Федеральное агентство по образованию РФ

**Федеральное государственное образовательное
учреждение высшего профессионального образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

Факультет математики, механики и компьютерных наук

Кафедра алгебры и дискретной математики

АЛГОРИТМ БЕРЛЕКЭМПА—МЕССИ ДЛЯ ПОЛЕЙ ГАЛУА И ЕГО ПРИМЕНЕНИЕ

Выпускная квалификационная работа
студента 4 курса очной формы обучения
А. М. Пеленицына

Научный руководитель:
доцент, к.ф.-м.н.
В. М. Деундяк

Ростов-на-Дону
2007

Содержание

1. Введение и постановка задачи

Первоначальная версия алгоритма Берлекэмпа–Месси была изложена Берлекэмпом в 1968 году [1] в качестве элемента конструкции декодера кодов Боуза–Чоудхудри–Хоквингема над конечным полем. Хотя в этой работе была указана возможность формулировки решаемой задачи с использованием понятия линейного регистра сдвига с обратной связью, алгоритм описывался исключительно в терминах полиномов и был весьма сложен для понимания. Спустя год Месси [2] предложил свою интерпретацию алгоритма, как позволяющего строить линейный регистр сдвига минимальной длины, генерирующий заданную последовательность. Эта интерпретация оказалась полезной для более широкого распространения алгоритма, получившего название по имени этих двух ученых. В некоторых работах алгоритм излагается также с помощью непрерывных дробей и рациональной аппроксимации.

С момента появления алгоритма вышло немало работ, развивающих и обобщающих его идеи. Ниже предложен краткий обзор его применений, исчерпывающую библиографию можно найти в [3].

Рядом авторов решались задачи построения полинома наименьшей степени, аннулирующего сразу несколько отрезков над полем, нахождения рангов (степеней минимальных многочленов) всех подотрезков заданного отрезка, обобщения на случай многомерных последовательностей (с использованием теории базисов идеалов в кольцах полиномов от нескольких переменных). Имеются многочисленные результаты применения алгоритма для последовательностей над различными алгебраическими структурами, кольцами разных видов. Были предложены вероятностные версии алгоритма.

Рассматриваемый алгоритм находит применение при декодировании различных классов кодов: кодов Рида–Соломона, кодов БЧХ, циклических и обобщенных циклических кодов, альтернантных кодов и кодов Гоппы, и, наконец, наиболее общего и актуального на сегодня класса кодов – алгеб-

ро-геометрических кодов (вернее, некоторых их подклассов). Алгоритм Берлекэмпа–Месси используется для решения ганкелевых и теплицевых, разреженных и общих систем линейных уравнений, при поиске рациональных аппроксимаций функций и, в частности, аппроксимации Паде. Известны также его приложения в криптографии, при тестировании псевдослучайных последовательностей и для быстрого вычисления в конечных полях.

В работе поставлены следующие задачи:

1) разобрать и представить полное обоснование принципиального алгоритма Берлекэмпа—Месси по работе [4];
2) сконструировать структурную схему алгоритма и получить программную реализацию;
3) разобрать конструкцию декодера для кодов Рида—Соломона, основанного на алгоритме Берлекэмпа—Месси, и построить структурную схему.

2. Алгоритм Берлекэмпа–Месси

Как было отмечено во введении, задачу, решаемую алгоритмом Берлекэмпа–Месси, можно сформулировать по-разному. Для детального понимания принципов работы алгоритма приходится также иметь в виду несколько формулировок и прибегать к каждой в разные моменты времени. Наиболее естественным представляется отталкиваться от задачи нахождения закона рекурсии для линейной рекуррентной последовательности. Введем ряд определений.

Определение 1. *Последовательностью* над полем K будем называть любую функцию $u : \mathrm{N}_0 \to K$, заданную на множестве целых неотрицательных чисел и принимающую значения в этом поле.

Элементы последовательности u будут обозначаться $u(i)$. Будет встречаться также понятие *отрезка последовательности*, которое получается естественным образом из ограничения функции, упомянутой в определении.

Определение 2. Последовательность u будем называть *линейной рекуррентной последовательностью* (ЛРП) *порядка* $m > 0$ *над полем* K, если существуют константы $f_0, \ldots, f_{m-1} \in K$ такие, что

$$u(i+m) = \sum_{j=0}^{m-1} f_j \cdot u(i+j),\ i \geq 0.$$

Указанное выражение назовем *законом рекурсии* или *линейным рекуррентным соотношением*.

Как видно, первые m элементов последовательности не связаны какими-либо ограничениями – они имеют особое значение, их, как правило, называют *начальным отрезком* последовательности u.

Пример 1. Пусть $K = F_2$, $u = (0,1,1,1,0,0,1,0,1,1)$. Легко убедиться, что u можно представить как отрезок ЛРП с $f = (1,0,1,1)$. Однако, в качестве множителей f_i могут быть взяты $f = (1,1,0)$. Зафиксируем возможность

появления такой ситуации, введя сначала понятие характеристического полинома ЛРП.

Определение 3. Пусть u – ЛРП. Многочлен:

$$F(x) = x^m - \sum_{j=0}^{m-1} f_j \cdot x^j$$

с коэффициентами из поля K (этот факт в дальнейшем оговариваться не будет) назовем *характеристическим многочленом* ЛРП u.

Таким образом, каждой ЛРП можно поставить в соответствие характеристический многочлен и обратно, каждому нормированному многочлену можно поставить в соответствие ЛРП. Как стало ясно из примера, одна и та же последовательность может задаваться разными законами рекурсии и, соответственно, иметь разные характеристические полиномы.

Определение 4. Характеристический полином ЛРП u, имеющий наименьшую степень, назовем ее *минимальным многочленом*, а его степень – *линейной сложностью* ЛРП u.

Минимальные многочлены ЛРП, а также их линейная сложность, являются важными характеристиками ЛРП.

Определение 5. Пусть u – последовательность над полем K. Обозначим через $u\left(\overline{0,l-1}\right) = (u(0), \ldots, u(l-1))$ начальный отрезок u. Будем говорить, что многочлен

$$G(x) = x^m - \sum_{j=0}^{m-1} b_j \cdot x^j$$

вырабатывает отрезок $u\left(\overline{0,l-1}\right)$, если

$$\forall i \in [0, l-m-1] : u(i+m) = \sum_{j=0}^{m-1} b_j \cdot u(i+j),$$

то есть если данный отрезок последовательности является отрезком некоторой ЛРП с характеристическим многочленом $G(x)$.

Естественным образом определяется понятие линейной сложности отрезка последовательности как минимальной степени из всех полиномов, вырабатывающих данный отрезок.

Далее будем рассматривать последовательности и многочлены над некоторым полем K. Алгоритм Берлекэмпа–Месси строит многочлен $G(x)$ наименьшей степени, вырабатывающий отрезок $u\left(\overline{0,l-1}\right)$. Чтобы перейти к непосредственному описанию алгоритма, требуется ввести еще ряд технических определений.

Определение 6. Пусть $H(x)=\sum_{j=0}^{n} h_j x^j$ – произвольный многочлен, а v – последовательность. Определим операцию умножения многочлена на последовательность, результатом которой будет последовательность, такая что:

$$H(x)\cdot u = w$$

$$w(i)=\sum_{j=0}^{n} h_j \cdot v(i+j)$$

Очевидно, операция является линейной относительно полинома, входящего в нее.

Для нормированного полинома $G(x)$ определим параметры:

- $k_u(G)$ – количество лидирующих нулей последовательности $G(x)\cdot u$ или ∞, если эта последовательность нулевая;
- $l_u(G)=k_u(G)+\deg(G)$.

Легко убедиться, что $l_u(G)$ – максимальная длина начального отрезка u, вырабатываемого $G(x)$. Действительно, пусть

$$G(x)=\sum_{j=0}^{m} g_j \cdot x^j = x^m - \sum_{j=0}^{m-1} b_j \cdot x^j,\ G(x)\cdot u = v,$$

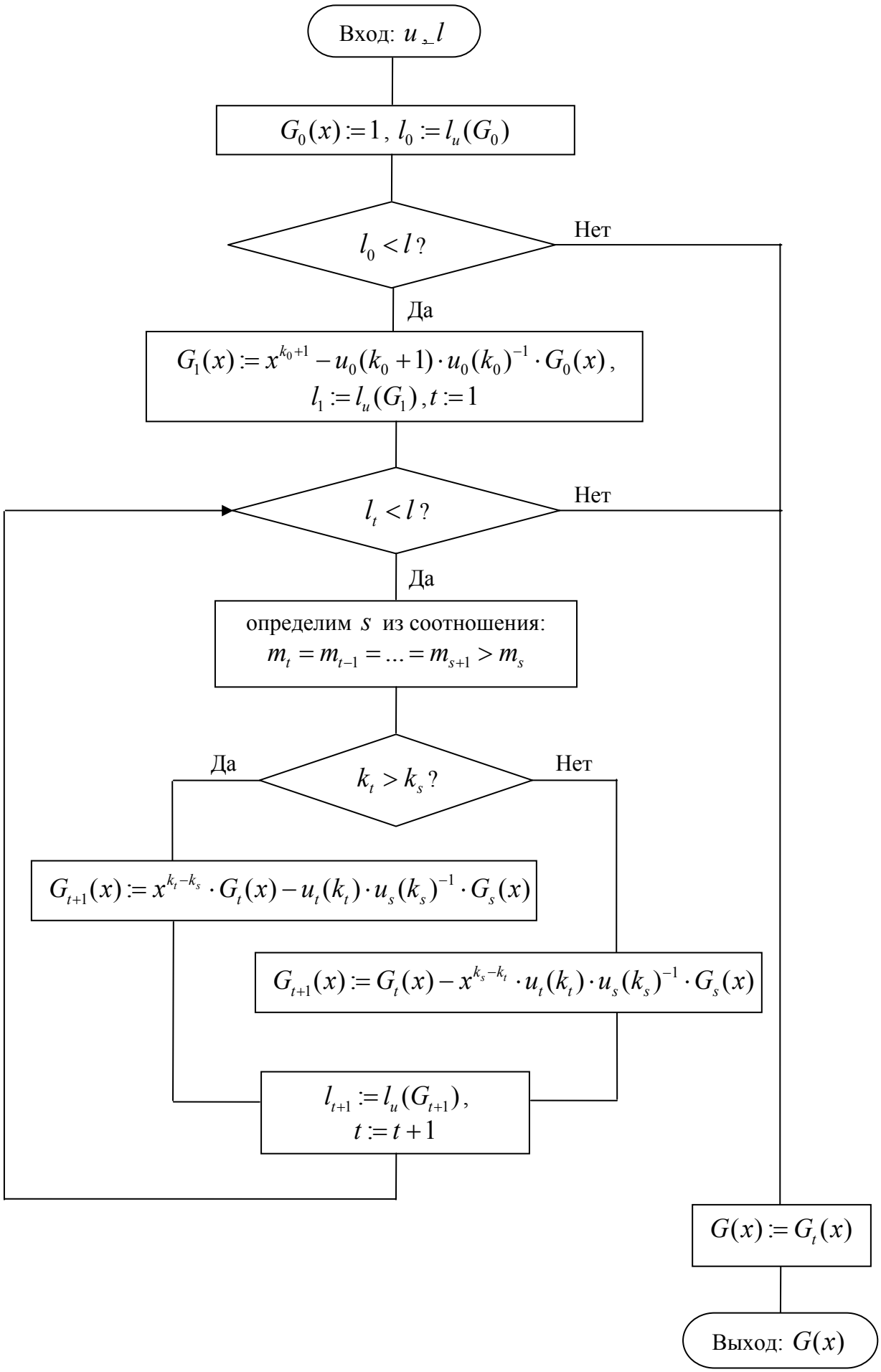
тогда: $\forall i \in [0, l_u(G)-m-1]: v(i)=0$, но:

$$0 = v(i) = \sum_{j=0}^{n} g_j \cdot u(i+j) = u(i+m) - \sum_{j=0}^{m-1} b_j \cdot u(i+j),$$

что и дает искомое:

$$\forall i \in [0, l_u(G) - m - 1] : u(i+m) = \sum_{j=0}^{m-1} b_j \cdot u(i+j).$$

Как было отмечено, существует множество вариаций исследуемого алгоритма, приведенное ниже изложение основано на [4]. Зададимся последовательностью u и числом l, найдем минимальный полином, вырабатывающий отрезок $u\left(\overline{0, l-1}\right)$. Будем строить последовательность нормированных полиномов $G_0(x), G_1(x), \ldots$ неубывающих степеней $0 = m_0 < m_1 \le m_2 \le \ldots$ в соответствии с нижеследующей структурной схемой.


Вход: u, l
$G_0(x) := 1$, $l_0 := l_u(G_0)$
$l_0 < l$?
Нет
Да
$G_1(x) := x^{k_0+1} - u_0(k_0+1) \cdot u_0(k_0)^{-1} \cdot G_0(x)$,
$l_1 := l_u(G_1)$, $t := 1$
$l_t < l$?
Нет
Да
определим s из соотношения:
$m_t = m_{t-1} = \ldots = m_{s+1} > m_s$
$k_t > k_s$?
Да
Нет
$G_{t+1}(x) := x^{k_t-k_s} \cdot G_t(x) - u_t(k_t) \cdot u_s(k_s)^{-1} \cdot G_s(x)$
$G_{t+1}(x) := G_t(x) - x^{k_s-k_t} \cdot u_t(k_t) \cdot u_s(k_s)^{-1} \cdot G_s(x)$
$l_{t+1} := l_u(G_{t+1})$,
$t := t+1$
$G(x) := G_t(x)$
Выход: $G(x)$

Обоснование алгоритма Берлекэмпа–Месси содержится в [4], но ряд его элементов в этой работе опущен. Приведенные ниже рассуждения восполняют эти пробелы. Обоснование алгоритма проведем с помощью доказательства ряда лемм.

Лемма 1. Пусть даны целое число r, полином $G(x)=\sum_{j=0}^{m} g_j \cdot x^j$ и последовательность u, удовлетворяющие условию $r \le k_u(G)$. Тогда для полинома $\widetilde{G}(x)=x^r \cdot G(x)$ имеет место равенство:

$$k_u(\widetilde{G}) = k_u(G) - r .$$

Доказательство. Обозначим:

$$k = k_u(G),\ \tilde{k} = k_u(\widetilde{G}),\ G(x)\cdot u = v,\ \widetilde{G}(x)\cdot u = \tilde{v} .$$

Пусть $\widetilde{G}(x)=x^r \cdot G(x)=\sum_{j=0}^{m+r} \tilde{g}_j \cdot x^j$, где

$$\tilde{g} = \begin{cases} 0, 0 < i < r, \\ g_{i-r}, r \le i \le m+r. \end{cases}$$

Тогда:

$$\tilde{v}(i)=\sum_{j=0}^{m+r} \tilde{g}_j \cdot u(i+j)=\sum_{j=r}^{m+r} g_{j-r} \cdot u(i+j)=[s=j-r]=\sum_{s=0}^{m} g_s \cdot u(i+s+r)=v(i+r).$$

Если $r=k$, то $\tilde{v}(0)=v(r)=v(k)\neq 0$ (определение k), иначе, так как $\forall i \in [0,k-1]: v(i)=0$, то $\forall i \in [0,k-r-1]: v(i+r)=0$, тогда:

$$\forall i \in [0,k-r-1]: \tilde{v}(i)=0 .$$

Последнее, с учетом того, что $\tilde{v}\left(\tilde{k}\right)=\tilde{v}(k-r)=v(k)\neq 0$, и означает $\tilde{k}=k-r$. Лемма доказана.

Лемма 1'. Пусть даны полиномы $F(x)$, $G(x)$ и последовательность u, удовлетворяющие условию $\deg\left(F(x)\right) \le k_u(G)$. Тогда для полинома $\widetilde{G}(x)=F(x)\cdot G(x)$ имеет место равенство:

$$k_u(\widetilde{G}) = k_u(G) - \deg\left(F(x)\right).$$

Доказательство. Следует из леммы 1 и линейности операции умножения полинома на последовательность.

Корректность алгоритма будет определяться двумя фактами: полином, получающийся на каждом шаге, во-первых, вырабатывает более длинный отрезок последовательности u, чем предыдущий, во-вторых, имеет наименьшую степень из всех полиномов, вырабатывающих отрезок данной длины.

Лемма 2. В ходе описанного алгоритма $l_u(G_t)$ возрастает, и степень полиномов m_t не убывает.

Доказательство. Индукция по t:

1) $t=0$: $m_1 = k_0 + 1 > m_0 = 0$, $l_1 = k_1 + m_1 = k_1 + k_0 + 1 > l_0 = k_0 + m_0 = k_0$.

2) Предположим: $\forall t \quad t < t' : l_{t+1} > l_t, m_{t+1} \geq m_t$.

3) Пусть $t = t'$, а s выбрано указанным в алгоритме образом, покажем $m_{t'+1} \geq m_{t'}$. В случае $k_t > k_s$, очевидно, $m_{t'+1} > m_{t'}$. В другом случае степень $G_{t'+1}(x)$ совпадает со степенью $G_{t'}(x)$, так как последняя удовлетворяет соотношению: $(\deg G_{t'}(x) =) m_{t'} > k_s - k_{t'} + m \quad (m_{t'} + k_{t'} = l_{t'} > l_s = m_s + k_s$ – предположение индукции), то есть превосходит степень второго слагаемого, составляющего $G_{t'+1}(x)$.

Индукция завершена и лемма доказана.

Лемма 3. Пусть дана последовательность u, m – линейная сложность ее отрезка длины r, $F(x)$ – полином степени m, такой что:

$$l_u(F) \geq r,$$

$H(x)$ – полином степени n, такой что:

$$l_u(H) > l_u(F),$$

тогда:

$$n \geq \max\{m, k_u(F) + 1\}.$$

Доказательство. Поскольку $l_u(H) > r$, то $n > m$. Остается показать, что $n \geq k_u(F) + 1$. Предположим противное: $n \leq k_u(F)$. Тогда в соответст-

вии с леммой 1' последовательность $w = H(x) \cdot F(x) \cdot u$ имеет в точности $k_u(F) - n$ лидирующих нулей. С другой стороны, $w = F(x) \cdot H(x) \cdot u$; покажем, что выполнены условия леммы 1':

$$k_u(H) = l_u(H) - n > l_u(F) - n \geq l_u(F) - k_u(F) = m,$$

то есть, действительно, $k_u(F) > m$. Таким образом, можно утверждать, что последовательность w имеет $k_u(x) - m$ лидирующих нулей. Тогда $k_u(H) + n = k_u(F) + m$. Окончательно имеем:

$$l_u(H) = k_u(H) + n = k_u(F) + m = l_u(F),$$

что противоречит условию леммы. Доказательство леммы завершено.

Лемма 4. Степени полиномов, порождаемых алгоритмом, удовлетворяют условию:

$$m_{t+1} = \max\{m_t, k_t + 1\}.$$

Доказательство. Индукция по t:

1) $t = 0$: очевидно.

2) Предположим $\forall t \quad t < t' : m_{t+1} = \max\{m_t, k_t + 1\}$.

3) Пусть $t = t'$, а s выбрано указанным в алгоритме образом, покажем, что $m_{t'+1} = \max\{m_{t'}, k_{t'} + 1\}$. Согласно предположению индукции $m_{s+1} = \max\{m_s, k_s + 1\}$, а так как $m_{t'} = m_{s+1} > m_s$, то $m_{t'} = k_s + 1$. Как видно из рассуждений, проведенных при доказательстве леммы 2, степени полиномов изменяются следующим образом: если $k_{t'} \leq k_s$, то $m_{t'+1} = m_{t'}$, иначе $m_{t'+1} = m_{t'} + k_{t'} - k_s$. Таким образом, в случае $k_{t'} \leq k_s$, необходимо показать, что $m_{t'} \geq k_{t'} + 1$. И действительно, $m_{t'} = k_s + 1 \geq k_{t'} + 1$. В случае же $k_{t'} > k_s$:

$$m_{t'} < m_{t'+1} = m_{t'} + k_{t'} - k_s = k_s + 1 + k_{t'} - k_s = k_{t'} + 1.$$

Тем самым индукция завершена и лемма доказана.

Теорема 1. Изложенный алгоритм строит многочлен минимальной степени, вырабатывающий отрезок последовательности u длины не меньше чем l.

Доказательство. В силу леммы 2 для любого конечного наперед заданного l за конечное число шагов на выходе алгоритма получается полином $G(x)$, для которого $l_u(G) \geq l$. Его степень удовлетворяет точной нижней границе для степеней всех полиномов, вырабатывающих отрезок не короче l (леммы 3 и 4).

3. Реализация алгоритма Берлекэмпа–Месси

Алгоритм Берлекэмпа–Месси реализован на языке программирования C++ в соответствии со стандартом языка [6] с применением объектно-ориентированной методологии. В процессе работы над реализацией алгоритма была использована свободно распространяемая интегрированная среда разработки Microsoft Visual C++ 2005 Express Edition.

В работе также использовалась библиотека с открытыми исходными кодами Schifra Reed-Solomon Error Correcting Code Library [7], распространяемая для академического и некоммерческого использования под лицензией GNU General Public License (версия 2) [8], которая обеспечивает арифметику полей Галуа и базовые операции с полиномами. Библиотека предусматривает работу в расширениях конечных полей характеристики два и хранит список примитивных над F_2 полиномов степени вплоть до шестнадцатой включительно. Последнее означает, что без дополнительных данных работа алгоритма поддерживается в полях мощности не более чем 2^{16}.

Математический аппарат, построенный во второй главе, отображается в программную модель с точностью до одного момента. Ограниченность памяти компьютера приводит к невозможности оперировать с бесконечными последовательностями, потому понятие линейной рекуррентной последовательности было замещено понятием *линейного регистра сдвига* (ЛРС). ЛРС хранит *множители*, соответствующие элементам $f_0, \ldots, f_{m-1}$ из определения ЛРП, и текущий отрезок последовательности (длины m), необходимый для вычисления каждого последующего элемента ЛРП в соответствии с линейным рекуррентным соотношением, называемый *состоянием*. Использование ЛРС позволит получать необходимые отрезки ЛРП.

Некоторые менее существенные детали реализации были скорректированы по сравнению с изложенным в главе 2, опираясь на [5].

3.1. Архитектура реализации

В реализации предусмотрено два класса для основных сущностей задачи: класс линейного регистра сдвига и собственно класс алгоритма. Также разработан ряд свободных функций и классов, предназначенных для тестирования. Основные два класса проекта имеют слабое зацепление: единственная информация, которую им следует разделять — фиксированный тип контейнера элементов поля (`GFElementsContainer`, описан в файле utilities.hpp).

Ниже приведен перечень файлов, содержащих исходный код, с кратким описанием их назначения:

- LFSR.cpp (LFSR.hpp) — класс линейного регистра сдвига;
- BM-algorithm.cpp (BM-algorithm.hpp) — класс, реализующий алгоритм вместе со вспомогательным вложенным классом `Monoms` (см. п. 3.3);
- utilities.hpp — средства, необходимые для взаимодействия основных классов программы;
- galois/* — средства библиотеки Schifra, ответственные за арифметику в полях Галуа и операции с полиномами над ними;
- test/* — тесты для основных частей программы;
- main.cpp — точка входа в программу.

В исходном коде присутствуют отладочные секции, которые компилируются только при наличии определения макроса `DEBUG`; его следует опускать в release-сборке.

3.2. Класс линейного регистра сдвига

Класс `LFSR` (Linear Feedback Shift Register — регистр сдвига с линейной обратной связью) предназначен для программного моделирования ЛРС, который, как сказано выше, является всего лишь особым представлением ЛРП. Его цель заключается в проверке корректности работы алгоритма: полученный на выходе алгоритма полином, дополненный последовательностью элементов, выполняющей функцию, аналогичную начальному отрезку ЛРС, используется для построения экземпляра класса `LFSR`. Далее появляется возможность получить произвольный отрезок последовательности, моделируемой данным экземпляром класса `LFSR`, посредством последовательных вызовов экземплярной функции `operator()` и убедиться таким образом, что полученный полином действительно является характеристическим полиномом последовательности, начальный отрезок которой совпадает с начальным отрезком последовательности, заданным алгоритму.

У данного класса имеется набор конструкторов, позволяющих корректно инициализировать два ключевых поля: кортежи множителей и состояния. Кортеж множителей реализован шаблоном стандартной библиотеки C++ `vector<T>`, параметризованным типом элемента поля Галуа `field_element`; его размер фиксирован и не изменяется на протяжении жизни объекта. Для кортежа состояния более приемлемым показалось использование шаблона стандартной библиотеки C++ `deque<T>` — очередь с двумя концами: на каждом такте работы `LFSR` (то есть после каждого вызова функции `operator()`) вычисляется очередной элемент ЛРП, который добавляется в конец кортежа состояния, а клиенту возвращается элемент из начала этого кортежа. Таким образом размер очереди также остается неизменным.

Для класса `LFSR` предусмотрен оператор вывода в поток, обеспечивающий возможность печати содержимого объекта данного класса.

3.3. Класс алгоритма

В классе алгоритма `BMAlgorithm` реализованы основные операции, использованные при изложении алгоритма Берлекэмпа — Месси, как-то: умножение полинома на последовательность и подсчет количества лидирующих нулей в последовательности. После создания экземпляра класса (конструктор принимает лишь ссылку на объект класса `field`, представляющего конечное поле, в котором производятся вычисления), его запуск производится посредством вызова функции-члена `operator();` в качестве единственного аргумента указанной функции передается отрезок ЛРП, для которого необходимо построить минимальный многочлен.

Описанный в главе 2 алгоритм непосредственно выполняется в теле функции `operator()`. Шаги алгоритма отображаются на операторы языка программирования почти дословно.

В ходе алгоритма возникает необходимость многократно вычислять мономы различных степеней (выражения вида x^r). С целью оптимизации этого действия был создан класс `Monoms`, объявленный внутри класса `BMAlgorithm` и являющийся тривиальной оберткой шаблона стандартной библиотеки C++ `map<T>`. Объект этого класса создается вместе с экземпляром класса `BMAlgorithm`, он кэширует мономы, вычисленные на предыдущих шагах работы алгоритма, и в ответ на запрос монома конкретной степени (вызов функции `operator[]`) либо возвращает ссылку на константу (такого уровня доступа оказывается достаточно, учитывая что мономы используются при расчете арифметических выражений), вычисленную ранее, либо вычисляет требуемое значение, добавляет в свою коллекцию и также возвращает ссылку на константу.

Стоит заметить, что выразительность, достигнутая широким использованием перегруженных арифметических операторов, как и обычно в языке C++, не обходится даром — в ходе работы алгоритма создается большое количество временных объектов. Возможным путем избежать накладных расходов, связанных с этим, является использование техники Expression Templates, описанной, например, в [9]. Впрочем, это потребовало бы существенного вмешательства в исходный код библиотеки, что выходит за рамки поставленных задач.

3.4. Тестирование

Хотя практически весь исходный код был в той или иной степени покрыт набором тестов, основным объектом тестирования стал, конечно, класс `BMAlgorithm`. Здесь же стоит отметить, что корректность библиотечных средств также была установлена при помощи написания отдельного ряда тестов. Для основного класса программы было предусмотрено два основных тестовых случая:

- длинная псевдослучайная последовательность;
- все возможные последовательности заданной длины (учитывая, что вычисления проводятся в конечных полях, количество таких последовательностей конечно).

Для запуска проверки указанных тестовых случаев созданы две свободных функции: `totalSequencesTesting()`, `testLongSequence()`. Для реализации обеих проверок был использован класс `SequenceTester`, описание которого приведено ниже.

Назначение класса `SequenceTester` — предоставить средства проверки корректности работы алгоритма на одной заданной последовательности. Проверка запускается вызовом метода `testSequence()`, в качестве аргумента которому передается тестовая последовательность. Далее логика работы такова:

4) построение с помощью имеющегося у каждого объекта-теста экземпляра алгоритма минимального полинома для данной последовательности;
5) создание ЛРС (объекта `LFSR`) на основе полученного полинома;
6) проверка на равенство исходной последовательности и той, которую генерирует ЛРС.

Класс способен регистрировать два вида ошибок, что отражено в наличии объявленного внутри `SequenceTester` перечисления `FailureReason`:

- п.1 завершается неудачей: зацикливание при работе алгоритма (в процессе создания полинома выбрасывается исключение) — константа `ENDLESS_LOOP`;
- п.3 завершается неудачей: получаемая на выходе ЛРС последовательность не совпадает с исходной — константа `NOT_MATCH`.

При возникновении одной из двух указанных ситуаций, тестируемая последовательность вместе с кодом ошибки заносится в коллекцию ошибок, хранимую объектом теста. Клиент класса может осведомиться о наличии ошибок, вызвав метод `hasFailures()` и, если таковые имеются, вывести их на консоль.

Если случай единственной, произвольно длинной (с учетом естественных ограничений вычислительных ресурсов) псевдослучайной последовательности не представил особых препятствий в реализации (все свелось к последовательным вызовам стандартной библиотечной функции `rand()` и передаче полученной последовательности объекту `SequenceTester`), то генерация всех последовательностей заданной длины является известной комбинаторной задачей, функционал которой был вынесен в отдельный класс — `SequencesGenerator`, при этом возникла необходимость принять не слишком привлекательное в отношении дизай-

на тестового модуля решение. Поскольку работа генератора строится из последовательных рекурсивных вызовов одной функции (`sequencesGenerating()`), то каждый раз возвращать клиенту очередную построенную последовательность не удается. Возможны два варианта: предлагать в качестве результата работы основного интерфейсного метода класса (`operator()`) набор из всех последовательностей и затем, для каждого элемента такого набора вызывать тестовый метод класса `SequenceTester` — но это вызвало бы большие затраты памяти; либо вызывать упомянутый тестовый метод прямо в процессе генерации, внутри объекта класса `SequencesGenerator`. В итоге был реализован второй вариант: объект класса-генератора владеет экземпляром `SequenceTester` и вызывает его тестовый метод по мере формирования каждой следующей последовательности. Следует отметить, что в современных языках программирования имеются средства получать промежуточные результаты работы рекурсивных функций без полной раскрутки стека вызовов (например, инструмент `generators` в языке Python).

4. Декодер для кодов Рида–Соломона

Как уже было отмечено во введении, алгоритм Берлекэмпа–Месси имеет многочисленные применения. Однако наиболее важным с практической точки зрения представляется использование данного алгоритма в качестве конструктивного элемента декодера кодов Рида–Соломона. Ниже приведено определение кодов Рида–Соломона и описание декодера [10].

Определение 1. *Линейным кодом длины n размерности k* называется k-мерное подпространство линейного векторного пространства $F_q^{\,n}$.

Определение 2. *Кодом Рида–Соломона* размерности k называется линейный код длины $n = q - 1$, изоморфный идеалу $I = \big(g(x)\big)$ факторкольца $F_q^{\,n}[x] \big/ (x^n - 1)$, где:

$$g(x) = (x - \alpha)(x - \alpha^2)...(x - \alpha^{n-k}),$$

α — примитивный элемент $F_q^{\,n}$.

Можно показать, что коды Рида—Соломона являются МДР-кодами, то есть имеют минимальное кодовое расстояние $d = n - k + 1$ и, таким образом, исправляют не более $t = \left\lfloor \dfrac{n-k}{2} \right\rfloor$ ошибок.

Информационное сообщение — вектор длины k над $F_q^{\,n}$ представляется в виде полинома $f(x)$ степени не выше $k - 1$. Кодирование производится посредством умножения:

$$c(x) = f(x) \cdot g(x).$$

По каналу приходит слово $v(x)$, в котором имеется аддитивная ошибка $e(x) = \sum_{i=0}^{n-1} e_i \cdot x^i$:

$$r(x) = c(x) + e(x).$$

Задача декодера — вычислить $e(x)$.

После получения слова из канала декодер вычисляет компоненты *синдрома* S_j по формуле:

$$\forall j \in [1, n-k] : S_j = r(\alpha^j),$$

заметим, что $r(\alpha^j) = e(\alpha^j)$. То есть:

$$\forall j \in [1, n-k] : S_j = \sum_{i=0}^{n-1} e_i \cdot \alpha^{i \cdot j}.$$

Если все компоненты равны нулю, то считается, что ошибок не произошло.

Определим *локаторы ошибок* x_j и *величины ошибок* y_j следующим образом:

$$\forall j \in [1, \nu] : x_j = \alpha^{a_j},\ y_j = e_{a_j}, \quad a_j \in \{i \mid e_i \neq 0\}.$$

Тогда для S_j получим систему уравнений:

$$\forall j \in [1, n-k] : S_j = \sum_{i=0}^{\nu} y_i \cdot x_i^j \qquad (*)$$

Декодер работает в предположении, что в канале произошло не более чем t ошибок, в таком случае: $\nu \le t$.

Получившаяся система выглядит пессимистично: неизвестны y_i, x_i, ν. Будем находить сначала x_i и ν. Введем еще один вспомогательный объект — *полином локаторов ошибок* $\sigma(z)$:

$$\sigma(z) = (z - x_1)(z - x_2)\ldots(z - x_\nu) = z^\nu + \sum_{i=1}^{\nu-1} \sigma_i z^i.$$

Запишем для $\forall i \in [1, \nu], \forall j \in [1, n-k]$:

$$0 = y_i \cdot x_i^j \cdot \sigma(x_i) = y_i \cdot x_i^{j+\nu} + \sigma_1 \cdot y_i \cdot x_i^{j+\nu-1} + \ldots + \sigma_\nu \cdot y_i \cdot x_i^j.$$

Просуммировав по i, находим:

$$\forall j \in [1, n-k] : 0 = \sum_{i=1}^{\nu} y_i \cdot x_i^{j+\nu} + \sigma_1 \cdot \sum_{i=1}^{\nu} y_i \cdot x_i^{j+\nu-1} + \ldots + \sigma_\nu \cdot \sum_{i=1}^{\nu} y_i \cdot x_i^j,$$

или:

$$\forall j \in [1, n-k] : S_{j+\nu} = -\sigma_1 \cdot S_{j+\nu-1} - \ldots - \sigma_\nu \cdot S_j.$$

Последнее можно представить в матричном виде:

$$\left(\begin{array}{cccc} S_1 & S_2 & \dots & S_\nu \\ \hline S_2 & S_3 & \dots & S_{\nu+1} \\ \hline S_{n-k} & S_{n-k+1} & \dots & S_{n-k+\nu-1} \end{array}\right) \cdot \left(\begin{array}{c} -\sigma_\nu \\ \hline -\sigma_{\nu-1} \\ \hline -\sigma_1 \end{array}\right) = \left(\begin{array}{c} S_{\nu+1} \\ \hline S_{\nu+2} \\ \hline S_{n-k+\nu} \end{array}\right)$$

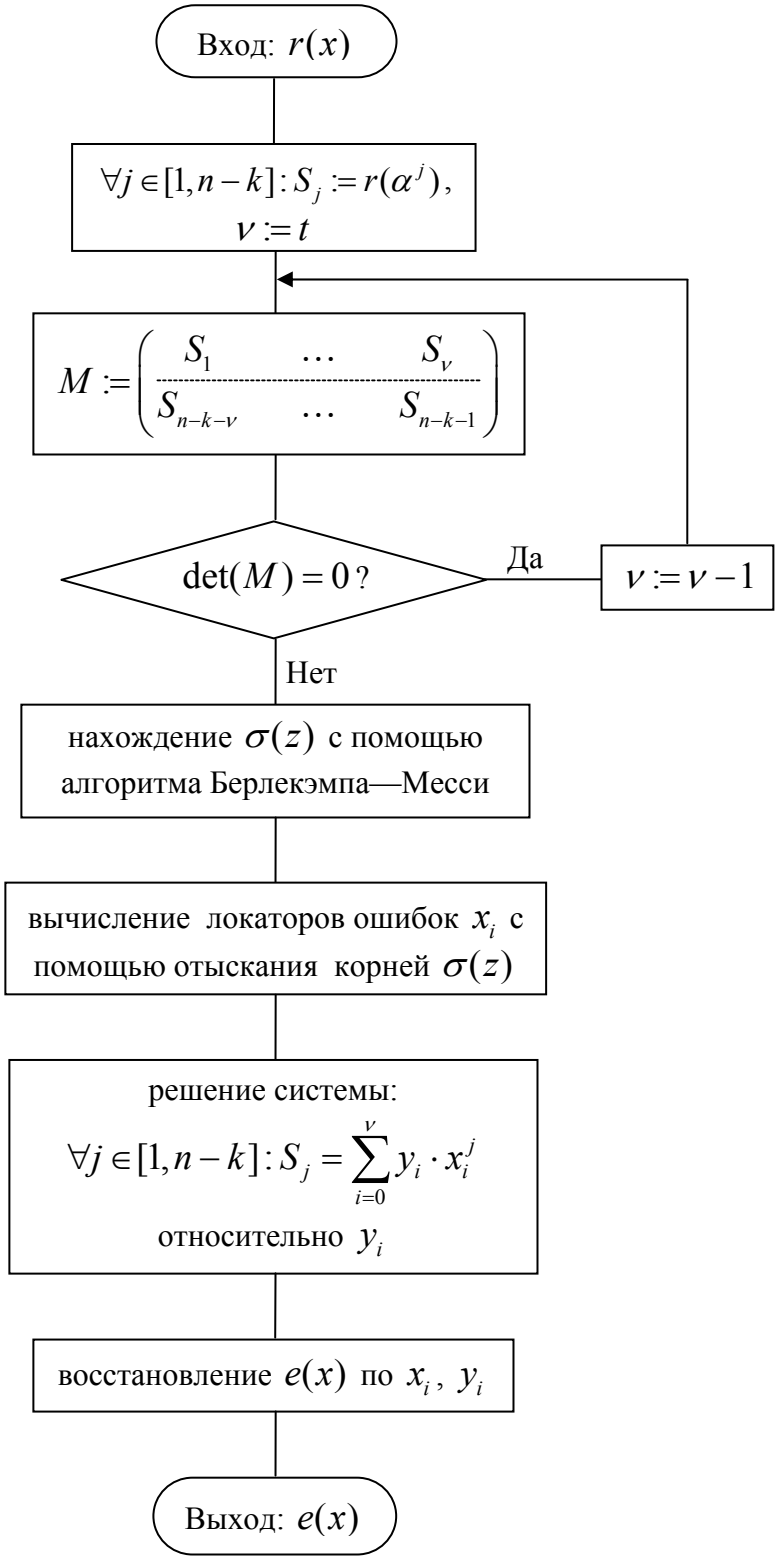
В действительности имеется $n-k$ компонент синдрома, потому последние ν уравнений не рассматриваются. Система примет вид:

$$\left(\begin{array}{cccc} S_1 & S_2 & \dots & S_\nu \\ \hline S_2 & S_3 & \dots & S_{\nu+1} \\ \hline S_{n-k-\nu} & S_{n-k-\nu+1} & \dots & S_{n-k-1} \end{array}\right) \cdot \left(\begin{array}{c} -\sigma_\nu \\ \hline -\sigma_{\nu-1} \\ \hline -\sigma_1 \end{array}\right) = \left(\begin{array}{c} S_{\nu+1} \\ \hline S_{\nu+2} \\ \hline S_{n-k} \end{array}\right).$$

Можно показать, что матрица этой системы – назовем ее M – невырождена в том случае, если в канале произошло ровно ν ошибок (полином $e(x)$ имеет ν ненулевых коэффициентов), и вырождена, если ошибок меньше. Это условие дает способ нахождения ν.

Как видно, полученная система задает отрезок линейной рекуррентной последовательности $S(i)$, множители $-\sigma_j$ которой могут быть найдены с помощью алгоритма Берлекэмпа—Месси. Таким образом может быть найден полином локаторов ошибок $\sigma(z)$.

Далее находятся корни $\sigma(z)$ — как правило, с помощью перебора элементов поля F_q ($процедура\ Ченя$). Затем решается система (*) — существуют эффективные методы ее решения, например, $алгоритм\ Форни$. После этого уже может быть восстановлен $e(x)$. Окончательно, общая схема декодера, носящего имена Питерсона, Горенстейна и Цирлера, может быть представлена следующим образом.


Вход: $r(x)$
$\forall j \in [1, n-k] : S_j := r(\alpha^j)$,
$\nu := t$
$M := \begin{pmatrix} S_1 & \dots & S_\nu \\ S_{n-k-\nu} & \dots & S_{n-k-1} \end{pmatrix}$
$\det(M) = 0$?
Да
$\nu := \nu - 1$
Нет
нахождение $\sigma(z)$ с помощью алгоритма Берлекэмпа—Месси
вычисление локаторов ошибок x_i с помощью отыскания корней $\sigma(z)$
решение системы:
$\forall j \in [1, n-k] : S_j = \sum_{i=0}^{\nu} y_i \cdot x_i^j$
относительно y_i
восстановление $e(x)$ по x_i, y_i
Выход: $e(x)$

5. Литература

1. Berlekamp E. R. Algebraic Coding Theory. – New York: McGrow Hill, 1968 (перевод: Берлекэмп Э. Алгебраическая теория кодирования. – М.: Мир, 1971).
2. Massey J.L., Shift Register Synthesis and BCH Decoding, // IEEE Trans. Inform. Theory. — vol. IT-15, no. 1, 1969.
3. Куракин. Алгоритм Берлекэмпа-Месси над коммутативными артиновыми кольцами главных идеалов // Фундаментальная и прикладная математика. — Том 5, вып. 4, 1999.
4. V. L. Kurakin, A. S. Kuzmin, A. V. Mikhalev, A. A. Nechaev. Linear recurring sequences over rings and modules. // I. of Math. Science. Contemporary Math. and it's Appl. Thematic surveys, vol. 10, 1994, I. of Math. Sciences, vol. 76, № 6, 1995.
5. Алферов А.П., Зубов А.Ю., Кузьмин А.С., Черемушкин А.В. Основы криптографии: учебное пособие. 3-е изд., испр. и доп. — М.: Гелиос АРВ, 2005.
6. ISO Information Technology — Programming Languages — C++ Document Number ISO/IEC 14882-1998 ISO/IEC, 1998.
7. Schifra Reed-Solomon Error Correcting Code Library, http://www.schifra.com/.
8. The GNU General Public License (GPL), Version 2, June 1991, http://www.opensource.org/licenses/gpl-license.php.
9. Vandevoorde, D., N.M. Josuttis. C++ Templates. Boston: Addison-Wesley, 2003. ISBN 0201734842 (перевод: Вандевурд Д., Джосаттис Н. Шаблоны C++: справочник разработчика. — М.: Издательский дом «Вильямс», 2003).
10. Блейхут Р. Теория и практика кодов, контролирующих ошибки: Пер. с англ. — М.: Мир, 1986.

Приложение

Класс линейного регистра сдвига

LFSR.hpp

```
#ifndef LFSR_HPP
#define LFSR_HPP

#include "utilities.hpp"
#include "galois\schifra_galois_field.hpp"
#include "galois\schifra_galois_field_polynomial.hpp"
#include <vector>
#include <deque>
#include <cstddef>
#include <cstdlib>
#include <algorithm>
#include <iostream>
#include <iterator>

using namespace schifra::galois;
using std::vector;
using std::deque;
using std::size_t;
using std::back_inserter;
using std::ostream;

class LFSR {

    typedef vector<field_element> MultipliersContainer;
    typedef deque<field_element> StateContainer;

    MultipliersContainer multipliers;
    StateContainer state;

public:

    LFSR( field_element* const _muls, field_element* const _st, const
size_t size ) {
        multipliers.reserve( size );

        std::copy( _muls, _muls + size, back_inserter( multipliers ) );
        std::copy( _st, _st + size, back_inserter( state ) );

    }

    LFSR( const field_polynomial& poly, field_element* _st );

    LFSR( const field_polynomial& poly, const GFElementsContainer& _st );

    LFSR( field& _gf ) : multipliers( 1, field_element( &_gf, 0) ),
                         state( 1, field_element( &_gf, 0) ) {}

    field_element operator()();

    size_t length() const {
        return multipliers.size();
```

```
    }

    void setState( const StateContainer& newstate ) {
        state = newstate;
    }

    friend ostream& operator<<( ostream& os, const LFSR& lfsr );
};

ostream& operator<<( ostream& os, const LFSR& lfsr );

#endif  // LFSR_HPP
```

LFSR.cpp

```
#include "LFSR.hpp"
#include <algorithm>
#include <numeric>
#include <string>
#include "galois\schifra_galois_field_element.hpp"

ostream& operator<<( ostream& os, const LFSR& lfsr ) {
    using namespace std;
    const size_t width = 79;
    string boldline( width, '=' );
    // string line( width, '-' );

    cout << boldline << endl << "LFSR:" << endl << "size:\t\t" <<
lfsr.length() << endl;

    cout << "multipliers:\t";
    copy( lfsr.multipliers.begin(), lfsr.multipliers.end(),
ostream_iterator<schifra::galois::field_element>(cout," ") );

    cout << endl << "state:\t\t";
    copy( lfsr.state.begin(), lfsr.state.end(),
ostream_iterator<schifra::galois::field_element>(cout," ") );

    cout << endl << boldline << endl;

    return os;
}

field_element LFSR::operator()() {
    field_element result( state.front() );
    const field_element INIT( result.galois_field(), 0 );

    field_element newElement = std::inner_product( state.begin(),
state.end(), multipliers.begin(), INIT );

    state.push_back( newElement );
    state.pop_front();

    return result;
}

LFSR::LFSR( const field_polynomial& poly, field_element* _st ) {
    // it is assumed that the _st points to poly.deg()-number of
field_elements
    size_t polydeg = poly.deg();
    const field_element ZERO( poly.galois_field(), 0 );
    multipliers.reserve( polydeg );
```

```
    for( size_t i = 0; i < polydeg; ++i, ++_st ) {
        multipliers.push_back( ZERO - poly[i] );
        state.push_back( *_st );
    }
}

LFSR::LFSR( const field_polynomial& poly, const GFElementsContainer& _st )
{
    // it is assumed that the _st contains poly.deg()-number of
field_elements
    size_t polydeg = poly.deg();
    const field_element ZERO( poly.galois_field(), 0 );
    multipliers.reserve( polydeg );

    GFElementsContainer::const_iterator stIter = _st.begin();
    for( size_t i = 0; i < polydeg; ++i, ++stIter ) {
        multipliers.push_back( ZERO - poly[i] );
        state.push_back( *stIter );
    }
}
```

Класс алгоритма

BM-algorithm.hpp

```
#ifndef BM_ALGORITHM_HPP
#define BM_ALGORITHM_HPP

#include "utilities.hpp"
#include "galois\schifra_galois_field.hpp"
#include "galois\schifra_galois_field_element.hpp"
#include "galois\schifra_galois_field_polynomial.hpp"
#include <vector>
#include <map>
#include <cstddef>

using std::size_t;

using namespace schifra::galois;

class BMAlgorithm {

public:

    BMAlgorithm( field& _gf ) :  gf(_gf), monoms( gf ) {}

    GFElementsContainer polySeqProduct( const field_polynomial& poly,
                                                        const
GFElementsContainer& seq );
    size_t countLeadingZeroes( GFElementsContainer seq );

    field_polynomial operator()( const GFElementsContainer& seq );
    field_polynomial operator()( field_element* const gfes, size_t size );

    class Monoms {
    public:
        Monoms( field& _gf ) : gf(_gf) {
            field_element gfes[] = { field_element ( &gf, 0 ),
field_element ( &gf, 1 ) };
            data[0] = field_polynomial( &gf, 0, gfes + 1 );
```

```
            data[1] = field_polynomial( &gf, 1, gfes );
        }

        field_polynomial const & operator[]( size_t degree );

    private:
        typedef std::map <size_t, field_polynomial> Container;
        field& gf;
        Container data;

        void operator=( const Monoms& );
    };

private:
    field& gf;
    Monoms monoms;

    // BMAlgorithm
    void operator=( const BMAlgorithm& );

};

#endif  // BM_ALGORITHM_HPP
```

BM-algorithm.cpp

```
#include "BM-algorithm.hpp"
#include "galois\schifra_galois_field.hpp"
#include "galois\schifra_galois_field_polynomial.hpp"
#include <algorithm>
#include <iterator>
#include <fstream>
#include <vector>
#include <iostream>
#include <utility>
#include <limits>
#include <stdexcept>
#include <cassert>
#include <cstddef>

using schifra::galois::field_element;
using schifra::galois::field_polynomial;
using std::size_t;

static void printSequence( const GFElementsContainer& seq,
            std::ostream &out = std::cout  );

GFElementsContainer BMAlgorithm::polySeqProduct( const field_polynomial&
poly,
                                                 const
GFElementsContainer& seq ) {
    assert( poly.deg() < (int)seq.size() );

    const size_t poly_deg = poly.deg();
    const size_t res_size = seq.size() - poly_deg;


    GFElementsContainer result( res_size );

    for ( size_t i = 0; i < res_size; ++i ) {
        result[i] = poly[0] * seq[i];
        for ( size_t j = 1; j <= poly_deg; ++j ) {
```

```
            result[i] += poly[j] * seq[i + j];
        }
    }

    return result;
}

size_t BMAlgorithm::countLeadingZeroes( GFElementsContainer seq ) {
    size_t res = 0;
    const field_element ZERO( &gf, 0 );
    while( res < seq.size() && seq[res] == ZERO )
        ++res;
    // return (res == seq.size()) ? std::numeric_limits<size_t>::max() :
res;
    return res;
}

field_polynomial const & BMAlgorithm::Monoms::operator[]( size_t degree ) {
    Container::iterator it = data.find( degree );
    if ( it == data.end() ) {
        field_polynomial& x = data[1];
        field_polynomial result( x );
        for (size_t i = 1; i < degree; ++i) {
            result *= x;
        }
        return ( ( data.insert( std::make_pair( degree, result ) ) ).first
)->second;
    } else {
        return it->second;
    }
}

field_polynomial BMAlgorithm::operator()( field_element* const gfes, size_t
size ) {
    GFElementsContainer seq;
    seq.reserve( size );

    std::copy( gfes, gfes + size, back_inserter( seq ) );

    return (*this)(seq);
}

field_polynomial BMAlgorithm::operator()( const GFElementsContainer& seq )
{
    size_t k_old = 0, k = 0;
    size_t step = 1, maxsteps = seq.size();
    const size_t INFINITY = std::numeric_limits<size_t>::max();

    //std::cout << "Sequence: ";
    //std::copy( seq.begin(), seq.end(),
    //        std::ostream_iterator<field_element>(std::cout, " ") );
    //std::cout << std::endl;

    k_old = countLeadingZeroes( seq );
    // if ( k_old == INFINITY )    // initial sequence is nullity
    if ( k_old == seq.size() )
        return monoms[1];       // minimal polynomial "x" leads to LFSR
with one multiplier - 0

    if ( k_old == seq.size() - 1 )      // sequences like 0...0a with
length n produced by
        return monoms[ seq.size() ];    // LFSR 0...0 whith length n - not
shorter!
```

```
    field_polynomial G_old( &gf );
    field_polynomial G( &gf );
    field_polynomial G_new( &gf );

    try {
        G_old = monoms[0];
        G = monoms[k_old + 1]
                        - seq[k_old + 1] * seq[k_old].inverse() *
monoms[k_old] * G_old ;
        G_new = G;
    } catch (...) {
        std::cout << "critical error" << std::endl;
    }
    GFElementsContainer u, u_old;

    u = polySeqProduct( G, seq );  u_old = seq ;
    k = countLeadingZeroes( u );
    // std::ostream &out = std::cout;
    // std::ofstream outf("D:/Coding/trace.txt");
    // std::ostream &out = outf;
    // while ( k != INFINITY ) {
    while ( k + G.deg() < seq.size() ) {
        // std::cout << G.deg() << std::endl;
        if( step++ > maxsteps )
            throw std::logic_error("Too many iterations!");
        //if ( int(k) - int(k_old) > 1 ) {
            //out << "G_old:\t" << G_old << std::endl;
            //out << "G:\t\t" << G << std::endl;
            //out << "k_old:\t" << k_old << std::endl;
            //out << "k:\t\t" << k << std::endl;
            //out << "u_old:\t";
            //printSequence(u_old, out);
            //out << "u:\t\t";
            //printSequence(u, out);
            //out << "_____________________________________________";
            //out << std::endl;
        //}
        field_polynomial temp( u[k] * u_old[k_old].inverse() * G_old );
        if ( k <= k_old ) {
            G_new = G - monoms[k_old - k] * temp;
        } else {
            G_new = monoms[k - k_old] * G - temp;
        }
        assert( G_new.deg() >= G.deg() );
        if ( G_new.deg() > G.deg() ) {
            k_old = k;
            u_old = u;
            G_old = G;
        }
        G = G_new;
        u = polySeqProduct( G_new, seq );
        k = countLeadingZeroes( u );
    }

    return G_new;
}

void printSequence( const GFElementsContainer& seq,
        std::ostream &out ) {
    copy( seq.begin(), seq.end(),
            std::ostream_iterator<field_element>( out, " " ) );
    out << std::endl;
}
```

Модуль тестирования алгоритма

testBM-Algorithm.hpp

```
#ifndef TESTBM_ALGORITHM_HPP
#define TESTBM_ALGORITHM_HPP
int testBMA();
#endif  // TESTBM_ALGORITHM_HPP
```

testBM-Algorithm.cpp

```
#include "testBM-Algorithm.hpp"
#include "BM-Algorithm.hpp"
#include "LFSR.hpp"
#include "galois\schifra_galois_field.hpp"
#include "galois\schifra_galois_field_polynomial.hpp"
#include "galois\schifra_galois_field_element.hpp"
#include <algorithm>
#include <utility>
#include <iostream>
#include <iomanip>
#include <vector>
#include <list>
#include <iterator>
#include <limits>
#include <cstddef>
#include <cstdlib>
#include <stdexcept>

using namespace schifra::galois;
using std::copy;
using std::cout;
using std::endl;
using std::size_t;
using std::ostream_iterator;

unsigned int prim_poly2[] = {1, 1, 1};
const unsigned int * prim_poly3 = primitive_polynomial00;
const unsigned int size = 3;
field gf ( 3, 3, prim_poly3 );
const field_element ZERO( &gf, 0 );
BMAlgorithm alg( gf );

field_element gfes[] = {
        field_element(&gf,1),
        field_element(&gf,2),
        field_element(&gf,3),
        field_element(&gf,3),
        field_element(&gf,2),
        field_element(&gf,1),
        field_element(&gf,1),
        field_element(&gf,3),
        field_element(&gf,2),
        field_element(&gf,2),
        field_element(&gf,3),
        field_element(&gf,1)

};

field_polynomial poly1( &gf, 1, gfes ); // ( &gf, 2, gfes);
```

```
int testPolySeqProduct();
int testCountLeadingZeroes();
int testMonoms();
int testAlg();
int totalSequencesTesting();
int testLongSequence();
void printSequence( const GFElementsContainer& seq,
            std::ostream &out = std::cout  );

int testBMA() {
    //testPolySeqProduct();
    //testCountLeadingZeroes();
    // testMonoms();
    // testAlg();
    totalSequencesTesting();
    // testLongSequence();
    return 0;
}

int testMonoms() {
    BMAlgorithm::Monoms monoms( gf );
    for ( size_t i = 0; i < 9; ++i) {
        cout << monoms[i] << endl;
    }

    for ( int  i = 10; i >= 0; --i)
        cout << monoms[0] << endl;
    return 0;
}
int testCountLeadingZeroes() {

    GFElementsContainer seq1(4);
    seq1[0] = gfes[0];
    seq1[1] = gfes[1];
    seq1[2] = gfes[2];
    seq1[3] = gfes[3];

    BMAlgorithm alg( gf );

    GFElementsContainer seq2(4);
    seq2[0] = field_element(&gf,1);
    seq2[1] = gfes[1];
    seq2[2] = gfes[2];
    seq2[3] = gfes[3];

    assert( alg.countLeadingZeroes(seq2) == 0 );

    seq2[0] = field_element(&gf,0);
    seq2[1] = field_element(&gf,0);
    assert( alg.countLeadingZeroes(seq2) == 2 );

    seq2[2] = field_element(&gf,0);
    seq2[3] = field_element(&gf,0);
    assert( alg.countLeadingZeroes(seq2) ==
std::numeric_limits<size_t>::max() );
    //cout <<

    return 0;
}

int testPolySeqProduct() {
    GFElementsContainer seq1(4);
    seq1[0] = gfes[0];
```

```
    seq1[1] = gfes[1];
    seq1[2] = gfes[2];
    seq1[3] = gfes[3];

    BMAlgorithm alg( gf );
    GFElementsContainer prod1 = alg.polySeqProduct( poly1, seq1 );
    assert( gfes[0]*gfes[1] + gfes[1]*gfes[2] == prod1[1] );

    // cout << alg(seq1) << endl;
    //std::cout << prod1.size() << endl;
    //copy( prod1.begin(), prod1.end(),
std::ostream_iterator<field_element>( std::cout," ") );
    //std::cout << endl;

    // cout << gfes[0]*gfes[1] + gfes[1]*gfes[2] << endl;

    return 0;
}

int testAlg() {
    field_element gfes[] = {
        field_element(&gf,3),
        field_element(&gf,1),
        field_element(&gf,0),
        field_element(&gf,5),
        field_element(&gf,4),
        field_element(&gf,7),
        field_element(&gf,2)
    };
    cout << "GF(8)" << endl << "init sequence:\t";
    for ( size_t i = 0; i < 7; ++i ) {
        cout << gfes[i] << " ";
    }
    cout << endl << endl;

    field_polynomial poly( alg( gfes, 7 ) );
    LFSR lfsr( poly, gfes );
    cout << lfsr;
    cout << "LFSR works:\t";
    for ( size_t i = 0; i < 20; ++i ) {
        cout << lfsr() << " ";
    }
    cout << endl;

    return 0;
}

class SequenceTester {
public:
    enum FailureReason { ENDLESS_LOOP = 0, NOT_MATCH = 1 };

    SequenceTester( field& _gf ) : gf( _gf ),
                                   alg( gf ),
                                   failures() {
    }

    void testSequence( const GFElementsContainer& initseq ) {
        try {
            field_polynomial poly( alg( initseq ) );
            LFSR lfsr( poly, initseq );
            GFElementsContainer resultseq;
            std::generate_n( std::back_inserter( resultseq ),
initseq.size(), lfsr);
            if ( initseq != resultseq )
```

```
                failures.push_back( std::make_pair( NOT_MATCH, initseq ) );
        } catch( const std::logic_error& ) {
            failures.push_back( std::make_pair( ENDLESS_LOOP, initseq ) );
        }
    }

    bool hasFailures() { return !failures.empty(); }

    void printFailures() {
        const FailuresContainer::const_iterator end = failures.end();
        for ( FailuresContainer::const_iterator it = failures.begin(); it
!= end; ++it ) {
            cout << std::setw(12) <<
                    ( it->first == NOT_MATCH ) ? "Not match:" : "Endless
loop:";
            copy( it->second.begin(), it->second.end(),
                    ostream_iterator<field_element>( cout, " " ) );
            cout << endl;
        }
    }
private:
    field& gf;
    BMAlgorithm alg;
    typedef std::list< std::pair< FailureReason , GFElementsContainer > >
FailuresContainer;
    FailuresContainer failures;
};

class SequencesGenerator {
    field& gf;
    const size_t length;
    GFElementsContainer currentSequence;
    SequenceTester& tester;

public:
    SequencesGenerator( field& _gf, size_t _len, SequenceTester& _tester )
: gf( _gf ),
                            length( _len ), currentSequence( length ),
tester( _tester ) {}

    void operator()() {
        sequencesGenerating( 0 );
        if ( tester.hasFailures() ) {
             cout << "There were failures!" << endl;
        } else {
            cout << "Succeed!" << endl;
        }
    }

    void sequencesGenerating( size_t currentElementToChangeIdx ) {
        for ( size_t i = 0; i <= gf.mask(); ++i ) {
            currentSequence[ currentElementToChangeIdx ] = field_element(
&gf, i );
            if ( currentElementToChangeIdx == length - 1 ) {
                tester.testSequence( currentSequence ); // next sequence to
test is ready
                // testSequence(); // - just to print
            } else {
                sequencesGenerating( currentElementToChangeIdx + 1 );   //
recursion!
            }
        }
    }
```

```
    void testSequence() {   // stub
        copy( currentSequence.begin(), currentSequence.end(),
                ostream_iterator<field_element>( cout, " " ) );
        cout << endl;
    }
private:
    void operator=( const SequencesGenerator& );
};

int totalSequencesTesting() {
    SequenceTester tester( gf );
    SequencesGenerator generator( gf, 5, tester );
    generator();
    return 0;
}


int testLongSequence() {
    //field gf(primitive_polynomial_size14 - 1, primitive_polynomial_size14
- 1,
    //        primitive_polynomial14); // 17
    //field gf( primitive_polynomial_size11 - 1,
primitive_polynomial_size11 - 1,
    //        primitive_polynomial11); // 13
    const size_t len = 100;
    const size_t fieldSize = gf.mask() + 1;
    GFElementsContainer seq;
    for( size_t i = 0; i < len; ++i )
        seq.push_back( field_element( &gf, rand()%fieldSize ) );
    // printSequence( seq );
    SequenceTester tester( gf );
    tester.testSequence( seq );
    if ( tester.hasFailures() ) {
         cout << "There were failures!" << endl;
    } else {
        cout << "Succeed!" << endl;
    }
    // std::system("pause");

    return 0;
}

void printSequence( const GFElementsContainer& seq,
        std::ostream &out ) {
    copy( seq.begin(), seq.end(),
            ostream_iterator<field_element>( out, " " ) );
    out << endl;
}
```