

Processing math: 100%

In [800]:
```
# Находит начало последовательности из d - 1 соседних элеме
нтов. В объединение циклотомических классов cycle_numbers
def check_cycle(cycle_numbers, d):
    all = set()
    for cc in cycle_numbers:
        all |= set(cyclic[cc])

    if 0 in all:
        current = 1
    else:
        current = 0

    for n in range(1, 2**m):
        if n in all:
            current += 1
        else:
            current = 0

        if current >= d - 1:
            return n - current + 1

    return -1
```

Функия для проверки кодирования / декодирования

In [805]:
```
def check_code_decode(v, cycle_classes, start, d, g):
    print('CHECK')
    v_copy = np.array(v)
    # получили позиции ошибок для выбранного кода.
    # заметим для декодирования нужно знать только начало п
оследовательности соседних элементов и расстояние
    errors = solve_pgc(v, start, d)

    # исправим ошибки
    for e in set(errors):
        invert(v_copy, e)

    # чтобы последовательность была кодовым словом, нужно ч
тобы остаток от деления на g был равен = 0
    res = np.polydiv(v_copy, g)[1] % 2

    print('cycle classes = {}'.format(cycle_classes))
    print('g mod v = {}'.format(res))
```

Processing math: 100%

In [811]:
```
# Перебираем все комбинации минимальных полиномов и пытаемс
я декодировать
def check_codes(v):
    for poly_g in generate(7)[1:]:
        g = [1]
        cycle_classes = []
        for j in range(7):
            if poly_g[j] == 1:
                cycle_classes.append(j)
                g = np.polymul(g, Ms[j]) % 2

        if check_cycle(cycle_classes, 7):
            check_code_decode(v, cycle_classes, start, 7, g
)
```

Processing math: 100%

In [812]:
```
check_codes(v) # полный вывод приложил в файле Block7pgc_ou
t
```

Processing math: 100%

```
CHECK
v = [0 0 0 1 1 1 0 1 1 1 1 0 0 1 0 0 0 1 1 1 0 1 1 1 1 1 0
0 1 0 0]
S = [2, 4, 11, 8, 19, 22]
lambdas = [7, 2]
lambda poly = [ 7.  2.  0.]
error positions = [25, 13]
cycle classes = [6]
g mod v = [ 1.  0.  0.  1.  1.]
CHECK
v = [0 0 0 1 1 1 0 1 1 1 1 0 0 1 0 0 0 1 1 1 0 1 1 1 1 1 0
0 1 0 0]
S = [2, 4, 11, 8, 19, 22]
lambdas = [7, 2]
lambda poly = [ 7.  2.  0.]
error positions = [25, 13]
cycle classes = [5]
g mod v = [ 1.  0.  1.  1.  0.]
CHECK
v = [0 0 0 1 1 1 0 1 1 1 1 0 0 1 0 0 0 1 1 1 0 1 1 1 1 1 0
0 1 0 0]
S = [2, 4, 11, 8, 19, 22]
lambdas = [7, 2]
lambda poly = [ 7.  2.  0.]
error positions = [25, 13]
cycle classes = [5, 6]
g mod v = [ 0.  1.  1.  1.  0.  0.  0.  0.  0.  0.]
CHECK
v = [0 0 0 1 1 1 0 1 1 1 1 0 0 1 0 0 0 1 1 1 0 1 1 1 1 1 0
0 1 0 0]
S = [2, 4, 11, 8, 19, 22]
lambdas = [7, 2]
lambda poly = [ 7.  2.  0.]
error positions = [25, 13]
cycle classes = [4]
g mod v = [ 1.  0.  0.  1.  0.]
CHECK
v = [0 0 0 1 1 1 0 1 1 1 1 0 0 1 0 0 0 1 1 1 0 1 1 1 1 1 0
0 1 0 0]
S = [2, 4, 11, 8, 19, 22]
lambdas = [7, 2]
lambda poly = [ 7.  2.  0.]
error positions = [25, 13]
cycle classes = [4, 6]
g mod v = [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.]
CHECK
v = [0 0 0 1 1 1 0 1 1 1 1 0 0 1 0 0 0 1 1 1 0 1 1 1 1 1 0
0 1 0 0]
S = [2, 4, 11, 8, 19, 22]
lambdas = [7, 2]
lambda poly = [ 7.  2.  0.]
error positions = [25, 13]
cycle classes = [4, 5]
g mod v = [ 1.  0.  1.  1.  0.  1.  1.  1.  1.  0.]
CHECK
v = [0 0 0 1 1 1 0 1 1 1 1 0 0 1 0 0 0 1 1 1 0 1 1 1 1 1 0
0 1 0 0]
```

Processing math: 100%

Видно что ни один код не может декодировать исходную последовательность. =(