S. M. Ratseev, On implementation of Reed–Solomon codes and decoding algorithms, *Uchenyye zapiski UlGU. Seriya "Matematika i informatsionnyye tekhnologii"*, 2023, Issue 1, 60–96

https://www.mathnet.ru/eng/ulsu164

Ссылка на статью:
// Ученые записки УлГУ. Сер. Математика и информационные технологии. 2023, № 1, с. 60–96.
Поступила: 09.01.2023
Окончательный вариант: 26.01.2023

УДК 519.7

О реализации кодов Рида—Соломона и алгоритмов декодирования

Рацеев С. М.

ratseevsm@mail.ru
УлГУ, Ульяновск, Россия

В работе приводится программная реализация кодов Рида—Соломона над полем $GF(2^8)$. Приводится кодирование кодов с помощью дискретного преобразования Фурье, декодирование кодов на основе алгоритма Сугиямы и алгоритма Гао. Работа носит учебно-методический характер и может помочь с программной реализацией кодеров и декодеров кодов Рида—Соломона.

Ключевые слова: *помехоустойчивые коды, коды Рида—Соломона, декодирование кода, алгоритм Сугиямы, алгоритм Гао*

Содержание

Введение

Пусть $\alpha = (\alpha_0, \alpha_1, \ldots, \alpha_{n-1})$, где α_i — различные элементы конечного поля $F = GF(q)$, $y = (y_0, y_1, \ldots, y_{n-1})$ — ненулевые (не обязательно различные) элементы из F. Тогда обобщенный код Рида—Соломона, обозначаемый $GRS_k(\alpha, y)$, состоит из всех кодовых векторов вида:

$$u = (y_0 b(\alpha_0), y_1 b(\alpha_1), \ldots, y_{n-1} b(\alpha_{n-1})),$$

где $b(x)$ — информационные многочлены над полем F степени не выше $k-1$. Кодовое расстояние кода $GRS_k(\alpha, y)$ равно $d = n - k + 1$. Если $n = q - 1$, вектор y состоит из единиц и $\alpha_i = \alpha^i$, $i = 0, 1, \ldots, n-1$, где α — примитивный элемент поля F, то в этом случае получаем код Рида—Соломона (РС), т. е. код РС состоит из всех кодовых векторов вида:

$$u = (b(1), b(\alpha), \ldots, b(\alpha^{n-1})). \tag{1}$$

Для декодирования кодов РС и кодов БЧХ хорошо известны следующие алгоритмы [1–6]: алгоритм на основе метода Сугиямы, алгоритм на основе метода Берлекэмпа—Месси, алгоритм Питерсона—Горенстейна—Цирлера, алгоритм Гао.

Работа носит учебно-методический характер. В ней приводится реализация классического алгоритма декодирования на основе метода Сугиямы и алгоритма декодирования Гао кода РС над полем $GF(2^8)$ на языке программирования Си. Цель этой работы, прежде всего, учебная. Во-первых, нелишним будет показать нужность и важность таблицы степеней образующего элемента поля и таблицы дискретных логарифмов по основанию образующего элемента. В этом случае некоторые вычисления в поле становятся более быстрыми. Во-вторых, алгоритмы декодирования Сугиямы и Гао с небольшими модификациями можно применять, в частности, для декодирования обобщенных кодов Рида—Соломона и для декодирования кодов Гоппы, при этом именно на основе кодов Гоппы строится перспективная постквантовая криптосистема Мак-Элиса [7]. Поэтому приведенная программная реализация с пояснениями может помочь в учебных целях при написании лабораторных работ, курсовых и дипломных работ, связанных с данной тематикой.

1. Вычислительная сложность алгоритмов декодирования кодов РС

Первые три из перечисленных выше алгоритма декодирования (на основе метода Сугиямы, на основе метода Берлекэмпа—Месси, алгоритм Питерсона—Горенстейна—Цирлера) относятся к классу алгоритмов синдромного декодирования и их можно разбить на четыре шага:

1) вычисление компонентов синдромного вектора на основе полученного вектора;

2) нахождение многочлена локаторов ошибок $\sigma(x)$ (который содержит информацию о позициях ошибок);

3) нахождение корней многочлена $\sigma(x)$, по которым определяются позиции ошибок;

4) нахождение значений ошибок.

Данные три алгоритма декодирования (в классическом представлении) отличаются только методами нахождения многочлена локаторов ошибок $\sigma(x)$ на 2-м шаге алгоритма. Оценим их вычислительную сложность.

Обозначим $r = n - k$ — число проверочных символов кода. Заметим, что $r = n - k = d - 1 = 2t$, где t — максимальное число гарантированно исправляемых ошибок.

Процедура вычисления компонентов синдромного вектора на 1-м шаге алгоритма имеет вычислительную сложность $O(rn)$. При вычислении синдромных компонент $S_1 = v(\alpha)$,..., $S_{2t}(\alpha^{2t})$, где v — принятый вектор для вычисления $S_i = v(\alpha^i)$, $i = 1, \dots, 2t$, с помощью схемы Горнера потребуется $n - 1$ операций умножения и столько же операций сложения. Поэтому всего получаем $2t(n - 1)$ операций умножения (и столько же операций сложения). Поэтому получаем $O(rn)$ операций на данном шаге.

Вычислительная сложность нахождения многочлена локаторов ошибок $\sigma(x)$ на 2-м шаге алгоритма с помощью алгоритма Берлекэмпа—Месси или с помощью эквивалентного ему обобщенного алгоритма Евклида (который используется в алгоритме Сугиямы) составляет $O(r^2)$ операций. В алгоритме Питерсона—Горенстейна—Цирлера вычислительная сложность составляет $O(t^3)$ операций, так как методом Гаусса решается система из не более t линейных уравнений с не более t переменными. Поэтому алгоритм Питерсона—Горенстейна—Цирлера полезен, прежде всего, в учебных целях либо имеет практическое применение для малых значений t. Очень часто именно с этого алгоритма начинается знакомство с методами декодирования кодов Боуза—Чоудхури—Хоквингема (более точно, алгоритм Питерсона применяется для двоичных кодов БЧХ, а алгоритм Горенстейна—Цирлера для общего случая, в частности, двоичного).

Нахождение корней многочлена $\sigma(x)$ на 3-м шаге алгоритма с помощью метода Ченя (последовательной подстановки ненулевых элементов поля F в многочлен $\sigma(x)$) имеет вычислительную сложность $O(tn)$ операций, так как $\deg \sigma(x) \leq t$, а при подстановке элементов поля в многочлен используется упомянутая выше схема Горнера. При этом число ненулевых элементов поля F равно $n = q - 1$.

Нахождение значений ошибок на 4-м шаге алгоритма проводится с помощью метода Форни. В данном случае для вычисления каждого значения ошибки находится значение многочлена значений ошибок $\omega(x)$, степень которого меньше числа t. Используя схему Горнера получим не более t операций умножения (и столько же операций сложения). Так же вычисляется произведение не более $t - 1$ скобок вида $(1 - X_j X_i^{-1})$. Поэтому общее число операций умножения для нахождения значения одной ошибки не превосходит числа $3t$. При этом максимальное число ошибок может достигать числа t. Поэтому метод Форни имеет сложность $O(r^2)$ операций.

Таким образом, при $r < n/2$ (наиболее практичный случай) самыми трудоемкими являются 1-й и 3-й шаги приведенного алгоритма. Итоговая вычислительная сложность классических алгоритмов декодирования на основе методов Сугиямы и Берлекэмпа—Месси составляет $O(rn)$ операций (иногда приводят более грубую оценку в виде $O(n^2)$).

Заметим, что если число проверочных символов r достаточное небольшое (т. е. на достаточные длинные кодовые сообщения приходится достаточно небольшое число ошибок), то алгоритмы синдромного декодирования со сложностью $O(rn)$ становятся достаточно практичными. При этом алгоритм Сугиямы хорошо приспособлен к эффективной аппаратной реализации (см., напр., [8]). В то же время алгоритм Берлекэмпа—Месси имеет меньшее число операций в конечном поле, нежели алгоритм Сугиямы [9] (хоть они и сопоставимы в плане вычислительной сложности), поэтому часто применяется в программных декодерах.

Алгоритм декодирования Гао относится к классу алгоритмов бессиндромного декодирования. В нем нет вычислений синдромных компонент, нет процедуры Ченя и метода нахождения значений ошибок Форни. Этот алгоритм состоит из трех шагов:

1) построение интерполяционного многочлена;

2) обобщенный алгоритм Евклида;

3) деление многочлена на многочлен.

Если выполнять шаг 1 данного алгоритма с помощью, например, схемы Горнера, а шаг 2 с помощью классического обобщенного алгоритма Евклида, то вычислительная сложность метода Гао будет оцениваться величиной $O(n^2)$ операций. Но можно поступить иначе. Первый шаг данного алгоритма может быть выполнен любым быстрым алгоритмом дискретного преобразования Фурье над конечным полем со сложностью $O(n\,(\log n)^2)$ операций. Для выполнения 2-го шага алгоритма можно использовать алгоритм Менка [10] со сложностью $O(n\,(\log n)^2)$ операций. Деление многочленов на 3-м шаге можно выполнить за $O(n\ \log n\ \log\log n)$ операций. Тогда итоговая сложность алгоритма Гао составит $O(n\,(\log n)^2)$ операций.

В работах [11, 12] приводятся алгоритмы декодирования кодов РС со сложностью $O(n\log r + r(\log r)^2)$. Такая вычислительная сложность является наилучшей на сегодняшний день.

Заметим, что кодирование информационного вектора и его последующее извлечение из кодового вектора с помощью дискретного преобразования Фурье, используя схему Горнера, имеет вычислительную сложность $O(kn)$ операций. Но если это делать любым быстрым алгоритмом дискретного преобразования Фурье над конечным полем, то вычислительная сложность составит $O(n\,(\log n)^2)$ операций.

Если рассмотренные алгоритмы реализовать над полем $GF(2^m)$ с использованием таблицы степеней и таблицы дискретных логарифмов, что и сделано в данной работе, то операция умножения элементов поля заменится на операцию сложения и обращение к элементам таблицы, что заметно влияет на скорость работы программы.

2. Описание алгоритма декодирования Сугиямы

Более подробную информацию о данном алгоритме с полным доказательством его корректности можно найти в [6].

Кодирование информационных векторов b будем осуществлять с помощью дискретного преобразования Фурье, а именно, на основе формулы (1). В этом случае извлечение информационного вектора b из кодового (после декодирования) примет такой вид:

$$(b_0, b_1, \ldots, b_{k-1}) = (-u(1), -u(\alpha^{-1}), \ldots, -u(\alpha^{-(k-1)})).$$

Ниже приводится алгоритм декодирования методом Сугиямы.

Алгоритм 1 (декодирование кода Рида—Соломона на основе алгоритма Сугиямы).

Вход: принятый вектор v.

Выход: исходный кодовый вектор u, если произошло не более $[(d-1)/2]$ ошибок.

1) Пусть $t = [(d-1)/2]$, где $d = n-k+1$ — кодовое расстояние кода РС. Находятся компоненты $S_1, S_2, \ldots, S_{2t}$ синдромного вектора: $S_i = v(\alpha^i)$, $i = 1, 2, \ldots, 2t$. Если они все равны нулю, то полагается, что ошибок нет и алгоритм завершается с возвращением кодового вектора $u = v$.

   На основе синдромных компонент составляется синдромный многочлен

$$S(x) = \sum_{i=1}^{2t} S_i x^{i-1}.$$

2) Пусть $r_{-1}(x) = x^{2t}$, $r_0(x) = S(x)$, $v_{-1}(x) = 0$, $v_0(x) = 1$. Производится последовательность вычислений обобщенного алгоритма Евклида:

$$r_{i-2}(x) = r_{i-1}(x)q_{i-1}(x) + r_i(x),$$
$$v_i(x) = v_{i-2}(x) - v_{i-1}(x)q_{i-1}(x), \quad i \geq 1,$$

   до тех пор, пока для некоторого $r_j(x)$ не будет выполнено условие:

$$\deg r_{j-1}(x) \geq t, \quad \deg r_j(x) \leq t-1.$$

   Тогда

$$\sigma(x) = \lambda v_j(x), \quad \omega(x) = \lambda r_j(x),$$

   где константа $\lambda \in GF(q)$ задается так, чтобы удовлетворялось условие $\sigma(0) = 1$.

   Пусть $s = \deg \sigma(x)$. Тогда вектор v содержит s ошибок.

3) Отыскиваются s корней многочлена $\sigma(x)$ последовательной подстановкой в него ненулевых элементов поля $GF(q)$. При этом локаторы ошибок — это величины, обратные корням многочлена $\sigma(x)$.

4) Находятся значения ошибок $Y_1, \ldots, Y_s$, например, с помощью метода Форни:

$$Y_i = \frac{X_i^{-1}\omega(X_i^{-1})}{\prod\limits_{\substack{1 \leq j \leq t, \\ j \neq i}} (1 - X_j X_i^{-1})}, \quad i = 1, \ldots, s.$$

   Наконец, у вектора v из символа с номером i_j, $X_j = \alpha^{i_j}$, вычитается значение Y_j, $j = 1, \ldots, s$. Тем самым получается вектор u.

После исправления ошибок и получения исходного кодового вектора u происходит извлечение исходного информационного вектора b.

3. Программная реализация кодов РС и алгоритма декодирования Сугиямы

Рассмотрим код РС над полем $GF(2^8)$, построенного на основе примитивного многочлена $p(x) = x^8 + x^4 + x^3 + x^2 + 1$. Многочлену $p(x)$ соответствует двоичный вектор 100011101, который в шестнадцатеричной форме имеет вид 11d. В программе параметры данного поля отражены в следующих константах:

```
#define M 8 // поле GF(2^M)
#define NMAX 255 // число 2^M - 1 - максимальное M-битное число
#define P 0x11d // многочлен p(x)=x^8+x^4+x^3+x^2+1 в шестнадцатеричном виде
#define ALPHA 2 // 00000010 - корень многочлена p(x)
```

Заметим, что константу P можно задать и более естественным образом:

```
const unsigned short P = (1 << 8) | (1 << 4) | (1 << 3) | (1 << 2) | 1;
```

В программе будет построена таблица степеней и таблица дискретных логарифмов, которые облегчают нахождение произведений, степеней, частных, обратных и т.д. Пусть $a, b \in GF(2^8)$, $a \neq 0$, $b \neq 0$, $n \in \mathbb{N}$. Тогда для нахождения, например, значений $c = a \cdot b$, $d = a^n$, a^{-1} в поле $GF(2^8)$, учитывая равенства

$$a \cdot b = g^{\log_g a \cdot b} = g^{\log_g a + \log_g b},$$

$$a^n = g^{\log_g a^n} = g^{n \cdot \log_g a},$$

$$a^{-1} = g^{2^8 - 1 - \log_g a},$$

достаточно записать

```
c = deg_alpha[(log_alpha[a] + log_alpha[b]) % NMAX];
d = deg_alpha[(n * log[a]) % NMAX];
deg_alpha[NMAX - log[a]];
```

Здесь deg_alpha — массив степеней образующего элемента α, log_alpha — массив дискретных логарифмов по основанию α.

Заметим, что процесс нахождения произведений элементов поля можно еще немного оптимизировать, затратив на это чуть больше памяти компьютера. Так как в массиве логарифмов log_alpha хранятся значения от 1 до NMAX, то в выражении

```
(log_alpha[a] + log_alpha[b]) % NMAX
```

можно избавиться от взятия остатка от деления, если массив степеней deg_alpha сделать в два раза больше и продублировать в нем значения с индексами от 1 до NMAX в соответствующие значения с индексами от NMAX+1 до 2*NMAX. Тогда произведение элементов a и b можно найти следующим образом:

```
c = deg_alpha[ log_alpha[a] + log_alpha[b] ];
```

Таким образом, в построенном ниже программном кодере и декодере совсем не будет делений (/ и %).

Длина кода РС над полем $GF(2^8)$ равна $n = 2^8 - 1 = 255$. Размерность кода k может быть произвольной ($1 \leq k \leq n$). Она зависит от того, какое максимальное число ошибок будет исправлять декодер в векторе из 255 символов над полем $GF(2^8)$. Если декодер будет исправлять до t ошибок, то $k = n - 2t$, так как для кодового расстояния кода РС выполнено равенство $d = n - k + 1$ (также полезно вспомнить критерий исправления до t ошибок). Зададим параметры нашего кода в виде следующих констант:

```
#define N 255 // длина кода
#define T 10 // максимальное число исправляемых ошибок
#define K 235 // размерность кода K=N-2T
#define D 21 // D = N - K + 1 - кодовое расстояние
```

Так как мы задали $t = 10$, то наш код может исправлять до 10 ошибок (т. е. искаженных символов поля $GF(2^8)$ в сообщении длины 255), поэтому размерность кода равна $n-2t = 235$. Это значит, что кодироваться будут информационные векторы b длины 235 в кодовые векторы u длины 255 над $GF(2^8)$.

Наша программа будет, в частности, состоять из следующих функций.

Функция

```
uint8 Product_alpha(uint8 a)
```

возвращает произведение элемента $a \in GF(2^8)$ и примитивного элемента $\alpha \in GF(2^8)$.

Функция

```
uint8 Product(uint8 a, uint8 b)
```

возвращает произведение элементов $a, b \in GF(2^8)$.

Функция

```
void Itoa(uint8 a, char *s)
```

переводит элемент поля a в строку s с изображением двоичного представления этого элемента. Это сделано для наглядности отображения на экране сообщений. Эта функция никак не влияет на алгоритмы кодирования и декодирования.

Функция

```
void Deg(POLYNOMIAL *a)
```

вычисляет степень многочлена $a(x)$. Сама структурная переменная POLYNOMIAL имеет вид

```
typedef struct POLYNOMIAL
{
    int deg; // степень многочлена
    uint8 array[N]; // коэффициенты многочлена
} POLYNOMIAL;
```

Она состоит из двух полей — степени многочлена и коэффициентов многочлена. Функция Deg() записывает степень многочлена в поле *deg* переменной a.

Функция

```
void Deg_n(POLYNOMIAL *a, int n)
```

вычисляет степень многочлена $a(x)$, если известно, что его степень ограничена числом n. В этом случае в отличие от функции Deg() может потребоваться меньшее число операций сравнения.

Функция

```
void Mult(const POLYNOMIAL *a, const POLYNOMIAL *b, POLYNOMIAL *c)
```

вычисляет произведение многочленов $a(x)$ и $b(x)$ над полем $GF(2^8)$ и результат записывает в многочлен $c(x)$, записывая при этом в поле *deg* структурной переменной c степень многочлена $c(x)$.

Функция

```
void Add(const POLYNOMIAL *a, const POLYNOMIAL *b, POLYNOMIAL *c)
```

вычисляет сумму многочленов $a(x)$ и $b(x)$ над полем $GF(2^8)$ и результат записывает в многочлен $c(x)$, записывая при этом в поле *deg* структурной переменной c степень многочлена $c(x)$.

Функция

```
void Mult_lambda(const POLYNOMIAL *a, uint8 lambda, POLYNOMIAL *b)
```

умножает многочлен $a(x)$ на константу λ и результат записывает в многочлен $b(x)$. Также в поле *deg* структурной переменной b записывается степень многочлена $b(x)$.

Функция

```
void Copy(POLYNOMIAL *a, const POLYNOMIAL *b)
```

копирует структурную переменную b в a. Здесь принципиально не используется операция присваивания, применимая для структур, так как копирование производится не всего массива коэффициентов, а только коэффициентов с номерами от нуля до степени многочлена $b(x)$.

Функция

```
void Init_zeros(POLYNOMIAL *a)
```

заполняет коэффициенты многочлена $a(x)$ нулями. При этом степень такого полинома определяется как -1.

Логическая функция

```
int Mod(const POLYNOMIAL *a, const POLYNOMIAL *b, POLYNOMIAL *q, POLYNOMIAL *r)
```

делит многочлен $a(x)$ на $b(x)$ с остатком. Целая часть от деления записывается в многочлен $q(x)$, остаток от деления — в $r(x)$. В структурные переменные q и r также записываются степени соответствующих многочленов. Если многочлен $b(x)$ нулевой, то функция возвращает ложное значение, в противном случае — истинное.

Функция

```
uint8 Value_of_polynomial(const uint8 *a, int n, int i)
```

вычисляет значение многочлена $a(x)$ степени n при $x = \alpha^i$ на основе схемы Горнера.

Функция

```
void Code(const uint8 *b, uint8 *u)
```

кодирует информационный вектор b с помощью дискретного преобразования Фурье и результат записывает в кодовый вектор u.

Функция

```
void Extract_information_vector(const uint8 *u, uint8 *b)
```

извлекает информационный вектор b из кодового вектора u.

Функция

```
void Search_roots(const POLYNOMIAL *sigma, uint8 *X)
```

ищет корни многочлена локаторов ошибок $\sigma(x)$ методом Ченя и записывает позиции ошибок в массив X.

Функция

```
void Serach_error_values(const POLYNOMIAL *omega, const uint8 *X, uint8 *Y, uint8 count)
```

вычисляет значения ошибок с помощью метода Форни и результат записывает в массив ошибок Y.

Логическая функция

```
int Syndrome_polynomial(const uint8 *v, POLYNOMIAL *s)
```

вычисляет синдромный многочлен $s(x)$ на основе принятого многочлена $v(x)$. Если все синдромные компоненты равны нулю, то возвращает ложь, иначе — истину.

Функция

```
int Generalized_Euclid_algorithm(const POLYNOMIAL *s, POLYNOMIAL *sigma,
                                       POLYNOMIAL *omega)
```

на основе синдромного многочлена $s(x)$ вычисляет многочлен локаторов ошибок $\sigma(x)$ и многочлен значений ошибок $\omega(x)$. В данной функции переменные r_0, r_1, q, r отвечают за соотношения $r_0 = r_1 q + r$, где q, r — соответственно частное и остаток от деления многочлена $r_0(x)$ на $r_1(x)$. После таких вычислений нужно переопределение значений переменных: $r_0(x) := r_1(x)$, $r_1(x) := r(x)$. Чтобы не перезаписывать многочлены целиком, переменные r_0, r_1, r будут переменными-указателями на многочлены. Сами многочлены будут храниться в переменных a, b, c. После очередного вычисления $r_0 = r_1 q + r$ достаточно будет переопределить только адреса многочленов. Аналогично поступим и с многочленами v_0, v_1, v во время вычислений вида $v = v_0 - v_1 q$.

Функция

```
int Decode_Sugiyama(uint8 *v)
```

декодирует принятый вектор v с помощью метода Сугиямы, исправляя в нем ошибки.

Ниже приводится программа кодирования и декодирования сообщений методом Сугиямы.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

// Параметры конечного поля
#define M 8 // поле GF(2^M)
#define NMAX 255 // число 2^M - 1 - максимальное M-битное число
#define P 0x11d // многочлен p(x)=x^8+x^4+x^3+x^2+1 в шестнадцатеричном виде
#define ALPHA 2 // 00000010 - корень многочлена p(x)

typedef unsigned char uint8;
typedef unsigned short uint16;

uint8 deg_alpha[2*NMAX + 1]; // степени элемента alpha
uint8 log_alpha[NMAX + 1];  // таблица дискретных логарифмов по основанию alpha

// Параметры кода
#define N 255 // длина кода
#define T 10 // максимальное число исправляемых ошибок
#define K 235 // размерность кода K=N-2T
#define D 21 // D = N - K + 1 - кодовое расстояние

// Структура, содержащая инфомацию о многочлене
typedef struct POLYNOMIAL
```

```
{
    int deg; // степень многочлена
    uint8 array[N]; // коэффициенты многочлена
} POLYNOMIAL;

// Произведение элементов a и alpha
uint8 Product_alpha(uint8 a)
{
    uint16 c = a;
    c <<= 1;
    if ((c >> M) & 1)
         c ^= P;
    return c;
}

// Произведение элементов поля a и b
uint8 Product(uint8 a, uint8 b)
{
    uint16 buf; // для промежуточных вычислений
    uint8 rez; // результат произведения
    buf = b;
    rez = 0;
    while(a)
    {
        if (a & 1)
            rez ^= buf;
        buf <<= 1;
        if ((buf >> M) & 1)
            buf ^= P;
        a >>= 1;
    }
    return rez;
}

// Вычисление степени многочлена a(x)
void Deg(POLYNOMIAL *a)
{
    int i;
    for(i = N-1; i >= 0 && !(a->array[i]); i--)
        ;
    a->deg = i;
```

```
}

// Вычисление степени многочлена a(x), если deg a(x) <= n
void Deg_n(POLYNOMIAL *a, int n)
{
    int i;
    for(i = n; i >= 0 && !(a->array[i]); i--)
        ;
    a->deg = i;
}

// Заполнение нулями
void Init_zeros(POLYNOMIAL *a)
{
    int i;
    for(i = 0; i < N; i++)
        a->array[i] = 0;
    a->deg = -1;
}

// Заполнение нулями коэффициенты с номерами от 0 до n
void Init_zeros_n(POLYNOMIAL *a, int n)
{
    int i;
    for(i = 0; i <= n; i++)
        a->array[i] = 0;
    a->deg = -1;
}

// Произведение многочленов a(x) и b(x) над GF(2^M)
// Результат записывается в многочлен c(x)
void Mult(const POLYNOMIAL *a, const POLYNOMIAL *b, POLYNOMIAL *c)
{
    int i, j;
    Init_zeros_n(c, a->deg + b->deg);
    for(i = 0; i <= a->deg; i++)
        for(j = 0; j <= b->deg; j++)
             if (a->array[i] && b->array[j])
                c->array[i+j] ^=  deg_alpha[ log_alpha[a->array[i]] +
                                             log_alpha[b->array[j]] ];
    c->deg = a->deg + b->deg;
```

```
}

// Сложение многочленов a(x) и b(x)
// Результат записывается в многочлен c(x)
void Add(const POLYNOMIAL *a, const POLYNOMIAL *b, POLYNOMIAL *c)
{
    int i;
    if (a->deg < b->deg)
    {
        for(i = 0; i <= a->deg; i++)
            c->array[i] = a->array[i] ^ b->array[i];
        for(i = a->deg + 1; i <= b->deg; i++)
            c->array[i] = b->array[i];
        c->deg = b->deg;
    }
    else
    {
        for(i = 0; i <= b->deg; i++)
            c->array[i] = a->array[i] ^ b->array[i];
        for(i = b->deg + 1; i <= a->deg; i++)
            c->array[i] = a->array[i];
        c->deg = a->deg;
    }
}

// Умножение многочлена a(x) на ненулевую константу lambda
// Результат записывается в многочлен b(x)
void Mult_lambda(const POLYNOMIAL *a, uint8 lambda, POLYNOMIAL *b)
{
    int i;
    for(i = 0; i <= a->deg; i++)
        if(a->array[i])
            b->array[i] = deg_alpha[ log_alpha[a->array[i]] +
                                     log_alpha[lambda] ];
        else b->array[i] = 0;
    b->deg = a->deg;
}

// Копирование b в a
void Copy(POLYNOMIAL *a, const POLYNOMIAL *b)
{
```

```
    int i;
    for(i = 0; i <= b->deg; i++)
        a->array[i] = b->array[i];
    a->deg = b->deg;
}

// Логическая функция деления с остатком многочлена a(x) на многочлен b(x).
// Остаток записывается в *r, целая часть - в *q.
int Mod(const POLYNOMIAL *a, const POLYNOMIAL *b, POLYNOMIAL *q, POLYNOMIAL *r)
{
    int i;
    uint8 c;
    if (b->deg < 0)
        return 0;
    Copy(r, a);
    Init_zeros_n(q, a->deg);
    while(r->deg >= b->deg)
    {
        c = deg_alpha[ NMAX + log_alpha[r->array[r->deg]] -
                              log_alpha[b->array[b->deg]] ];
        q->array[r->deg - b->deg] = c;
        for(i = b->deg; i >= 0; i--)
            if (b->array[i])
                r->array[i + r->deg - b->deg] ^=
                    deg_alpha[ log_alpha[b->array[i]] + log_alpha[c] ];
        Deg_n(r, r->deg - 1);
    }
    Deg_n(q, a->deg);
    return 1;
}

// Вычисление значения многочлена a(x) степени n при x = alpha^i,
// где 0 <= i <= NMAX, с помощью схемы Горнера
uint8 Value_of_polynomial(const uint8 *a, int n, int i)
{
    uint8 y;
    int j;
    if (n < 0)
        return 0;
    y = a[n];
    for(j = n - 1; j >= 0; j--)
```

```
        y = (y ? (a[j] ^ deg_alpha[ log_alpha[y] + i ]) : a[j]);
    return y;
}

// Кодирование информационного вектора b с помощью дискретного преобразования Фурье
void Code(const uint8 *b, uint8 *u)
{
    int i;
    for(i = 0; i < N; i++)
        u[i] = Value_of_polynomial(b, K - 1, i);
}

// Извлечение информационного вектора из кодового
void Extract_information_vector(const uint8 *u, uint8 *b)
{
    int i;
    for(i = 0; i < K; i++)
        b[i] = Value_of_polynomial(u, N - 1, NMAX - i);
}

// Поиск корней многочлена sigma(x) методом Ченя
void Search_roots(const POLYNOMIAL *sigma, uint8 *X)
{
    int i, k = 0;
    if (Value_of_polynomial(sigma->array, sigma->deg, 0) == 0)
        X[k++] = 0; // сохраняем позицию ошибки
    for(i = 1; i < NMAX && k < sigma->deg; i++)
        if (Value_of_polynomial(sigma->array, sigma->deg, i) == 0)
            X[k++] = NMAX - i; // сохраняем позицию ошибки
}

// Нахождение значений ошибок методом Форни.
// count - число ошибок, в X - позиции ошибок, значения ошибок записываются в Y
void Search_error_values(const POLYNOMIAL *omega, const uint8 *X, uint8 *Y, uint8 count)
{
    uint8 prod, b, y;
    int i, j, k;

    for(i = k = 0; i < count; i++)
    {
        prod = 1;
```

```
        for(j = 0; j < count; j++)
        {
            if (j != i)
            {
                // b = (1 - X_jX_i^{-1})
                b = 1 ^ deg_alpha[ X[j] + NMAX - X[i] ];
                // prod - произведение таких скобок
                prod = deg_alpha[ log_alpha[prod] + log_alpha[b] ];
            }
        }
        y = Value_of_polynomial(omega->array, omega->deg, NMAX - X[i]);
        y = deg_alpha[ NMAX - X[i] + log_alpha[y] ];
        y = deg_alpha[ log_alpha[y] + NMAX - log_alpha[prod] ];
        Y[k++] = y;
    }
}

// Нахождение синдромного многочлена s(x) на основе принятого многочлена v(x)
int Syndrome_polynomial(const uint8 *v, POLYNOMIAL *s)
{
    int i, flag = 0;
    // Вычисляем компоненты синдромного вектора s
    for(i = 0; i < 2*T; i++)
        if (s->array[i] = Value_of_polynomial(v, N - 1, i + 1))
            flag = 1; // хотя бы одна синдромная компонента не равна нулю
    Deg_n(s, 2*T-1); // вычисляем степень многочлена s(x)
    return flag;
}

// Вычисление многочлена локаторов ошибок и многочлена значений ошибок
// на основе обобщенного алгоритма Евклида
int Generalized_Euclid_algorithm(const POLYNOMIAL *s, POLYNOMIAL *sigma,
                                                      POLYNOMIAL *omega)
{
    // Многочлены для обобщенного алгоритма Евклида.
    // Указатели, чтобы не копировать многочлены
    POLYNOMIAL *r0, *r1, *r, *v0, *v1, *v, *buf;
    // фактически в этих многочлена будут храниться многочлены
    POLYNOMIAL *q, *a, *b, *c, *d, *e, *f;
    uint8 lambda;
    // Выделяем место для массивов q, a, b, c, d, e, f
```

```
    // при этом функция calloc заполняет все значения нулями
    q = (POLYNOMIAL *)calloc(1, sizeof(*q));
    a = (POLYNOMIAL *)calloc(1, sizeof(*a));
    b = (POLYNOMIAL *)calloc(1, sizeof(*b));
    c = (POLYNOMIAL *)calloc(1, sizeof(*c));
    d = (POLYNOMIAL *)calloc(1, sizeof(*d));
    e = (POLYNOMIAL *)calloc(1, sizeof(*e));
    f = (POLYNOMIAL *)calloc(1, sizeof(*f));
    if(q == NULL || a ==NULL || b == NULL || c == NULL || d == NULL ||
       e == NULL || f == NULL)
        return 0;
    a->deg = -1; b->deg = -1; c->deg = -1;
    d->deg = -1; e->deg = -1; f->deg = -1;
    r0 = a; r1 = b; r = c;
    v0 = d; v1 = e; v = f;
    // Применяем неполный обобщенный алгоритм Евклида (2-й шаг алгоритма 1)
    r0->array[2*T] = 1; r0->deg = 2*T;
    Copy(r1, s);
    v1->array[0] = 1; v1->deg = 0;
    while(r1->deg >= T)
    {
        Mod(r0, r1, q, r);
        buf = r0; r0 = r1; r1 = r; r = buf;
        Mult(q, v1, r);
        Add(v0, r, v);
        buf = v0; v0 = v1; v1 = v; v = buf;
    }
    // Находим многочлены sigma(x) и omega(x)
    lambda = deg_alpha[NMAX - log_alpha[v1->array[0]]];
    Mult_lambda(v1, lambda, sigma);
    Mult_lambda(r1, lambda, omega);
    free(q); free(a); free(b); free(c); free(d); free(e); free(f);
    return 1;
}

// Декодирование искаженого вектора с помощью алгоритма Сугиямы
int Decode_Sugiyama(uint8 *v)
{
    POLYNOMIAL s; // синдромный многочлен
    POLYNOMIAL sigma, omega; // многочлены локаторов и значений ошибок
    uint8 X[T]; // позиции ошибок
```

```
    uint8 Y[T]; // значения ошибок
    int i;

    // Вычисляем синдромный многочлоен s (1-й шаг алгоритма 1)
    // Если ошибок не было, завершаем процесс декодирования
    if (!Syndrome_polynomial(v, &s))
        return 0;
    // Вычисление многочлена локаторов ошибок и многочлена значений ошибок
    // (2-й шаг алгоритма 1)
    if (!Generalized_Euclid_algorithm(&s, &sigma, &omega))
        return -1;
    // Поиск корней многочллена sigma(x) методом Ченя (3-й шаг алгоритма 1)
    Search_roots(&sigma, X);
    // Вычисление значений ошибок методом Форни (4-й шаг алгоритма 1)
    Search_error_values(&omega, X, Y, sigma.deg);
    // Исправление ошибок
    for(i = 0; i < sigma.deg; i++)
        v[X[i]] ^= Y[i];
    return 1;
}

// Двоичное представление целого числа a записывается в строку s
void Itoa(uint8 a, char *s)
{
    int i, j, bit;
    for (i = M - 1, j = 0; i >= 0; i--, j++)
    {
       // выделяем i-й справа бит
       bit = (a >> i) & 1;
       s[j] = '0' + bit;
    }
    s[j] = '\0';
}

// Вывод сообщения a длины n на экран в двоичном виде
void Print_array(const uint8 *a, int n)
{
    int i;
    char s[N+1]; // для перевода двоичной записи в строку
    for(i = 0; i < n; i++)
    {
```

```
        Itoa(a[i], s);
        printf(s);
    }
    puts("\n");
}

int main()
{
    int i;
    uint8 b[K], // информационный вектор
          u[N], // кодовый вектор
          v[N], // принятый вектор
          e[N]; // вектор ошибок

    // Вычисление степеней образующего элемента alpha и
    // и построение таблицы дискретных логарифмов
    deg_alpha[0] = 1;
    for (i = 1; i <= NMAX; i++)
    {
        deg_alpha[i] = Product_alpha(deg_alpha[i-1]);
        deg_alpha[NMAX + i] = deg_alpha[i]; // чтобы не брать "% NMAX"
        log_alpha[deg_alpha[i]] = i;
    }

    srand(time(NULL));
    // Генерируем случайный информационный вектор
    for(i = 0; i < K; i++)
        b[i] = rand() % (NMAX + 1);
    // Выводим информационный вектор b
    puts("b = "); Print_array(b, K);

    // Кодируем информационный вектор b и получаем кодовый вектор u
    Code(b, u);
    // Выводим кодовый вектор u
    puts("u = "); Print_array(u, N);

    // Генерируем вектор ошибок веса <= T
    for(i = 0; i < N; i++)
        e[i] = 0;
    for(i = 0; i < T; i++)
        e[rand() % N] = rand() % (NMAX + 1);
```

```
    // Выводим вектор ошибок e
    puts("e = "); Print_array(e, N);

    // Искажаем кодовый вектор u вектором ошибок e
    for(i = 0; i < N; i++)
        v[i] = u[i] ^ e[i];
    // Выводим искаженный вектор v
    puts("v = "); Print_array(v, N);

    // Декодируем искаженный вектор v
    if (Decode_Sugiyama(v) == -1)
    {
        puts("no enough memory");
        return 1;
    }

    // Извлекаем информационный вектор из кодового
    Extract_information_vector(v, b);
    // Выводим исходный информационный вектор b
    puts("b = "); Print_array(b, K);

    return 0;
}
```

Для наглядности представим результат работы программы. В одном из запусков программы получен следующий результат:
$b =$

1001001110010011110000001100110110000111100011101011001110001100010000100001100110000000000000000001111011110011010110001111101100
1010101001011010001111011010111100110110011110101101011000111001110110010000101111110010101111000000100010110100000000111000001101
0110101010111011001111000100110010010100110000111101000011000111101110001101010101011101000111000000110000100001111000000100111110
101001100011011100111010111110101000111011001010111001000000110110001011010110011110111100100101001101110001110001110100010011010
01110011100011111111101011000001111001000100111101101110111001101001010000000001010101001010101111111111010010111100001010100111001
00100110110000110110100111101111001010001001000111001001101101110110001101101100110101101111010110110111100001001001001111110100
0101010100010110110101001000100011010010001101011111010011110001011110111100101011000111000000011111010011101101011001111110000100
11010101000000111101100111101011000101101010110101110110011010101011001100011111111100001000000001100010101110100010111011111100011
10111001010110001100001010100010110110110001000001111111111110001011010001011011010010001011110000000000000110110111011101001000
010110000000101011111110111101101100010010101010101101110011111110010100011001110001010101111011010111101010011111110000101110
01000111010111101010010100011010000010111011100011100100101101110110110000110100110100010000100101100111101010110111001101001101
01011010110101010011010010011000000101100000101110010001101111000011010101010001000101100110101110101110100100111000101010011111
10100110001011101100111001110001000100110110100100011011111001110101101011101001001101100100110001000111000100100010110101011011
1000000101101110001110110000010001001010000000100111000100001001101010000101000000010011100101010100111001000110001000100111111100
01111111110100111110111001000100011111101000110011100000001010111110010110101010010111100

$u =$

0000001010101110101000011111111111000010010001101000110001110011011000110001110110010010100111111110110001011111110111111111110100001
10001000100110110010010000010000000110001111111101010110011100001011110101110010010111000100010010100100110100010011001111000100
10010101001101101010001001011001100111110010100001110000001101111001101011011000010010111101000001000101100001110000000010010101
1111111111100001111111011010110010111100110001000000111111010100011010000000101100011100111011111100111101001011011001110100110100
1110110000011111101111100110100010010111110101011111111110010001011101000111100000000101011100001001111111000000011000001011010110
00100110101101111001001011100100110101001110011001010011011000000111111001010111011000100101010110101100101100100001001101000011
01100111001011001111100001010010110100011101011001100110101010010000000011100010010110110010111001011111101110110000111110100100
10001111100000101011100001101010000010111001100101101100111000011100000011110010100110001101010110100110001010001010000101100010
1000011110011000100110111000101111111111100010011011000000100000111011000100100010100101001110011011000101110010111101100110100101
1010100110010000010010010010000101011000110111101001001100100111111111101000000010011011010101011100110001011111010101110101100000001
1011101011101011000101111100000001000000000100010100111111111110001111100110110000001111011110111000011011000001111101000100000100000
1001001011100010011111101000110111100111101010010101101100010010000011101100011101101011000101010011001000011000111111101110000 11
10010101111100111110101100100101100010011101110000101111101010001100110101101110001011110111001000011011000010010100100010011100
010010011111110001101010101111101100011111100000001001101110101110000110001100111100010101010100010110110000111011111110110111110110
01100000011010001100100000000011110100010101110101111101110000100001000110111111101111110100100100011011000000100001001110001000101
0010000000110100011001001001110011011101111001110100011010001100010100111100010001010010010000011000000110100111000011111

$e =$

00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000101100000000000000110000100000000000
00000000000000001111110100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000011110010000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000011011000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000110010000000000000000000000000000000000000000000
00000000000000000001001100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000011101010000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000011000010000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000010001001000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

$v = u \oplus e =$

0000001010101110101000011111111111000010010001101000110001110011011000110001110110010010100111111110110001011111110111111111110100001
1000100010011011001001000001000000011000111111110101011001110000101111010111001001011100010011110100100110100011111000111000100
1001010100110110010111110101100110011111001010000111000000110111100110101101100001001011110100000100010110000111000000000010010101
11111111111000011111110110101100101111001100010000001111110101101011010000000101100011100111011111100111101001011011001110100110100
111011000001111110111110011010001001011111010101111001000010001011101000111100000000101011100001001111111000000011000001011010110
00100110101101111001001011100100110101001110011001010011011000000111111001010111011000100101010110101100101100100001001101000011
01100111001011001111100001010010110100011101011001100110101010010000000011100010010110110010111001011111101110110000111110100100
10001111100000101011100001101010000010111001100101101100111000011100000011110010100110001101010110100110001010001010000101100010
1000011110011000100110111000101111111111100010011011000000100000111011000100100010010111001110011011000101110010111101100110100101
10101001100100000101101000100010101100011011110100100110010011111111101000000010011011010101011100110001011111010101110101100000001
101110101110101100010111110000000100000000010001010011111111111000111110011011000000110011010011100001101100000111110100010000010000 00
1001001011100010011111101000110111100111101010010101101100010010000011101100011101101011000101010011001000011000111111101110000 11
10010101111100111110101100100101100010011101110000101111101010001100110101101110010011100111001000011011000010010100100010011100
010010011111110001101010101111101100011111100000001001101110101110000110001100111100010101010100010010100010111011111110110111110110
01100000011010001100100000000011110100010101110101111101110000100001000110111111101111110100100100011011000000100001001110001000101
0010000000110100011001001001110011011101111001110100011010001100010100111100010001010010010000011000000110100111000011111

Сначала сгенерирован случайным образом информационный вектор b. Кодовый вектор u получен на основе вектора b с помощью дискретного преобразования Фурье. Далее сгенерирован вектор ошибок e, содержащий до 10 ошибок над $GF(2^8)$. После искажения вектора u на приемном конце получен вектор v. После работы декодера на основе метода Сугиямы получен вектор u, из которого извлечен исходный информационный вектор b.

Еще раз подчеркнем, что максимальное число гарантированно исправляемых ошибок для кода РС составляет $t = (n-k)/2$, поэтому $k = n - 2t$. Это означает, что если на вектор длины n символов (над полем $GF(q)$) приходится «достаточно» немного ошибок, то длины информационных и кодовых векторов будут отличаться не очень сильно при «достаточно» больших n.

4. Описание и реализация алгоритма Берлекэмпа—Месси

Более подробно про алгоритм Берлекэмпа—Месси можно посмотреть, например, в [2, 6]. Сам алгоритм имеет такой вид.

Алгоритм 2 (алгоритм Берлекэмпа—Месси).

Вход: последовательность $a_1, \ldots, a_n$ над некоторым полем.

Выход: LFSR $(L, f(x))$ минимальной длины L, для которого

$$-a_j = \sum_{i=1}^{L} f_i a_{j-i}, \quad j = L+1,\, L+2, \ldots, n.$$

1. Определить $r := 0$, $f(x) := 1$, $b(x) := 1$, $L := 0$.
2. Цикл $r := 1, \ldots, n$:
   - 2.1. Определить $\Delta := a_r + \sum\limits_{i=1}^{L} f_i a_{r-i}$.
   - 2.2. Если $\Delta = 0$, то $b(x) := x \cdot b(x)$.
   - 2.3. Если $\Delta \neq 0$:
     - 2.3.1. Если $2L < r$:
       $buf(x) := f(x) - \Delta \cdot x \cdot b(x)$,
       $b(x) := \Delta^{-1} \cdot f(x)$,
       $f(x) := buf(x)$,
       $L := r - L$.
     - 2.3.2. Иначе (т. е. выполнено $2L \geq r$):
       $f(x) := f(x) - \Delta \cdot x \cdot b(x)$,
       $b(x) := x \cdot b(x)$.

Заметим, что в алгоритме 2 инструкции вида $b(x) := x \cdot b(x)$ с практической точки зрения нужно реализовывать аккуратно, чтобы не делать лишних вычислений, поэтому не спешить с умножением x на $b(x)$, что, по сути, является сдвигом коэффициентов на одну позицию в сторону старших разрядов. В этом случае можно ввести переменную d, которая бы отвечала за вычисления вида $f(x) - \Delta \cdot x^d \cdot b(x)$.

Итак, добавим в предыдущую программу следующие две функции.

Функция

```
void Mult_mod(const POLYNOMIAL *a, const POLYNOMIAL *b, POLYNOMIAL *c, int n)
```

вычисляет произведение многочленов $a(x)$ и $b(x)$ над $GF(2^m)$ по модулю x^n. Эта функция нужна, чтобы не умножать с помощью рассмотренной ранее функции Mult, а потом находить остаток от деления с помощью функции Mod, а выполнить умножение более оптимальным образом.

Функция

```
void Algorithm_BM(POLYNOMIAL *s, POLYNOMIAL *sigma, POLYNOMIAL *omega)
```

вычисляет многочлен локаторов ошибок и многочлен значений ошибок на основе алгоритма Берлекэмпа—Месси. В данной функции также присутствуют переменные-многочлены, как и в функции Generalized_Euclid_algorithm (о чем говорилось ранее), а также переменные-указатели на многочлены, чтобы не перезаписывать многочлены целиком.

Сами функции имеют следующий вид:

```
// Произведение многочленов a(x) и b(x) над GF(2^M) по модулю x^n
// Результат записывается в многочлен c(x)
void Mult_mod(const POLYNOMIAL *a, const POLYNOMIAL *b, POLYNOMIAL *c, int n)
{
    int i, j;
    Init_zeros_n(c, n-1);
    for(i = 0; i <= a->deg; i++)
        for(j = 0; j <= b->deg && i + j < n; j++)
             if (a->array[i] && b->array[j])
                c->array[i+j] ^=  deg_alpha[ log_alpha[a->array[i]] +
                                             log_alpha[b->array[j]] ];
    Deg_n(c, n - 1);
}

// Вычисление многочлена локаторов ошибок и многочлена значений ошибок
// на основе алгоритма Берлекэмпа-Месси
void Algorithm_BM(POLYNOMIAL *s, POLYNOMIAL *sigma, POLYNOMIAL *omega)
{
    POLYNOMIAL c = {0, {1}}, d = {0, {1}}, e = {-1, {0}}, *buf, *f, *b, *buf2;
    int i, j, L = 0, deg_x = 0;
    uint8 delta;
    f = &c; b = &d; buf = &e;
    for(i = s->deg + 1; i < 2*T; i++)
        s->array[i] = 0;
    for(i = 0; i < 2*T; i++)
    {
        delta = s->array[i];
        for(j = 1; j <= f->deg; j++)
            if (f->array[j] && s->array[i - j])
                delta ^= deg_alpha[ log_alpha[f->array[j]] +
                                    log_alpha[s->array[i - j]] ];
        if (delta == 0)
        {
            deg_x++;
            (b->deg)++;
```

```
        }
        else
        {
            if (2*L < i + 1)
            {
                deg_x++;
                (b->deg)++;
                for(j = 0; j < deg_x; j++)
                    buf->array[j] = f->array[j];
                for(j = deg_x; j <= f->deg; j++)
                {
                    buf->array[j] = f->array[j];
                    if (b->array[j-deg_x])
                          buf->array[j] ^= deg_alpha[log_alpha[delta] +
                                                     log_alpha[b->array[j-deg_x]]];
                }
                buf->deg = f->deg;
                if (b->deg > f->deg)
                {
                    for(j = f->deg + 1; j <= b->deg; j++)
                        if (b->array[j-deg_x])
                              buf->array[j] ^= deg_alpha[log_alpha[delta] +
                                                         log_alpha[b->array[j-deg_x]]];
                    buf->deg = b->deg;
                }
                Mult_lambda(f, deg_alpha[NMAX - log_alpha[delta]], b);
                buf2 = f; f = buf; buf = buf2;
                L = i + 1 - L;
                deg_x = 0;
            }
            else
            {
                deg_x++;
                (b->deg)++;
                for(j = deg_x; j <= f->deg; j++)
                    if (b->array[j-deg_x])
                        f->array[j] ^= deg_alpha[log_alpha[delta] +
                                                 log_alpha[b->array[j-deg_x]]];
                if (b->deg > f->deg)
                {
                    for(j = f->deg + 1; j <= b->deg; j++)
```

```
                f->array[j] = b->array[j-deg_x];
            f->deg = b->deg;
        }
      }
    }
  }
  Copy(sigma, f);
  Mult_mod(sigma, s, omega, 2*T);
}
```

Поместив эти функции в предыдущую программу, можно в функции Decode_Sugiyama() заменить вызов функции Generalized_Euclid_algorithm() на Algorithm_BM().

5. Описание алгоритма декодирования Гао

Пусть код РС $[n, k, d = n - k + 1]$ над полем $F = GF(q)$ имеет длину $n = q - 1$, α — примитивный элемент поля F. Принятый вектор v представим многочленом

$$v(x) = \sum_{i=0}^{n-1} v_i x^i = u(x) + e(x) = \sum_{i=0}^{n-1} u_i x^i + \sum_{i=0}^{n-1} e_i x^i,$$

причем вес t многочлена ошибок $e(x)$ не превосходит $[(d-1)/2]$, а кодовый вектор u получен с помощью кодирования информационного многочлена $b(x) = b_0 + b_1 x + \ldots + b_{k-1}x^{k-1}$ рассмотренным ранее образом:

$$u = (u_0, u_1, \ldots, u_{n-1}) = (b(1), b(\alpha), \ldots, b(\alpha^{n-1})).$$

Пусть $X_1 = \alpha^{i_1}, \ldots, X_t = \alpha^{i_t}$ — локаторы ошибок. В данном алгоритме многочлен локаторов ошибок запишем в виде

$$\sigma(x) = (x - X_1) \ldots (x - X_t).$$

Если ошибок не было, то будем полагать, что $\sigma(x) = 1$.

Заметим, что если $v_i = u_i$, то $v_i = b(\alpha^i)$. Если $v_i \neq u_i$, то на позиции i произошла ошибка, поэтому $\sigma(\alpha^i) = 0$. Из этого следует, что

$$\sigma(\alpha^i)v_i = \sigma(\alpha^i)b(\alpha^i), \quad i = 0, 1, \ldots, n-1.$$

Обозначим $p(x) = \sigma(x)b(x)$. Тогда

$$\sigma(\alpha^i)v_i = p(\alpha^i), \quad i = 0, 1, \ldots, n-1.$$

Построим интерполяционный многочлен Лагранжа $f(x)$ степени не выше $n-1$, проходящий через точки $(1, v_0)$, (α, v_1),..., (α^{n-1}, v_{n-1}):

$$f(\alpha^i) = v_i, \quad i = 0, 1, \ldots, n-1, \quad \deg f(x) \leq n-1.$$

Если векторы u и v совпадают (искажений не было), то $f(x) = b(x)$. Из равенств

$$\sigma(\alpha^i)f(\alpha^i) = p(\alpha^i), \quad i = 0, 1, \dots, n-1,$$

получаем сравнение

$$\sigma(x)f(x) \equiv p(x) \pmod{x^n - 1}.$$

На основе данного сравнения получаем следующий алгоритм декодирования.

Алгоритм 3 (алгоритм декодирования Гао для кодов Рида—Соломона).

Вход: принятый многочлен $v(x)$.

Выход: исходный информационный многочлен $b(x)$, если произошло не более $[(d-1)/2]$ ошибок.

1. Интерполяция. Строится интерполяционный многочлен $f(x)$, для которого

$$f(\alpha^i) = v_i, \quad i = 0, 1, \dots, n-1.$$

2. Незаконченный обобщенный алгоритм Евклида. Пусть $r_{-1}(x) = x^n - 1$, $r_0(x) = f(x)$, $v_{-1}(x) = 0$, $v_0(x) = 1$. Производится последовательность действий обобщенного алгоритма Евклида:

$$r_{i-2}(x) = r_{i-1}(x)q_{i-1}(x) + r_i(x),$$
$$v_i(x) = v_{i-2} - v_{i-1}q_{i-1}(x), \quad i \geq 1,$$

до тех пор, пока не достигается такого $r_j(x)$, что

$$\deg r_{j-1}(x) \geq \frac{n+k}{2}, \quad \deg r_j(x) < \frac{n+k}{2}.$$

3. Деление. Информационный многочлен равен $b(x) = \dfrac{r_j(x)}{v_j(x)}$.

6. Программная реализация кодов РС и алгоритма декодирования Гао

Для программной реализации нам понадобятся функции из параграфа 3, за исключением следующих функций:

```
void Extract_information_vector(const uint8 *, uint8 *)
void Search_roots(const POLYNOMIAL *, uint8 *)
void Serach_error_values(const POLYNOMIAL *, const uint8 *, uint8 *, uint8)
int Syndrome_polynomial(const uint8 *, POLYNOMIAL *)
int Generalized_Euclid_algorithm(const POLYNOMIAL *, POLYNOMIAL *, POLYNOMIAL *)
int Decode_Sugiyama(uint8 *)
```

При этом функцию Decode_Sugiyama нужно заменить на

```
int Decode_Gao(const uint8 *v, uint8 *b),
```

которая принимает на вход вектор v и возвращает исходный информационный вектор b. В данной функции применен тот же подход с указателями на многочлены, что и в функции Generalized_Euclid_algorithm.

Ниже приводится программа кодирования и декодирования сообщений методом Гао. Заметим, что вычисление интерполяционного многочлена $f(x)$ в программе вычисляется на основе схемы Горнера, так как описание и реализация алгоритмов быстрого преобразования Фурье над конечным полем заслуживает отдельной работы и потребует от читателя более глубокого погружения в теорию полей (при этом либо код программной реализации очень сильно вырастет в объеме, либо нужно объявлять константы, полученные в ходе предвычислений для конкретного поля, которые сведут на нет понимание принципа действия). Про такие алгоритмы можно посмотреть, напр., в [13, 14]. Поэтому сложность полученного алгоритма составляет $O(n^2)$. Но, опять же, стоит оговориться, что в приведенной ниже реализации за счет таблицы степеней и таблицы дискретных логарифмов операция умножения сводится к операции сложения и обращению к элементам таблиц.

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

// Параметры конечного поля
#define M 8 // поле GF(2^M)
#define NMAX 255 // число 2^M - 1 - максимальное M-битное число
#define P 0x11d // многочлен p(x)=x^8+x^4+x^3+x^2+1 в шестнадцатеричном виде
#define ALPHA 2 // 00000010 - корень многочлена p(x)

typedef unsigned char uint8;
typedef unsigned short uint16;

uint8 deg_alpha[2*NMAX + 1]; // степени элемента alpha
uint8 log_alpha[NMAX + 1];  // таблица дискретных логарифмов по основанию alpha

// Параметры кода
#define N 255 // длина кода
#define T 10 // максимальное число исправляемых ошибок
#define K 235 // размерность кода K=N-2T
#define D 21 // D = N - K + 1 - кодовое расстояние

// Структура, содержащая инфомацию о многочлене
typedef struct POLYNOMIAL
{
```

```
    int deg; // степень многочлена
    uint8 array[N+1]; // коэффициенты многочлена
} POLYNOMIAL;

// Произведение элементов a и alpha
uint8 Product_alpha(uint8 a)
{
    uint16 c = a;
    c <<= 1;
    if ((c >> M) & 1)
         c ^= P;
    return c;
}

// Произведение элементов поля a и b
uint8 Product(uint8 a, uint8 b)
{
    uint16 buf; // для промежуточных вычислений
    uint8 rez; // результат произведения
    buf = b;
    rez = 0;
    while(a)
    {
        if (a & 1)
            rez ^= buf;
        buf <<= 1;
        if ((buf >> M) & 1)
            buf ^= P;
        a >>= 1;
    }
    return rez;
}

// Вычисление степеи многочлена a(x)
void Deg(POLYNOMIAL *a)
{
    int i;
    for(i = N-1; i >= 0 && !(a->array[i]); i--)
        ;
    a->deg = i;
}
```

```
// Вычисление степени многочлена a(x), если deg a(x) <= n
void Deg_n(POLYNOMIAL *a, int n)
{
    int i;
    for(i = n; i >= 0 && !(a->array[i]); i--)
        ;
    a->deg = i;
}

// Заполнение нулями
void Init_zeros(POLYNOMIAL *a)
{
    int i;
    for(i = 0; i < N; i++)
        a->array[i] = 0;
    a->deg = -1;
}

// Заполнение нулями коэффициенты с номерами от 0 до n
void Init_zeros_n(POLYNOMIAL *a, int n)
{
    int i;
    for(i = 0; i <= n; i++)
        a->array[i] = 0;
    a->deg = -1;
}

// Произведение многочленов a(x) и b(x) над GF(2^M)
// Результат записывается в многочлен c(x)
void Mult(const POLYNOMIAL *a, const POLYNOMIAL *b, POLYNOMIAL *c)
{
    int i, j;
    Init_zeros_n(c, a->deg + b->deg);
    for(i = 0; i <= a->deg; i++)
        for(j = 0; j <= b->deg; j++)
            if (a->array[i] && b->array[j])
               c->array[i+j] ^=  deg_alpha[ log_alpha[a->array[i]] +
                   log_alpha[b->array[j]] ];
    c->deg = a->deg + b->deg;
}
```

```
// Сложение многочленов a(x) и b(x)
// Результат записывается в многочлен c(x)
void Add(const POLYNOMIAL *a, const POLYNOMIAL *b, POLYNOMIAL *c)
{
    int i;
    if (a->deg < b->deg)
    {
        for(i = 0; i <= a->deg; i++)
            c->array[i] = a->array[i] ^ b->array[i];
        for(i = a->deg + 1; i <= b->deg; i++)
            c->array[i] = b->array[i];
        c->deg = b->deg;
    }
    else
    {
        for(i = 0; i <= b->deg; i++)
            c->array[i] = a->array[i] ^ b->array[i];
        for(i = b->deg + 1; i <= a->deg; i++)
            c->array[i] = a->array[i];
        c->deg = a->deg;
    }
}

// Умножение многочлена a(x) на ненулевую константу lambda
// Результат записывается в многочлен b(x)
void Mult_lambda(const POLYNOMIAL *a, uint8 lambda, POLYNOMIAL *b)
{
    int i;
    for(i = 0; i <= a->deg; i++)
        if(a->array[i])
            b->array[i] = deg_alpha[ log_alpha[a->array[i]] +
                log_alpha [lambda] ];
        else b->array[i] = 0;
    b->deg = a->deg;
}

// Копирование b в a
void Copy(POLYNOMIAL *a, const POLYNOMIAL *b)
{
    int i;
```

```
    for(i = 0; i <= b->deg; i++)
        a->array[i] = b->array[i];
    a->deg = b->deg;
}

// Логическая функция деления с остатком многочлена a(x) на многочлен b(x).
// Остаток записывается в *r, целая часть - в *q.
int Mod(const POLYNOMIAL *a, const POLYNOMIAL *b, POLYNOMIAL *q, POLYNOMIAL *r)
{
    int i;
    uint8 c;
    if (b->deg < 0)
        return 0;
    Copy(r, a);
    Init_zeros_n(q, a->deg);
    while(r->deg >= b->deg)
    {
        c = deg_alpha[ NMAX + log_alpha[r->array[r->deg]] -
                              log_alpha[b->array[b->deg]] ];
        q->array[r->deg - b->deg] = c;
        for(i = b->deg; i >= 0; i--)
            if (b->array[i])
                r->array[i + r->deg - b->deg] ^=
                    deg_alpha[ log_alpha[b->array[i]] + log_alpha[c] ];
        Deg_n(r, r->deg - 1);
    }
    Deg_n(q, a->deg);
    return 1;
}

// Вычисление значения многочлена a(x) степени n при x = alpha^i
// с помощью схемы Горнера
uint8 Value_of_polynomial(const uint8 *a, int n, int i)
{
    uint8 y;
    int j;
    if (n < 0)
        return 0;
    y = a[n];
    for(j = n - 1; j >= 0; j--)
        y = (y ? (a[j] ^ deg_alpha[ log_alpha[y] + i ]) : a[j]);
```

```
    return y;
}

// Кодирование информационного вектора b с помощью дискретного преобразования Фурье
void Code(const uint8 *b, uint8 *u)
{
    int i;
    for(i = 0; i < N; i++)
        u[i] = Value_of_polynomial(b, K - 1, i);
}

// Декодирование искаженого вектора с помощью алгоритма Гао
int Decode_Gao(const uint8 *v, uint8 *b)
{
    POLYNOMIAL *f; // интерполяционный многочлен
    // Многочлены для обобщенного алгоритма Евклида.
    // Указатели, чтобы не копировать многочлены
    POLYNOMIAL *r0, *r1, *r, *v0, *v1, *vr, *buf;
    // фактически в этих многочлена будут храниться многочлены
    POLYNOMIAL *q, *pa, *pb, *pc, *pd, *pe, *pf;
    int i;
    f = (POLYNOMIAL *)calloc(1, sizeof(*r0));
    if (f == NULL)
        return 0;
    // Вычисляем многочлен f (1-й шаг алгоритма 3)
    for(i = 0; i < N; i++)
        f->array[i] = Value_of_polynomial(v, N - 1, NMAX - i);
    Deg(f);
    // Выделяем место для массивов r0, r1, v0, v1, q, buf,
    // при этом функция calloc заполняет все значения нулями
    q = (POLYNOMIAL *)calloc(1, sizeof(*q));
    pa = (POLYNOMIAL *)calloc(1, sizeof(*pa));
    pb = (POLYNOMIAL *)calloc(1, sizeof(*pb));
    pc = (POLYNOMIAL *)calloc(1, sizeof(*pc));
    pd = (POLYNOMIAL *)calloc(1, sizeof(*pd));
    pe = (POLYNOMIAL *)calloc(1, sizeof(*pe));
    pf = (POLYNOMIAL *)calloc(1, sizeof(*pf));
    buf = (POLYNOMIAL *)calloc(1, sizeof(*buf));
    if(q == NULL || pa ==NULL || pb == NULL || pc == NULL || pd == NULL ||
       pe == NULL || pf == NULL)
        return 0;
```

```
    pa->deg = -1; pb->deg = -1; pc->deg = -1;
    pd->deg = -1; pe->deg = -1; pf->deg = -1;
    r0 = pa; r1 = pb; r = pc;
    v0 = pd; v1 = pe; vr = pf;
    // Применяем неполный обобщенный алгоритм Евклида (2-й шаг алгоритма 3)
    r0->array[N] = 1; r0->array[0] = 1; r0->deg = N;
    Copy(r1, f);
    v1->array[0] = 1; v1->deg = 0;
    while(r1->deg >= (N+K)/2)
    {
        Mod(r0, r1, q, r);
        buf = r0; r0 = r1; r1 = r; r = buf;
        Mult(q, v1, r);
        Add(v0, r, vr);
        buf = v0; v0 = v1; v1 = vr; vr = buf;
    }
    // Вычисляем информационный вектор (3-й шаг алгоритма 3)
    Mod(r1, v1, q, r);
    for(i = 0; i < K; i++)
        b[i] = q->array[i];

    // Освобождаем память от динамических массивов
    free(f); free(q); free(pa); free(pb); free(pc); free(pd); free(pe); free(pf);
    return 1;
}

// Двоичное представление целого числа a записывается в строку s
void Itoa(uint8 a, char *s)
{
    int i, j, bit;
    for (i = M - 1, j = 0; i >= 0; i--, j++)
    {
       /* выделяем i-й справа бит */
       bit = (a >> i) & 1;
       s[j] = '0' + bit;
    }
    s[j] = '\0';
}

// Вывод сообщения a длины n на экран в двоичном виде
void Print_array(const uint8 *a, int n)
```

```
{
    int i;
    char s[N+1]; // для перевода двоичной записи в строку
    for(i = 0; i < n; i++)
    {
        Itoa(a[i], s);
        printf(s);
    }
    puts("\n");
}

int main()
{
    int i, j;
    uint8 b[K], // информационный вектор
          u[N], // кодовый вектор
          v[N], // принятый вектор
          e[N]; // вектор ошибок

    // Вычисление степеней образующего элемента alpha и
    // и построение таблицы дискретных логарифмов
    deg_alpha[0] = 1;
    for (i = 1; i <= NMAX; i++)
    {
        deg_alpha[i] = Product_alpha(deg_alpha[i-1]);
        deg_alpha[NMAX + i] = deg_alpha[i]; // чтобы не брать "% NMAX"
        log_alpha[deg_alpha[i]] = i;
    }

    srand(time(NULL));
    // Генерируем случайный информационный вектор
    for(i = 0; i < K; i++)
        b[i] = rand() % (NMAX + 1);
    // Выводим информационный вектор b
    puts("b = "); Print_array(b, K);

    // Кодируем информационный вектор b и получаем кодовый вектор u
    Code(b, u);
    // Выводим кодовый вектор u
    puts("u = "); Print_array(u, N);
```

```
    // Генерируем вектор ошибок веса <= T
    for(i = 0; i < N; i++)
        e[i] = 0;
    for(i = 0; i < T; i++)
        e[rand() % N] = rand() % (NMAX + 1);
    // Выводим вектор ошибок e
    puts("e = "); Print_array(e, N);

    // Искажаем кодовый вектор u вектором ошибок e
    for(i = 0; i < N; i++)
        v[i] = u[i] ^ e[i];
    // Выводим искаженный вектор v
    puts("v = "); Print_array(v, N);

    // Декодируем искаженный вектор v
    if (!Decode_Gao(v, b))
    {
        puts("no enough memory");
        return 1;
    }

    // Выводим исходный информационный вектор b
    puts("b = "); Print_array(b, K);

    return 0;
}
```

Заключение

В работе приведена программная реализации кодов Рида—Соломона и алгоритмов декодирования Сугиямы и Гао. При реализации используется таблица степеней и таблица дискретных логарифмов по основанию образующего элемента поля, что значительно упрощает вычисления над конечным полем.

Список литературы

1. Gao S. A new algorithm for decoding Reed–Solomon codes // *Communications, Information and Network Security – Norwell, MA*: Kluwer, 2003, v. 712, p. 55–68.

2. Блейхут Р. *Теория и практика кодов, контролирующих ошибки* / Пер. с англ. М.: Мир, 1986. 576 с.

3. Huffman W. C., Pless V. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2003. 646 p.

4. Elia M., Viterbo E., Bertinetti G. Decoding of binary separable Goppa codes using Berlekamp–Massey algorithm // *Electronics Letters*. 1999, v. 35, no. 20, p. 1720–1721.

5. Fedorenko S. V. A spectral algorithm for decoding systematic BCH codes // *IEEE Access*. 2022, v. 10, p. 110639–110645.

6. Рацеев С. М. *Элементы высшей алгебры и теории кодирования* : учебное пособие для вузов. СПб.: Лань, 2022. 656 с.

7. *Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process*. National Institute of Standards and Technology Interagency or Internal Report. NIST IR 8413-upd1. July 2022, 102 p. https://csrc.nist.gov/publications/detail/nistir/8413/final.

8. Hanho Lee. High-Speed VLSI Architecture for Parallel Reed—Solomon Decoder // *IEEE Transactions on VLSI Systems*. 2003, v. 11, no. 2, p. 288–294.

9. Yongge Wang. Decoding Generalized Reed–Solomon Codes and Its Application to RLCE Encryption Scheme // *ArXiv abs*/1702.07737 (2017).

10. Блейхут Р. *Быстрые алгоритмы цифровой обработки сигналов* / перевод с англ. И. И. Грушко. М. : Мир, 1989. 448 с.

11. Lin S. J., Al-Naffouri T. Y., and Han Y. S. FFT algorithm for binary extension finite fields and its application to Reed–Solomon codes. *IEEE Trans. Inf. Theory*. v. 62, no. 10. p. 5343–5358.

12. Nianqi Tang, Yunghsiang S. Han. A New Decoding Method for Reed–Solomon Codes Based on FFT and Modular Approach. *IEEE Transactions on Communications*. 2022, v. 70, no. 12, p. 7790–7801.

13. Трифонов П. В. Методы построения и декодирования многочленных кодов. *дис. ... д-ра техн. наук*: 05.13.17, защищена 01.10.2018. СПб, 2018.

14. Трифонов П. В., Федоренко С. В. Метод быстрого вычисления преобразования Фурье над конечным полем // *Проблемы передачи информации*. 2003, т. 39, вып. 3, с. 3–10.

On implementation of Reed—Solomon codes and decoding algorithms

Ratseev, S. M.

ratseevsm@mail.ru
Ulyanovsk State University, Russia

In the paper a programming implementation of Reed—Solomon codes over a field $GF(2^8)$ is investigated. The implementation of encoding codes using discrete Fourier transform, decoding algorithm based on the Sugiyama method and the Gao decoding algorithm is given. The paper is educational and methodical in nature and can help with the programming implementation of encoders and decoders of Reed—Solomon codes.

Keywords: *error-correcting codes, Reed—Solomon codes, Sugiyama algorithm, Gao algorithm*