Глава 5 (Сазанович Владислав M3339)

In [1]:
```
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 [2]:
```
# 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)
```

In [119]:
```
def polynomial_to_string(P):
    p = ''
    for i in np.arange(len(P), 0, -1):
        if P[-i] == 0:
            continue
        if i - 1 == 0:
            p = p + '{} + '.format(int(P[-i]))
            continue

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

    return p[:-3]
```

Задание 1

In [120]:
```
# Строит порождающую матрицу для цикличесткого кода с порождающим полиномом g
def get_G(n, k, g):
    G = []
    shift = 0
    for i in range(k):
        current = np.zeros(n)
        for j in range(1, len(g) + 1):
            current[(j - 1 + shift) % n] = g[-j]
        shift += 1
        G.append(current)
    return np.array(G)
```

In [121]:
```
# Строит проверочную матрицу для цикличесткого кода с проверочным полиномом h
def get_H(n, k, h):
    H = []
    shift = 0
    for i in range(n - k):
        current = np.zeros(n)
        for j in range(len(h)):
            current[(shift + j) % n] = h[j]
        shift += 1
        H.append(current)
    return np.array(H)
```

In [122]:
```
# Проверим на примере из учебника
G = get_G(7, 4, [1, 1, 0, 1])
H = get_H(7, 4, [1, 1, 1, 0, 1])
G.dot(H.T) % 2
```

Out[122]:
```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

In [123]:
```
# x^n - 1
def create_xn(n):
    xn = np.zeros(n + 1)
    xn[0] = 1
    xn[-1] = 1
    return xn
```

In [124]:
```
# Находение делителя полинома
def find_factor(p):
    for f in generate(len(p) - 1)[2:]:
        f = np.trim_zeros(f, 'f')
        if np.all(np.polydiv(p, f)[1] % 2 == 0): # p mod f == 0
            return f
    return None # cannot factorize, already minimal
```

In [712]:
```
# Факторизация полинома
def factorize(p, prt=False):
    if prt:
        print('factorizing: {}'.format(polynomial_to_string(p)))

    cur = np.array(p)
    factors = []

    while True:
        factor = find_factor(cur)
        if factor is None:
            factors.append(cur)
            break

        factors.append(factor)
        cur = np.polydiv(cur, factor)[0] % 2

    if prt:
        for factor in factors:
            print(polynomial_to_string(factor))
    return factors
```

In [714]:
```
# Проверка на примере из учебника
f = factorize(create_xn(7), True)
```

```
factorizing: x^7 + 1
x + 1
x^3 + x + 1
x^3 + x^2 + 1
```

In [715]:
```
# Вычисляет h = (x^n - 1) / g
def get_h(n, g):
    xn = create_xn(n) # x^n - 1

    # calc h = (x^n - 1) / g
    temp = np.polydiv(xn, g)
    assert np.all(temp[1] % 2 == 0) # проверка того что делится без остатка
    return temp[0] % 2
```

In [716]:
```
# Строит порождающую и проверочную матрицу для кода с порождающим полиномом g
def build_code(n, g):
    r = len(g) - 1
    k = n - r

    h = get_h(n, g)

    return get_G(n, k, g).astype(int), get_H(n, k, h).astype(int)
```

In [720]:
```
# Проверка на примере из учебника
G, H = build_code(7, [1, 1, 0, 1])
print('G = \n{}'.format(G))
print('H = \n{}'.format(H))
print('G * H.T = \n{}'.format(G.dot(H.T) % 2))
```

```
G =
[[1 0 1 1 0 0 0]
 [0 1 0 1 1 0 0]
 [0 0 1 0 1 1 0]
 [0 0 0 1 0 1 1]]
H =
[[1 1 1 0 1 0 0]
 [0 1 1 1 0 1 0]
 [0 0 1 1 1 0 1]]
G * H.T =
[[0 0 0]
 [0 0 0]
 [0 0 0]
 [0 0 0]]
```

Научимся считать минимальное расстояние кода. Переберм все кодовые слова. Посчитаем расстояние. Ограничения на $n <= 9$ позволяют это сделать за разумное время.

In [721]:
```
def get_d(H):
    n = len(H[0])
    d = n
    for word in generate(n)[1:]:
        if not np.all(np.dot(word, H.T) % 2 == 0): # w * H.T == 0 => слово кодов
            continue
        d = min(d, np.sum(word)) # обновляем d весом кодового слова

    return d
```

In [722]:
```
# Проверим на примере из учебника
get_d(np.array([
    [1, 1, 1, 0, 1, 0, 0],
    [0, 1, 1, 1, 0, 1, 0],
    [0, 0, 1, 1, 1, 0, 1]
]))
```

Out[722]: 3

Построение кодов с заданной длиной теперь запишется довольно просто. Нужно разложить $x^n - 1$ на множители. Тогда каждый его делитель будет порождать циклический код.

In [723]:
```
def build_codes_with_length(n):
    xn = create_xn(n)
    factors = factorize(xn)

    for g in factors: # каждый делитель будет порождать код
        G, H = build_code(n, g)
        d = get_d(H)

        print('\n\n---New code of length n = {}---'.format(n))
        print('Code: n = {}, k = {}, d = {}'.format(n, len(G), d))
        print('g(x) = {}'.format(polynomial_to_string(g)))
        print('G = \n{}'.format(G))
        print('H = \n{}'.format(H))


        # Строим дуальный код
        G, H = build_code(n, get_h(n, g))
        d = get_d(H)
        print('\nDual code: n = {}, k = {}, d = {}'.format(n, len(G), d))
        print('g(x) = {}'.format(polynomial_to_string(get_h(n, g))))
        print('G = \n{}'.format(G))
        print('H = \n{}'.format(H))
```

In [134]:
```
# Построим циклические коды длины три (все остальные коды приложил отдельным фай
build_codes_with_length(3)
```

```


---New code of length n = 3---
Code: n = 3, k = 2, d = 2
g(x) = x + 1
G =
[[1 1 0]
 [0 1 1]]
H =
[[1 1 1]]

Dual code: n = 3, k = 1, d = 3
g(x) = x^2 + x + 1
G =
[[1 1 1]]
H =
[[1 1 0]
 [0 1 1]]


---New code of length n = 3---
Code: n = 3, k = 1, d = 3
g(x) = x^2 + x + 1
G =
[[1 1 1]]
H =
[[1 1 0]
 [0 1 1]]

Dual code: n = 3, k = 2, d = 2
g(x) = x + 1
G =
[[1 1 0]
 [0 1 1]]
H =
[[1 1 1]]
```

In [135]:
```
for n in range(3,10):
    build_codes_with_length(n)
```

```
---New code of length n = 3---
Code: n = 3, k = 2, d = 2
g(x) = x + 1
G =
[[1 1 0]
 [0 1 1]]
H =
[[1 1 1]]

Dual code: n = 3, k = 1, d = 3
g(x) = x^2 + x + 1
G =
[[1 1 1]]
H =
[[1 1 0]
 [0 1 1]]
```

Задание 2

In [673]:
```
n = 8
m = 3
```

In [674]:
```
# Определить j : alphas[j] = p
def locate(p, alphas):
    for i in range(len(alphas)):
        if (len(p) == len(alphas[i])) and (np.all(np.equal(p, alphas[i]))):
            return i
```

In [675]:
```
# находим все простые многочлены стпени не выше n
def get_all_prime_poly(n):
    prime = []
    for p in generate(n + 1)[1:]:
        p = np.trim_zeros(p, 'f')
        if find_factor(p) is None: # p is prime
            prime.append(p)
            print(polynomial_to_string(p))
    return prime
```

In [724]:
```
# С помощью примитивного многочлена находим примитивный элемент и строим поле
def get_field(p, m, order=1):
    found = 0
    for alpha in generate(len(p) - 1)[1:]:
        alpha = np.trim_zeros(alpha)
        alphas, unique = get_all_pows_of_alpha(alpha, p, m)
        if unique == 2**m - 1: # alpha - примитивный элемент
            found += 1
            if (found == order):
#                 print(alpha)
                return alphas
    return None
```

In [725]:
```
primitive = get_all_prime_poly(4) # все примитивные многочлены для поля GF(8)
```

```
1
x
x + 1
x^2 + x + 1
x^3 + x + 1
x^3 + x^2 + 1
x^4 + x + 1
x^4 + x^3 + 1
x^4 + x^3 + x^2 + x + 1
```

In [726]:
```
# Подходят только простые многочлены 4-й степени
primitive = primitive[-3:]
```

In [732]:
```
# Получение полей по примитивным многочленам
def get_fields(primitives, m):
    fields = []
    for i in range(len(primitives)):
        if (i == 2):
            fields.append(get_field(primitives[i], m, order = 4))
        else:
            fields.append(get_field(primitives[i], m))
    return fields
```

In [728]:
```
fields = get_fields(primitive, m)
```

In [729]:
```
# Сложение элементов в поле
def field_add(x, y):
    l = np.array(x)
    r = np.array(y)
    if len(x) > len(y):
        l = y
        r = x

    l = np.append(np.zeros(len(r) - len(l)), l)
    return np.trim_zeros((l + r) % 2, 'f')
```

In [730]:
```
# Умножение элементов в поле
def field_mul(p, x, y):
    res = np.polymul(x, y) % 2
    return np.trim_zeros(np.polydiv(res, p)[1] % 2, 'f')
```

In [731]:
```
# Проверка на то что поля изоморфны
# Строим номерное соответствие
def check_add_mul_equal(p1, field1, p2, field2):
    assert len(field1) == len(field2)
    n = len(field1)

    for i in range(n):
        for j in range(n):
            add1 = field_add(field1[i], field1[j])
            add2 = field_add(field2[i], field2[j])

            mul1 = field_mul(p1, field1[i], field1[j])
            mul2 = field_mul(p2, field2[i], field2[j])

            # номера должны совпадать для однозначного соответствия
            if not (locate(add1, field1) == locate(add2, field2)):
                print('add', i, j)
                print(add1, add2)
                return False

            # номера должны совпадать для однозначного соответствия
            if not (locate(mul1, field1) == locate(mul2, field2)):
                print('mul', i, j)
                print(mul1, mul2)
                return False
    return True
```

In [696]:
```
# Проверим что поля изморфны
print(check_add_mul_equal(primitive[0], fields[0], primitive[1], fields[1]))

print(check_add_mul_equal(primitive[0], fields[0], primitive[2], fields[2]))

print(check_add_mul_equal(primitive[1], fields[1], primitive[2], fields[2]))
```

```
True
True
True
```

Задание 3

In [752]:
```
# Порядок элемента x^k == 1
# Над полем полиномов x = alpha^i => (alpha^i)^k == 1 <=> i * k == 0
def get_element_order(q, i):
    for k in range(1, q + 1):
        if (i * k) % q == 0:
            return k

def get_order(q):
    orders = ''
    for i in range(0, q):
        orders = orders + '{} - {}, '.format(i, get_element_order(q, i))
    return orders
```

In [753]:
```
# 1 - 2 означает что элемент 1 имеет порядок 2
for q in [2,3,4,5,7,8,9]:
    print('q:{} -> {}'.format(q, get_order(q)))
```

```
q:2 -> 0 - 1, 1 - 2,
q:3 -> 0 - 1, 1 - 3, 2 - 3,
q:4 -> 0 - 1, 1 - 4, 2 - 2, 3 - 4,
q:5 -> 0 - 1, 1 - 5, 2 - 5, 3 - 5, 4 - 5,
q:7 -> 0 - 1, 1 - 7, 2 - 7, 3 - 7, 4 - 7, 5 - 7, 6 - 7,
q:8 -> 0 - 1, 1 - 8, 2 - 4, 3 - 8, 4 - 2, 5 - 8, 6 - 4, 7 - 8,
q:9 -> 0 - 1, 1 - 9, 2 - 9, 3 - 3, 4 - 9, 5 - 9, 6 - 3, 7 - 9, 8 - 9,
```

Задание 4

In [185]:
```
p = np.array([1, 0, 0, 1, 1]) # x^4 + x + 1
m = 4
```

In [186]:
```
# Фунцкия возводящая x в степени 0 .. qm по модулю p
def get_all_pows_of_alpha(alpha, p, m, prt=False):
    unique = set() # уникальные полиномы
    alphas = []

    x = np.array([1]).astype(int) # x^1
    for i in range(0, 2**m - 1): # перебираем все степени x
        poly = (np.polydiv(x, p)[1] + 2) % 2 # берем остаток от деления по моду
        poly = np.trim_zeros(poly, 'f')
        alphas.append(poly)
        unique.add(tuple(poly))

        if prt:
            print(i, polynomial_to_string(poly))
        x = np.polymul(x, alpha)

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

    return alphas, len(unique)
```

In [187]:
```
alphas, unique = get_all_pows_of_alpha([1, 0], p, m, True) # возьмем x в качеств
```

```
0 1
1 x
2 x^2
3 x^3
4 x + 1
5 x^2 + x
6 x^3 + x^2
7 x^3 + x + 1
8 x^2 + 1
9 x^3 + x
10 x^2 + x + 1
11 x^3 + x^2 + x
12 x^3 + x^2 + x + 1
13 x^3 + x^2 + 1
14 x^3 + 1
Unique: 15, All: 15
```

Получилось 15 различных элементов => поле построено. Найдем обратные

In [188]:
```
def get_opposite(alphas):
    for i in range(len(alphas)):
        alpha1 = alphas[i]
        for j in range(len(alphas)):
            alpha2 = alphas[j]

            temp = np.polydiv(np.polymul(alpha1, alpha2), p)

            mod = temp[1] % 2
            mod = np.trim_zeros(mod, 'f')

            if len(mod) == 1: # остаток == 1
                print('Обратный к {} ({}): {} ({})'.format(i, polynomial_to_stri
```

In [189]:
```
get_opposite(alphas)
```

```
Обратный к 0 (1): 0 (1)
Обратный к 1 (x): 14 (x^3 + 1)
Обратный к 2 (x^2): 13 (x^3 + x^2 + 1)
Обратный к 3 (x^3): 12 (x^3 + x^2 + x + 1)
Обратный к 4 (x + 1): 11 (x^3 + x^2 + x)
Обратный к 5 (x^2 + x): 10 (x^2 + x + 1)
Обратный к 6 (x^3 + x^2): 9 (x^3 + x)
Обратный к 7 (x^3 + x + 1): 8 (x^2 + 1)
Обратный к 8 (x^2 + 1): 7 (x^3 + x + 1)
Обратный к 9 (x^3 + x): 6 (x^3 + x^2)
Обратный к 10 (x^2 + x + 1): 5 (x^2 + x)
Обратный к 11 (x^3 + x^2 + x): 4 (x + 1)
Обратный к 12 (x^3 + x^2 + x + 1): 3 (x^3)
Обратный к 13 (x^3 + x^2 + 1): 2 (x^2)
Обратный к 14 (x^3 + 1): 1 (x)
```

Задание 5

In [735]:
```
p = np.array([1, 1, 1, 1, 1]) # x^4 + x^3 + x^2 + x + 1
m = 4
```

In [736]:
```
# Найдем элементы поля
alphas, unique = get_all_pows_of_alpha([1,1], p, m, True) # возьмем в качестве г
```

```
0 1
1 x + 1
2 x^2 + 1
3 x^3 + x^2 + x + 1
4 x^3 + x^2 + x
5 x^3 + x^2 + 1
6 x^3
7 x^2 + x + 1
8 x^3 + 1
9 x^2
10 x^3 + x^2
11 x^3 + x + 1
12 x
13 x^2 + x
14 x^3 + x
Unique: 15, All: 15
```

In [737]:
```
# Найдем обратные
get_opposite(alphas)
```

```
Обратный к 0 (1): 0 (1)
Обратный к 1 (x + 1): 14 (x^3 + x)
Обратный к 2 (x^2 + 1): 13 (x^2 + x)
Обратный к 3 (x^3 + x^2 + x + 1): 12 (x)
Обратный к 4 (x^3 + x^2 + x): 11 (x^3 + x + 1)
Обратный к 5 (x^3 + x^2 + 1): 10 (x^3 + x^2)
Обратный к 6 (x^3): 9 (x^2)
Обратный к 7 (x^2 + x + 1): 8 (x^3 + 1)
Обратный к 8 (x^3 + 1): 7 (x^2 + x + 1)
Обратный к 9 (x^2): 6 (x^3)
Обратный к 10 (x^3 + x^2): 5 (x^3 + x^2 + 1)
Обратный к 11 (x^3 + x + 1): 4 (x^3 + x^2 + x)
Обратный к 12 (x): 3 (x^3 + x^2 + x + 1)
Обратный к 13 (x^2 + x): 2 (x^2 + 1)
Обратный к 14 (x^3 + x): 1 (x + 1)
```

Задание 6

In [411]:
```
G = get_G(7, 4, [1, 1, 0, 1]).astype(int)
```

In [412]:
```
# Приведем в систематический вид
G[1] = (G[1] - G[3]) % 2
G[0] = (G[0] - G[3]) % 2
G[0] = (G[0] - G[2]) % 2
```

In [413]:
```
print('G = \n{}'.format(G))
```

```
G =
[[1 0 0 0 1 0 1]
 [0 1 0 0 1 1 1]
 [0 0 1 0 1 1 0]
 [0 0 0 1 0 1 1]]
```

In [416]:
```
H = np.hstack((G[:, 4:].T, np.identity(3).astype(int)))
```

In [417]:
```
print('H = \n{}'.format(H))
```

```
H =
[[1 1 1 0 1 0 0]
 [0 1 1 1 0 1 0]
 [1 1 0 1 0 0 1]]
```

In [738]:
```
# Матрица соответствия синдромов - ошибкам.
# Такая память соответствует декодеру Меггита
def get_syndrom_error_map(H):
    n = H.shape[1]
    r = H.shape[0]
    k = n - r

    se_map = {}

    c = np.zeros(n)
    for e in generate(n):
        if np.sum(e) > 1:
            continue
        s = np.dot((c + e) % 2, H.T).astype(int) % 2

        has = False
        cur_s = np.array(s)
        for i in range(n):
            if tuple(cur_s) in se_map:
                has = True
                break
            temp = cur_s[0]
            for j in range(0, r - 1):
                cur_s[j] = cur_s[j + 1]
            cur_s[-1] = temp

        if not has:
            se_map[tuple(s)] = e
    return se_map
```

In [739]:
```
get_syndrom_error_map(H)
```

Out[739]:
```
{(0, 0, 0): array([0, 0, 0, 0, 0, 0, 0]),
 (0, 0, 1): array([0, 0, 0, 0, 0, 0, 1]),
 (1, 0, 1): array([0, 0, 0, 0, 1, 0, 0]),
 (1, 1, 1): array([0, 0, 1, 0, 0, 0, 0])}
```

Задание 7

In [740]:
```
# Построим код хемминга (15, 11)
# Факторизуем x^15 - 1
n = 15
k = 11
f = factorize(create_xn(n), True)
```

```
factorizing: x^15 + 1
x + 1
x^2 + x + 1
x^4 + x + 1
x^4 + x^3 + 1
x^4 + x^3 + x^2 + x + 1
```

In [741]:
```
# Возьмем полином x^4 + x^3 + 1 и проверим что расстояние = 3
H = get_H(n, k, get_h(n, [1, 1, 0, 0, 1])).astype(int)
get_d(H)
```

Out[741]: 3

In [742]:
```
g = np.array([1, 1, 0, 0, 1])
h = get_h(n, g)
print('g(x) = {}'.format(polynomial_to_string(g)))
print('h(x) = {}'.format(polynomial_to_string(h)))

G = get_G(n, k, g).astype(int)
print('G = \n{}'.format(G))
```

```
g(x) = x^4 + x^3 + 1
h(x) = x^11 + x^10 + x^9 + x^8 + x^6 + x^4 + x^3 + 1
G = 
[[1 0 0 1 1 0 0 0 0 0 0 0 0 0 0]
 [0 1 0 0 1 1 0 0 0 0 0 0 0 0 0]
 [0 0 1 0 0 1 1 0 0 0 0 0 0 0 0]
 [0 0 0 1 0 0 1 1 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 1 1 0 0 0 0 0 0]
 [0 0 0 0 0 1 0 0 1 1 0 0 0 0 0]
 [0 0 0 0 0 0 1 0 0 1 1 0 0 0 0]
 [0 0 0 0 0 0 0 1 0 0 1 1 0 0 0]
 [0 0 0 0 0 0 0 0 1 0 0 1 1 0 0]
 [0 0 0 0 0 0 0 0 0 1 0 0 1 1 0]
 [0 0 0 0 0 0 0 0 0 0 1 0 0 1 1]]
```

In [743]:
```
# Приведем G в систематический вид
for i in np.arange(len(G) - 1, 3, -1):
    G[i - 4] = (G[i - 4] - G[i]) % 2
    G[i - 3] = (G[i - 3] - G[i]) % 2
G[0] = (G[0] - G[3]) % 2

print('G = \n{}'.format(G))
```

```
G = 
[[1 0 0 0 0 0 0 0 0 0 0 1 0 0 1]
 [0 1 0 0 0 0 0 0 0 0 0 1 1 0 1]
 [0 0 1 0 0 0 0 0 0 0 0 1 1 1 1]
 [0 0 0 1 0 0 0 0 0 0 0 1 1 1 0]
 [0 0 0 0 1 0 0 0 0 0 0 0 1 1 1]
 [0 0 0 0 0 1 0 0 0 0 0 1 0 1 0]
 [0 0 0 0 0 0 1 0 0 0 0 0 1 0 1]
 [0 0 0 0 0 0 0 1 0 0 0 1 0 1 1]
 [0 0 0 0 0 0 0 0 1 0 0 1 1 0 0]
 [0 0 0 0 0 0 0 0 0 1 0 0 1 1 0]
 [0 0 0 0 0 0 0 0 0 0 1 0 0 1 1]]
```

In [744]:
```
H = np.hstack((G[:, k:].T, np.identity(n - k).astype(int)))
print('H = \n{}'.format(H))
```

```
H =
[[1 1 1 1 0 1 0 1 1 0 0 1 0 0 0]
 [0 1 1 1 1 0 1 0 1 1 0 0 1 0 0]
 [0 0 1 1 1 1 0 1 0 1 1 0 0 1 0]
 [1 1 1 0 1 0 1 1 0 0 1 0 0 0 1]]
```

Кодирование сообщения

In [745]:
```
m = np.array([1,0,1,1,1,1,0,1,0,0,1])
c = np.dot(m, G) % 2
```

In [746]:
```
# Внесем ошибку в 7-й бит
v = np.array(c)
v[7] = (v[7] + 1) % 2
```

In [747]:
```
print('c(x) = {}'.format(polynomial_to_string(c)))
print('v(x) = {}'.format(polynomial_to_string(v)))
```

```
c(x) = x^14 + x^12 + x^11 + x^10 + x^9 + x^7 + x^4 + x^3 + x^2 + 1
v(x) = x^14 + x^12 + x^11 + x^10 + x^9 + x^4 + x^3 + x^2 + 1
```

Декодирование сообщения

In [748]:
```
# Создадим соответствие синдромов и ошибок для декодера Меггита
se_map = get_syndrom_error_map(H)
for k in se_map.keys():
    print(k, se_map[k])
```

```
(0, 0, 0, 0) [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
(0, 0, 0, 1) [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
(0, 0, 1, 1) [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
(1, 0, 1, 1) [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
(0, 1, 0, 1) [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
(1, 1, 1, 1) [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
```

In [749]:
```
# Декодер Меггита.
# Считаем синдром, смотрим есть ли такой синдром в памяти декодера,
#     если есть то возвращаем сдвинутый вектор ошибок
#     если нет то умножаем входную последовательность на x и повторяем
def decode_meggit(v, H, se_map):
    n = H.shape[1]
    xn = create_xn(n)

    cur_v = np.array(v)
    for shift in range(n):
        # Вычислим синдром:
        s = np.dot(cur_v, H.T) % 2

        if (tuple(s) in se_map): # если синдром уже есть выводим ошибку сдвинуту
            e = np.array(se_map[tuple(s)])
            # сдвиг
            shifted_e = e
            for i in range(len(e)):
                shifted_e[i] = e[(i + shift) % len(e)]
            return shifted_e

        # умножение на x
        cur_v = np.polydiv(np.polymul(cur_v, [1, 0]) % 2, xn)[1] % 2
        cur_v = np.append(np.zeros(n - len(cur_v)).astype(int), cur_v)
```

In [750]:
```
e = decode_meggit(v, H, se_map)
```

In [751]:
```
# Проверка
np.dot((v + e), H.T) % 2
```

Out[751]:
```
array([0, 0, 0, 0])
```