**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**Кафедра математического моделирования и анализа данных**

АЛИФЕРОВИЧ
Ростислав Михайлович

# ГЕНЕРАЦИЯ ПРОСТЫХ ЧИСЕЛ И АРИФМЕТИКА В ПОЛЯХ ХАРАКТЕРИСТИКИ 2 В КРИПТОГРАФИИ

Дипломная работа

Научный руководитель:
кандидат физ.-мат. наук,
зав. лабораторией НИИ ППМИ БГУ
С.В. Агиевич

Допущена к защите
«___»_____ 2015 г

Зав. кафедрой математического моделирования
и анализа данных
профессор, доктор физ.-мат. наук,
чл.-корр. НАН Беларуси Ю.С. Харин

Минск, 2015

# Реферат

Дипломная работа, 48 с., 9 рис., 5 источников, 1 приложение.

Ключевые слова: криптография, арифметика, оптимальный нормальный базис, оптимизация, генерация простых чисел.

Объект исследования – алгоритмы арифметики в полях характеристики 2, алгоритмы генерации простых чисел

Цель работы – разработка и реализация алгоритмов для ускорения арифметических операций в нормальных базисах полей характеристики 2, оптимизация проверки простоты числа.

Результатом являются разработанные алгоритмы арифметики в оптимальных нормальных базисах, алгоритм предобработки числа для теста на простоту.

Область применения – криптографические алгоритмы над эллиптическими кривыми.

# Рэферат

Дыпломная праца, 48 с., 9 мал., 5 крыніц, 1 дадатак.

Ключавыя словы: крыптаграфія, арыфметыка, аптымальны нармальны базіс, аптымізацыя, генерацыя простых лікаў.

Аб'ект даследавання – алгарытмы арыфметыкі ў палях характарыстыкі 2, алгарытмы генерацыі простых лікаў.

Мэта працы – распрацоўка и рэалізацыя алгарытмаў для паскарэння арыфметычных аперацый ў нармальных базісах палей характарыстыкі 2, аптымізацыя праверкі прастаты ліка.

Вынікам з'яўляецца распрацаваныя алгарытмы арыфметыкі ў аптымальных нармальных базісах, алгарытм папярэдапрацоўкі ліка для тэста на прастату.

Вобласцю ўжывання з'яўляецца крыптаграфічныя алгарытмы над эллiптычнымі крывымі.

# Summary

Graduation assignment, 48 p., 9 pic., 5 sources, 1 appendix.

Keywords: Cryptography, arithmetics, optimal normal base, optimization, prime numbers generation

The object of the study – arithmetics algorithms in the fields with characteristics 2, prime numbers generation algorithms.

The purpose of work – development and implementation of algorithms for accelerate arithmetics operations in normal bases in the fields with characteristics 2, optimization of primality test.

The result is the implemented arithmetics algorithms in optimal normal bases, preprocessing algorithm for checking number on primality.

The application field is the cryptographic algorithms over elliptic curves.

# Содержание

# УСЛОВНЫЕ ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

$GF(q)$ - поле Галуа характеристики $q$

ОНБ - оптимальный нормальный базис

$\delta_{i,j}$ - дельта-символ Кронекера, равный 1 при $i = j$, и 0 в остальных случаях.

$deg\ a$ - степень многочлена $a$.

$PRIMES$ - множество простых чисел

# Введение

Современные криптографические системы с открытым ключом существенно используют вычисления с большими целыми числами. Более того, во многих криптосистемах (например, RSA) используются большие простые числа. Генерация простых чисел является важным компонентом таких систем. Обычно генерация простых чисел выполняется по следующей схеме: выбрать число-кандидат, проверить делимость числа на малые простые, проверить простоту с помощью теста Рабина – Миллера. Также современная криптография немыслима без использования эллиптических кривых. Этим математическим объектам посвящен целый раздел в криптографии. Для реализации достаточно надежных эллиптических кривых используются расширения поля $GF(2)$. Важно быстро уметь проводить операции над объектами этих полей, чему посвящен раздел по ускорению вычислений с использованием оптимальных нормальных базисов. Их преимущество в возможности быстро возводить в степень, кратную двум. Дипломная работа посвящена ускорению проверки делимости числа на малые простые и оптимизации арифметических операций в полях характеристики 2.

# 1 Генерация простых чисел

## 1.1 Постановка задач

Для генерации простых чисел используются различные методы: детерменированный алгоритм генерации простого числа по заданному числу (seed), решето Эратосфена и тому подобные алгоритмы, генерация случайного числа с последующей проверкой его на вероятностных тестах.

В этой работе для будем оптимизировать генерацию простого числа с помощью проверки случаного числа в тесте на простоту (в Частности теста Миллера-Рабина). Наша цель заключается в выработке некоторого предварительного этапа, после которого вероятность теста Рабина-Миллера была бы выше, чем без него(не в ущерб времени выполнения с наперед заданной вероятностью). Для выполнения этой задачи есть некоторые подходы, описанные ниже. Речь пойдет о следующем:

### 1.1.1 Фактор-множество

На любом конечном интервале натуральных чисел почти половина чисел делится на 2, примерно треть на три и так далее. Таким образом, проверив делимость числа на малые простые, мы многократно повысим вероятность того, что оно простое. Поэтому есть смысл ее проверить. Создадим множество простых последовательных чисел, назовем его фактор-множеством, и обозначим $F$, где $F = \{x_i \mid x_i \in PRIMES\}$. Ниже будет описан алгоритм быстрого нахождения остатка от деления на малое число ModDigit. В нем есть некоторый порог $m$, который означает, что можно быстро найти остаток от деления числа $n$ на число, не большее $m$. При построении фактор-множества придется это учесть.

При увеличении мощности фактор-множества растет вероятность успеха теста Миллера-Рабина, и в предельном случае когда фактор-множество максимально, вероятность теста равна 1, но асимптотика времени работы - экспоненциальна. Таким образом задача состоит в оценке мощности фактор-множества.

### 1.1.2 Разбиение

Существует также следующая оптимизация: фактор-множество можно разбить на подмножества, внутри которых произведение чисел меньше порога для алгоритма ModDigit. Пусть $F = \{F'_i\}, F'_i = \{x_{ij}\}, \prod x_{ij} < m, m$ - порог алоритма ModDigits. Очевидно, что мощность множества $\{F'_i\}$ меньше мощности множества $\{x_i\}$. В этом случае количество вызовов алгоритма ModDigits меньше. Тогда множество остатков нужно считать немного по-другому: сначала находим все $n'_i = n \,(mod \prod x_{ij})$, затем $n'_{ij} = n'_i \,(mod\, x_{ij})$. $R_n = \{n'_{ij}\}$. Таким образом стоит задача оптимального разбиения фактор-множества.

### 1.1.3 Увеличение порога

Предположим, что решив задачу оценки мощности фактор-множества, обнаружим, что этот порог настолько большой, что в фактор-множестве найдутся числа, большие порога для алгоритма быстрого деления. Тогда встает еще одна задача: можно ли модифицировать алгоритм ModDigits таким образом, чтобы его порог можно было увеличить в достаточное число раз?

### 1.1.4 Множество кандидатов

Составим множество остатков $R_n = \{n_i = n \,(mod\, x_i) \mid x_i \in F\}$. Если хотя бы одно из $n_i$ равно нулю, то число n - составное. Отметим, чтобы составить множество $R_{n+2}$ не нужны приведения по модулю больших чисел, а всего лишь сложение по модулю для маленьких $R_{n+2} = \{n_i + 2 \,(mod\, x_i) \mid n_i \in R_n\}$. Таким образом можно быстро проверить делимость на малые простые некоторого множества кандидатов, назовем его $K$. Понятно, что когда-либо в множество кандидатов добавится число, взаимно простое с фактор-множеством. Только при этом может статься, что мощность множества кандидатов велика настолько, что проверка этого множества требует больше времени, чем тест Миллера-Рабина для случайных чисел. Поэтому стоит задача оценки мощности множества кандидатов.

## 1.2 Оптимизация проверки простоты

### 1.2.1 Деление на малый разряд

При пробных делениях на простые делитель много меньше основания системы счисления $b = 2^\omega$. Этим можно воспользоваться и заменить при вычис-

лениях медленное деление разрядов на быстрое их умножение. Такая замена реализована в следующем алгоритме.

---

**Алгоритм 1** ModDigit

---

**Вход:** $u = (u_{n-1} \dots u_1 u_0)_b$ — делимое, $m$ — модуль, $0 < m < \sqrt{b}$.
**Выход:** $u_b \mod m$.
**Шаги:**

1. $(r_1 r_0)_b \leftarrow 0$.
2. $m^* \leftarrow b \mod m$.
3. Для $i = n-1, \dots, 0$:
   (a) $(r_1 r_0)_b \leftarrow (r_1 m^* + r_0)m^* + u_i$.
4. Пока $r_1 \neq 0$:
   (a) $(r_1 r_0)_b \leftarrow r_1 m^* + (r_{0_b} \mod m)$.
5. Возвратить $r_{0_b} \mod m$.

**Корректность:** На шагах алгоритма неявно вычисляется сумма

$$\begin{aligned} r &= u_{n-1}(m^*)^{n-1} + u_{n-2}(m^*)^{n-2} + \ldots + u_1 m^* + u_0 \\ &= (\ldots(u_{n-1}m^* + u_{n-2})m^* + \ldots + u_1)m^* + u_0, \end{aligned}$$

которая сравнима с $u$ по модулю $m$. Контролируется регистр $(r_1 r_0)_b$, содержимое которого сравнимо с $r$ по модулю $m$.
В регистр $(r_1 r_0)_b$ на шаге 3.1 помещается значение

$$u_i + r_0 m^* + r_1 (m^*)^2 \leq (b-1)(1 + (m-1) + (m-1)^2) = (b-1)(m^2 - m + 1) < b^2.$$

Полученная оценка означает, что переполнение при записи не произойдет.
*Сложность.* Требуется $2 + d$ деления разрядов, $2n + d$ умножений и столько же сложений. Здесь $d$ — число дополнительных итераций на шаге 4.
Оценим $d$. По окончании первой дополнительной итерации

$$(r_1 r_0)_b \leq (b-1)(m-1) + (m-1) = bm,$$

а по окончании второй

$$(r_1 r_0)_b \leq m(m-1) + (m-1) = m^2 - 1 < b.$$

Таким образом, $d \leq 2$.
Окончательно получаем: $C_n(ModDigit) = 4D + (2n+2)M + (2n+2)A$.

---

Таким образом видим, что перед запуском более трудоемкого теста Рабина-Миллера или теста Ферма, можно предварительно проверить делимость длинного числа на малые простые. С имеющимся у нас алгоритмом малые простые числа можно брать в диапазоне $[2...\sqrt{b}]$. Известна оценка количества простых чисел от 1 до n: $\pi(x) \to \frac{x}{lnx}$. Понятно, что брать x, близкий

к числу n невыгодно, так как время работы алгоритма перебора будет экспоненциальным. Зато в диапазоне $[2...\sqrt{b}]$ можно сделать перебор по простым делителям. Если число n делится хоть на одно из них, то n однозначно не является простым.

### 1.2.2 Деление на несколько малых разрядов

Следующий алгоритм может использоваться для деления сразу на несколько малых простых.

---

**Алгоритм 2** ModDigits

---

**Вход:** $u = (u_{n-1}...u_0)_b$ - делимое, $m_1, ..., m_s$ - взаимно простые модули, $\prod_{i=1}^{s} m_i < \sqrt{b}$.
**Выход:** $(u \ mod \ m_1, ..., u \ mod \ m_s)$.
**Шаги:**

1. $m \leftarrow 1$.
2. Для $i = 1, ..., s$ :
3. $\quad m \leftarrow m \cdot m_i$.
4. $m \leftarrow ModDigit(u, m)$.
5. Возвратить $(m \ mod \ m_1, ..., m \ mod \ m_s)$.

*Сложность:* $C_n(ModDigit) + (s-1)M + sD = (4+s)D + (2n+4)(1M+1A) + (s-1)M$. Для сравнения, расчет вычетов $u \ mod \ m_i$ по отдельности выполняется за время $4sD + s(2n+4)(1M+1A)$.

---

Имея этот алгоритм, можно несколько соптимизировать предварительное деление n на малые простые в диапазоне $[2...\sqrt{b}]$. Числа из этого диапазона можно разбить на группы чисел$(p_{i_1}, p_{i_2}...p_{i_n})$, произведение которых меньше $\sqrt{b}$. Тогда для этих групп можно провести алгоритм ModDigits. Задача состоит в оптимальности разбиения диапазона на группы. Отметим, что для всех простых битовой длины $\ln(\sqrt{b}) - 1$ нельзя подобрать такого малого простого, чтобы их можно было объединить в одну группу.Таких чисел примерно $\frac{\sqrt{b}}{\ln(\sqrt{b})} - \frac{\sqrt{b}}{2(\ln(\sqrt{b})-1)}$, что довольно много.

### 1.2.3 Тест Миллера-Рабина

**Алгоритм 3** Miller-Rabin

**Вход:** $n$ - число для проверки на простоту, $base$ - базовое множество, $r$ - количество раундов (мощность $base$).
**Выход:** Ответ "составное"или "вероятно простое".
**Шаги:**

1. Представить $n-1$ в виде $2^s t, t$ - нечетно.
2. $r$ раундов:
3. $a \leftarrow base[r]$
4. $a \leftarrow a^t \mod n$
5. Если ($x = 1$или $x = n-1$)
6. $r \leftarrow r-1$, перейти на шаг 2.
7. Цикл $s-1$ раз:
8. $x \leftarrow x^2 \mod n$
9. Если ($x = n-1$)
10. $r \leftarrow r-1$, перейти на шаг 2.
11. Вернуть "составное".
12. Вернуть "вероятно простое".

**Сложность:** $O(\log_2^3 n)$. В программе будет использовано базовое множество $base = \{2, 7, 61\}$, которое гарантирует безошибочность теста на числах до 4759123141.

### 1.2.4 Оценка сложности генерации простого числа

Будем решать задачу генерации простого числа заданной длины. Для этого выберем случайное $n \xleftarrow{R} [2^{\log_2 n}, 2^{\log_2 n+1} - 1]$. Также у нас есть некоторое фактор-множество $F = \{x_i \mid x_i \in PRIMES\}$, которое состоит из последовательных малых простых чисел, исключая число 2. Составим систему остатков $R_n = \{n_i = n \ (mod \ x_i) \mid x_i \in F\}$. Если среди них есть хотя бы один ноль, то число $n$ делится на соответствующее малое простое, следовательно $n$ является составным. Заметим, что можно просто построить систему $R_{n+2}$, зная заранее $R_n$. Нужно просто к остаткам прибавить двойку и, если нужно, привести по модулю. Таким образом можно просто построить любую другую систему вида $R_{n+k}$, причем имеет смысл $k$ выбирать всегда четным, иначе $n+k$ получится четным, следовательно составным. Метод состоит в том, чтобы строить $R_n$, и в случае того, если $R_n$ содержит ноль, то строить следующе $R_{n+2}$, и так далее, пока не найдем такой $R_{n+2i}$, что он не содержит

нуля, т.е. взаимно прост со всеми числами фактор-множества. Недостаток метода в том, что при большой цепочке подряд идущих составных числах, может оказаться, что $i$ довольно велико, при этом необходимо на каждой итерации заново пересчитывать все суммы.

В связи с этим предлагается следующая оптимизация этого метода. Рассмотрим конкретное число из факторной базы $p = 5$, и $n = 29$. $n \mod p = 29 \mod 5 = 4$. Чтобы получить число, делящееся на 5, к 29 необходимо добавить минимальное четное число 6, а потом добавлять всегда по 10. Таким образом после постройки $R_{n=29}$ не нужно строить $R_{n+6}$, $R_{n+16}$, и т.д. Такие числа как $6, 16, 26, ...$ для $n = 29$ будем называть запрещенными. Таким образом можно построить множество $D$, состоящее из запрещенных чисел для $n$ и всех чисел из факторной базы. Таким образом, если в множестве $D$ не будет содержаться, например число 4, то $R_{n+4}$ будет взаимно просто со всеми числами факторной базы. При реализации этого метода ограничим число $i$ некоторым порогом (например максимальным числом из факторной базы + 1), чтобы иметь гарантированно константное использование памяти. При этом необходимо предварительно посчитать $|F|$ раз остаток от деления $n$ на каждое простое число. При составлении множества запрещенных чисел на каждой итерации будет добавляться примерно $\frac{|F|}{p_i}$ чисел. Таким образом, генерация множества запрещенных чисел имеет асимптотику

$$O(|F| \sum_{p \in F} \frac{1}{p}) = O(|F| \ln \ln p_{max}),$$

где $p_{max}$ - максимальное число из факторной базы. Согласно теореме о распределении простых чисел

$$|F| \approx \frac{p_{max}}{\ln p_{max}}.$$

Следовательно генерация множества запрещенных чисел имеет асимптотику

$$O(\frac{p_{max} \ln \ln p_{max}}{\ln p_{max}}).$$

Этап построения $R_n$ имеет асимптотику $O(|F| \log_2 n)$, так как выполняется $|F|$ операций приведения по малому модулю. В итоге искомая асиптотика равна

$$O(\frac{p_{max}(\ln \ln p_{max} + \log_2 n)}{\ln p_{max}}).$$

# 2 Арифметика в полях характеристики 2

## 2.1 Базисы конечных полей

Здесь приведем основные сведения о стандартных и нормальных (в том числе и оптимальных) базисах описанные в [2].

### 2.1.1 Стандартные и нормальные базисы

Через $GF(q^n)$ обозначим конечное поле порядка $q^n$, рассматриваемое как расширение степени n поля $GF(q)$ порядка q. В качестве представления элементов поля $GF(q^n)$ используем многочлены степени не более $n-1$ с коэффициентами из поля $GF(q)$. Если многочлены записаны в стандартном базисе

$$B_\alpha = \{\alpha^0, \alpha^1, ..., \alpha^{n-1}\}$$

(в этом случае элемент $\alpha$ называем генератором базиса), то сложение элементов поля $GF(q^n)$ сводится к покомпонентному сложению в поле $GF(q)$ векторов коэффициентов, соответствующих данным многочленам, а умножение элементов поля есть умножение соответсвующих многочленов над полем $GF(q)$, выполняемое по модулю неприводимого над полем $GF(q)$ многочлена $g(x)$, определяющего рассматриваемое представление поля.

Иногда вместо стандартного базиса удобнее так называемый нормальный базис, то есть базис вида

$$B^\alpha = \{\alpha^{q^0}, \alpha^{q^1}, ..., \alpha^{q^{n-1}}\}$$

который порождается генератором $\alpha$ стандартного базиса - корнем неприводимого над полем $GF(q)$ многочлена $g(x)$ в своем поле разложения $GF(q^n)$. Нормальный базис существует для любого $n$, но порождается не всяким неприводимым многочленом $g(x)$, так как составляющие его степени элемента $\alpha$ должны быть линейно независимыми над полем $GF(q)$.

Если система степеней

$$\{\alpha^{q^0}, \alpha^{q^1}, ..., \alpha^{q^{n-1}}\}$$

образует нормальный базис, то любой элемент $\zeta$ поля $GF(q^n)$ однозначно представляется в виде

$$\zeta = x_0\alpha + x_1\alpha^q + x_2\alpha^{q^2} + ... + x_{n-1}\alpha^{q^{n-1}}$$

где $x_0, ..., x_{n-1}$ - коэффициенты из поля $GF(q)$.

Сложение в нормальном базисе, как и в стандартном, есть покомпонентное сложение векторов коэффициентов в поле $GF(q^n)$.

Возведение в степень $q$ (а значит, и в любую степень $q^m$) в нормальном базисе представляет собой циклический сдвиг коэффициентов, так как

$$\zeta^q = x_{n-1}\alpha + x_0\alpha^q + x_1\alpha^{q^2} + ... + x_{n-2}\alpha^{q^{n-1}}$$

Рассмотрим умножение в нормальных базисах.

Сложностью $C_B$ произвольного нормального базиса

$$B = \{\alpha, \alpha^{q^1}, \alpha^{q^2}, ..., \alpha^{q^{n-1}}\}$$

называется число ненулевых элементов в матрице $T$, $i$-я строка которой есть вектор коэффициентов элемента $\alpha\alpha^{q^i}$ поля $GF(q^n)$ относительно базиса $B$, то есть

$$\alpha\alpha^{q^i} = \sum_{j=0}^{n-1} t_{i,j}\alpha^{q^i}$$

Это определение мотивируется алгоритмом Месси-Омуры умножения в нормальном базисе $B$.

Пусть

$$\xi = \sum_{i=0}^{n-1} x_i\alpha^{q^i}, \qquad \zeta = \sum_{j=0}^{n-1} y_j\alpha^{q^j}$$

- произвольные элементы поля $GF(q^n)$, разложенные по нормальному базису $B$. Тогда их произведение можно вычислить по формуле

$$\pi = \xi\zeta = \sum_{i,j=0}^{n-1} x_i y_j \alpha^{q^i+q^j} = \sum_{i,j=0}^{n-1} x_i y_j \alpha^{(q^{i-j}+1)q^j},$$

где разность $i - j$ вычисляется по модулю n, а так как

$$\alpha^{(q^{i-j}+1)q^j} = (\alpha^{q^{i-j}+1})^{q^j} = (\sum_{k=0}^{n-1} t_{i-j,k}\alpha^{q^k})^{q^j} = \sum_{k=0}^{n-1} t_{i-j,k}\alpha^{q^{k+j}} = \sum_{m=0}^{n-1} t_{i-j,m-j}\alpha^{q^m},$$

где разность $m - j$ и сумма $k + j$ тоже вычисляются по модулю n, то

$$\pi = \sum_{m=0}^{n-1} p_m\alpha^{q^m},$$

где

$$p_m = \sum_{i,j=0}^{n-1} t_{i-j,m-j}x_i y_j.$$

Определив матрицу $A$ равенствами $a_{i,j} = t_{i-j,-j}$, где $i-j$ и $m-j$ вычисляются по модулю $n$, замечаем, что предыдущую формулу можно переписать в виде

$$p_m = \sum_{i,j=0}^{n-1} t_{i-j,m-j} x_i y_j = \sum_{k,l=0}^{n-1} t_{k-l,-l} x_{k+m} y_{l+m} = \sum_{i,j=0}^{n-1} a_{i,j} x_{i+m} y_{j+m} = \\ = \sum_{i,j=0}^{n-1} a_{i,j} S^m(x_i) S^m(y_j),$$

где $S^m$ - операция циклического сдвига координат вектора на $m$ компонент, а

$$A(x,y) = \sum_{i,j=0}^{n-1} a_{i,j} x_i y_j$$

- билинейная форма, связанная с матрицей $A$.

Матрица $A$ симметрическая и число $C_B$ ее ненулевых элементов, а также сумма элементов такие же, как и у матрицы $T$. Для вычисления билинейной формы $A(x,y)$ достаточно выполнить $2C_B + n - 1$ сложений и умножений в поле $GF(q)$. Если пренебречь временем выполнения циклических сдвигов, то сложность выполнения умножения над нормальным базисом поля $GF(q^n)$ оценивается сверху как $n(2C_B + n - 1)$ операций в поле $GF(q)$, что видно из следующей основной формулы:

$$\xi\zeta = A(\xi,\zeta)\alpha + A(\xi^{q^{n-1}}, \zeta^{q^{n-1}})\alpha^q + A(\xi^{q^{n-2}}, \zeta^{q^{n-2}})\alpha^{q^2} + ... + A(\xi^q, \zeta^q)\alpha^{q^{n-1}}.$$

Таким образом, арифметическая сложность умножения зависит только от количества ненулевых элементов $C_B$ в матрице $A$. Верхняя граница сложности умножения в произвольном нормальном базисе является кубической. Матрица $A$ (таблица умножения в базисе $B$) однозначно определяет операцию умножения в рассматриваемом поле.

О сложности нормальных базисов известно следующее:

**Теорема 2.1.** *Для любого нормального базиса $B$ поля $GF(q^n)$ его сложность $C_B$ не меньше $2n-1$. Более того, если $q = 2$, то сложность нечетна.*

Нормальные базисы, для которых достигается эта граница, называют оптимальными.

Для вычисления обратного элемента в оптимальном нормальном базисе используется следующая формула: $x^{-1} = x^{2n-2}, x \neq 0$. Существует эффективный алгоритм подсчета правой части этой формулы.

### 2.1.2 Оптимальные нормальные базисы и их использование в стандартах ECDSA и ДСТУ 4145-2002

Оптимальные нормальные базисы были описаны в [6]. В [5] показано, что других оптимальных базисов, кроме найденных в [6], не существует.

Поскольку оптимальные нормальные базисы существуют не во всех полях, то необходимо заранее выбирать поле для операций. В украинском стандарте ДСТУ 4145-2002 для алгоритмов над эллиптическими кривыми в случае использования оптимальных нормальных базисов рекомендуют использовать следующие поля:

Таблица 1 — Допустимые основные поля с оптимальным нормальным базисом

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Степень поля $n$ | 173 | 179 | 191 | 233 | 239 | 251 | 281 |
| Степень поля $n$ | 293 | 359 | 419 | 431 | 443 | 491 | 509 |

Все степени, приведенные в таблице, являются простыми Софи-Жермен, т.е. для них выполняется условие: если $p$ - простое Софи Жермен, то $2p+1$ - простое число.

Различают три типа оптимальных нормальных базисов в поле $GF(q^n)$ по типу их построения.

Первый тип оптимальных нормальных базисов можно построить, когда $n+1=p$ - простое число, а $q$ - примитивный корень по модулю $p$. В этом случае генератором оптимального нормального базиса будет один из примитивных корней $p$-й степени из единицы в поле $GF(q^n)$.

Второй тип оптимальных нормальных базисов возникает, когда $2n+1=p$ - простое число, а элемент $q$, как и в первом случае, - примитивный корень по модулю $p$. Генератором этого базиса служит элемент $\alpha=\zeta+\zeta^{-1}$, где $\zeta$ - примитивный корень $p$-й степени из единицы в поле $GF(q^{2n})$.

Третий тип оптимальных нормальных базисов порождается, когда $2n+1=p$ - простое число, $p \equiv 3 \mod 4$, а $q$ - квадратичный вычет по модулю $p$ и любой квадратичный вычет представляется в виде степени $q$ по модулю $p$ (или, другими словами, порядок элемента $q$ по модулю $p$ равен $n$). Как и в случае базиса второго типа в качестве порождающего элемента базиса третьего типа берется $\alpha=\zeta+\zeta^{-1}$, где $\zeta$ - примитивный корень $p$-й степени из единицы в поле $GF(q^{2n})$.

В американском стандарте ECDSA рассматриваются гауссовы нормальные базисы, которые порождаются элементами вида $x=\zeta+\zeta^{\gamma}+...+\zeta^{\gamma^{k-1}}$, $p=kn+1$ - простое число, $\gamma$ - примитивный корень $p$-й степени из единицы в кольце вычетов порядка $p$. При этом базис называется базисом $k-$го типа.

При $\gamma = 1$ или 2 гауссов нормальный базис является оптимальным нормальным базисом. При этом в тексте стандарта рекомендуют использовать именно эти типы гауссовых нормальных базисов и приоритет отдается гауссову нормальному базису 2-го типа. Остюда заключаем, что рекомендации стандарта ECDSA совпадают с рекомендациями стандарта ДСТУ 4145-2002. Поэтому далее будем рассматривать только поля, предложенные стандартом ДСТУ 4145-2002. Все эти поля имеют либо ОНБ 2-го, либо ОНБ 3-го типа. [3]

## 2.2 Арифметика в нормальных базисах

### 2.2.1 Алгоритм генерации оптимальных нормальных базисов второго типа

Базис второго типа возникает, когда $2n+1 = p$ - простое число, $q$ - является примитивным корнем из 1 по модулю $p$. Элемент $\zeta$ тогда будет примитивным корнем степени $p$ из 1 в поле $GF(q^{2n})$, в качестве порождающего элемента оптимального нормального базиса нужно взять $\alpha = \zeta + \zeta^{-1}$.

**Алгоритм 4** ONB2GEN

**Вход:** степень $n$ расширения поля $GF(2)$.

**Выход:** вектор $b = (b_1, ..., b_n)$ коэффициентов неприводимого многочлена $b_0 + b_1X + ... + b_{n-1}X^{n-1} + b_nX^n$ с корнем $x$ - генератором ОНБ 2-го типа, таблица умножения $A$ в ОНБ, порождаемом генератором $x$ (в виде матрицы или списка координат единичных элементов) или сообщение "ОНБ второго типа не существует".

**Шаги:**

1. Вычислить $p = 2n + 1$, проверить условия ОНБ второго типа:
   а) $p$ - простое число;
   б) 2 - является примитивным корнем 1 по модулю $p$:
   для каждого простого делителя $\delta$ числа $p - 1$ выполняется $2^{(p-1)/\delta} \neq 1$
   Если хотя бы одно из этих условий не выполняется, то вернуть сообщение "ОНБ второго типа не существует".

2. Вычислить вектор $b$:
   для $j = 0, ..., n$: вычислить двоичные коды $N$ и $M$ чисел $\lfloor \frac{(n+j)}{2} \rfloor$ и $j$,
   принять $b_j = 1$, если $\neg(\neg M \vee N) = 0$ и 0 в остальных случаях.

3. Образовать нулевую $n \times n$ матрицу $A$ (или образовать пустой список $A$).
   Заполнить ненулевые элементы:
   для $k = 0, ..., 2n$ вычислить $\pi(k) = 2^k \mod p$;
   для $k = 1, ..., 2n$ вычислить $\pi^{-1}(k) = dlog_2 k \mod p$;
   для $k = 0, ..., n - 1$
   вычислить $\sigma(k) = (\pi^{-1}(1 + 2^k) \mod p) \mod n = (dlog_2(1 + 2^k) \mod p) \mod n$;
   для $k = 1, ..., n - 1$
   вычислить $\mu(k) = (\pi^{-1}(-1 + 2^k) \mod p) \mod n = (dlog_2(-1 + 2^k) \mod p) \mod n$;
   для $k = 0, ..., n - 1$
   включить пару $(k, \sigma(k))$ в список $A$;
   для $k = 1, ..., n - 1$
   включить пару $(k, \mu(k))$ в список $A$;

4. Вернуть $b$ и $A$.

---

## Доказательство оптимальности.

Так как $q^n \equiv -1 \mod p$, то

$$\alpha^{q^n} = \zeta^{q^n} + \zeta^{-q^n} = \zeta + \zeta^{-1} = \alpha,$$

значит, $\alpha^{q^n} = \alpha$, поэтому $\alpha$ принадлежит подполю $GF(q^n)$ поля $GF(q^{2n})$. Система

$$\{\alpha, \alpha^q, \alpha^{q^2}, ..., \alpha^{q^{n-1}}\}$$

линейно независима, так как

$$\alpha^{q^k} = \zeta^{q^k} + \zeta^{-q^k} = \zeta^{q^k} + \zeta^{q^{k+n}}$$

в силу эквивалентности $q^{k+n} \equiv -q^k \mod p$, а система

$$\{\zeta, \zeta^q, \zeta^{q^2}, ..., \zeta^{q^{2n-1}}\}$$

линейно независима. Можно проверить, что при $n > k > 0$

$$\alpha\alpha^{q^k} = (\zeta + \zeta^{-1})(\zeta^{q^k} + \zeta^{-q^k}) = \zeta^{1+q^k} + \zeta^{-1-q^k} + \zeta^{1-q^k} + \zeta^{-1+q^k} = \alpha^{q^s} + \alpha^{q^t},$$

значит, соответствующая этому базису матрица $T$ при $q = 2$ содержит $2n-1$ единицу, потому что при $k = 0$ разложение $\alpha\alpha^{q^0}$ по базису состоит из одного слагаемого, ведь $\alpha^2 = (\zeta + \zeta^{-1})^2 = \zeta^2 + \zeta^{-2} = \alpha^{q^s}$.

Для выполнения указанной проверки заметим, что согласно определению последовательность $q^k \mod p, k = 0, ..., 2n-1$ является перестановкой $\pi(1), ..., \pi(2n)$ множества чисел $\{1, ..., 2n\}$ в силу того, что $q$-примитивный корень по модулю $p = 2n + 1$, причем в силу соотношения

$$q^{k+n} \equiv -q^k \mod p, k = 0, ..., n-1,$$

справедливо равенство

$$\pi(k) + \pi(k+n) = p, k = 0, ..., n-1,$$

а так как все неупорядоченные пары

$$(1 + q^k \mod p, -1 - p^k \mod p)$$

при любом $k, k = 0, ...n-1$, отличны от пары $(0, 0)$, то их последовательность состоит из пар

$$\pi(k), \pi(k+n)), k = 0, ..., n-1,$$

т.е. из неупорядоченных пар вида $(u, p-u), 0 < u \geq n$, значит существует такое отображение $(\sigma(1), ..., \sigma(n))$ множества чисел $1, ..., n$ в себя, что

$$\zeta^{1+q^k} + \zeta^{-1-q^k} = \zeta^{q^{\sigma(k)}} + \zeta^{-q^{\sigma(k)}} = \alpha^{q^{\sigma(k)}}.$$

Аналогично определяется отображение $\mu(1), ..., \mu(n)$ множества $1, ..., n$ в себя, такое, что

$$\zeta^{-1-q^k} + \zeta^{-1+q^k} = \zeta^{q^{\mu(k)}} + \zeta^{-q^{\mu(k)}} = \alpha^{q^{\mu(k)}}$$

при любом $n \geq k > 0$.

Так как

$$1 - q^k \neq 1 + q^k, 1 - q^k \neq -1 - q^k$$

по модулю $p$, то при любом $n > k > 0$ справедливо неравенство

$$\mu(k) \neq \sigma(k).$$

Вспоминая определение матрицы $T$ посредством равенств

$$\alpha\alpha^{q^i} = \sum_{j=0}^{n-1} t_{i,j}\alpha^{q^j}$$

и сравнивая их с полученными равенствами

$$\alpha\alpha^{q^i} = \alpha^{q^{\sigma(i)}} + \alpha^{q^{\mu(i)}}, i > 0,$$

$$\alpha\alpha^{q^0} = \zeta^{1+q^0} + \zeta^{-1-q^0} + \zeta^{1-q^0} + \zeta^{-1+q^0} = \zeta^{1+q^0} + \zeta^{-1-q^0} + \zeta^0 + \zeta^0 = \\ = \zeta^{1+q^0} + \zeta^{-1-q^0} = \alpha^{q^{\sigma(1)}},$$

имеем

$$t_{i,j} = \delta_{\sigma(i+1),j} + \delta_{\mu(i),j}, i \neq 0, t_0, j = \delta_{\sigma(1),j},$$

где $\sigma_{k,s}$ - дельта-символ Кронекера, равный 1 при $k = s$, и нулю в противном случае.

**Свойства матрицы $T$.**

Для оптимизации вычислений при построении оптимального нормального базиса второго типа в поле $GF(2^n)$ докажем два утверждения:

**Утверждение 2.1.** *Для матрицы $T$ оптимального нормального базиса второго типа в поле $GF(2^n)$ равенство $t_{i,j} = 1$ верно тогда и только тогда, когда выполняется одно из четырех соотношений*

$$2^i \pm 2^j \equiv \pm 1 \mod 2n + 1$$

Действительно, согласно определению матрицы $T$ ее элемент $t_{i,j} = 1$, тогда и только тогда, когда в разложении $\alpha\alpha^{2^i}$. Но в силу равенства

$$\alpha\alpha^{2^i} = (\xi + \xi^{-1})(\xi^{2^i} + \xi^{-2^i}) = (\xi^{2^i-1} + \xi^{-2^i+1}) + (\xi^{2^i+1} + \xi^{-2^i-1})$$

и того факта, что $\xi^p = 1, p = 2n + 1$, так как $\xi$ - первообразный корень $p$-й степени из единицы, это возможно, тогда и только тогда, когда $2^i \pm 1 \equiv \pm 2^j \mod 2n + 1$. Последняя эквивалентность равносильна $\pm 2^i \pm 2^j \equiv \pm 1 \mod 2n + 1$ и эквивалентности $2^i \pm 2^j \equiv \pm 1 \mod 2n + 1$.

**Утверждение 2.2.** *В случае оптимального нормального базиса второго типа в поле $GF(2^n)$ матрицы $T$ и $A$ совпадают.*

Действительно, так как $a_{i,j} = t_{i-j,-j}$, то согласно предыдущему утверждению $a_{i,j} = 1$, тогда и только тогда, когда $\pm 2^{i-j} \pm 2^{-j} \equiv \pm 1 \mod p$ , где $i-j$ и $-j$ вычисляются по модулю $n$. Поэтому вместо $-j$ можно взять $n-j$, а вместо $i-j$ при $i-j < 0$ можно взять $n+i-j$. Умножая обе части эквивалентности на $2^j$ по модулю $p$, получаем равносильное соотношение $\pm 2^{i+n} \pm 2^n \equiv \pm 2^j \mod p$, которое в силу эквивалентности $2^n \equiv \pm 1 \mod p$ равносильно соотношению $\pm 2^{i+n} \pm 2^n \equiv \pm 2^j \mod p$. Эквивалентность $2^n \equiv \pm 1 \mod p$ вытекает из малой теоремы Ферма $2^{2n} \equiv 1 \mod p$.

## 2.3 Алгоритм генерации оптимальных нормальных базисов третьего типа

Базис третьего типа возникает, когда $n$ нечетно, $2n + 1 = p$ - простое число, а условие на $q$ заменяется на то, что $q^n \equiv 1 \mod p$, и при любом $0 < k < n, q^k \not\equiv 1 \mod p$ (другими словами, число $q$ имеет по модулю $p$ порядок $n$, а не $2n$, как во втором случае, и тогда автоматически существует такое $r$, что $q = r^2 \mod p$, т.е. $q$ - квадратичный вычет по модулю $p$, и поэтому все его степени $q^k \mod p, k = 0, ..., n - 1$ образуют перестановку множества всех квадратичных вычетов по модулю $p$, так как их ровно $n$ штук).

Поскольку $p$ равно 3 по модулю 4, то -1 является квадратичным невычетом по модулю $p$, потому что в противном случае существовало бы такое число $r$, что $-1 \equiv r^2 \mod p$, и тогда получилось бы противоречие с малой теоремой Ферма:

$$r^{p-1} = (r^2)^{\frac{(p-1)}{2}} \equiv -1 \mod p$$

.

Поэтому из того, что произведение вычета на невычет является невычетом, следует, что последовательность $-q^k \mod p, k = 0, ..., n - 1$ образует перестановку множества всех квадратичных невычетов по модулю $p$.

Как и в случае базиса второго типа, в качестве элемента $\zeta$ берется примитивный корень степени $p$ из 1 в поле $GF(q^{2n})$, а в качестве порождающего элемента оптимального нормального базиса - элемент $\alpha = \zeta + \zeta^{-1}$.

**Алгоритм 5** ONB3GEN

**Вход:** степень $n$ расширения поля $GF(2)$.

**Выход:** вектор $b = (b_1, ..., b_n)$ коэффициентов многочлена $b_0 + b_1X + ... + b_{n-1}X^{n-1} + b_nX^n$ с корнем $x$ - генератором ОНБ 3-го типа, таблица умножения $A$ в ОНБ, порождаемом генератором $x$ (в виде матрицы или списка коордат единичных элементов) или сообщение "ОНБ третьего типа не существует".

**Шаги:**

1. Вычислить $p = 2n + 1$, проверить условия ОНБ третьего типа:
   а) $p$ - простое число;
   б) $2^n \equiv 1 \mod p$;
   в) для каждого простого делителя $\delta$ числа $n$ выполняется $2^{\frac{n}{\delta}} \mod p \neq 1$.
   Если хотя бы одно из этих условий не выполняется, то вернуть сообщение "ОНБ третьего типа не существует".

2. Вычислить вектор $b$:
   для $j = 0, ..., n$: вычислить двоичные коды $N$ и $M$ чисел $\lfloor \frac{(n+j)}{2} \rfloor$ и $j$,
   принять $b_j = 1$, если $\neg(\neg M \vee N) = 0$ и 0 в остальных случаях.

3. Образовать нулевую $n \times n$ матрицу $A$ (или образовать пустой список $A$).
   Заполнить ненулевые элементы:
   для $k = 0, ..., 2n$ вычислить $\pi(k) = 2^k \mod p$;
   для $k = 1, ..., 2n$ вычислить $\pi^{-1}(k) = dlog_2 k \mod p$;
   для $k = 0, ..., n-1$
   вычислить $\sigma(k) = (\pi^{-1}(\pm(1+2^k)) \mod p) \mod n = (dlog_2(\pm(1+2^k)) \mod p) \mod n$;
   для $k = 1, ..., n-1$
   вычислить $\mu(k) = (\pi^{-1}(\pm(-1+2^k)) \mod p) \mod n = (dlog_2(\pm(-1+2^k)) \mod p) \mod n$;
   для $k = 0, ..., n-1$
   включить пару $(k, \sigma(k))$ в список $A$;
   для $k = 1, ..., n-1$
   включить пару $(k, \mu(k))$ в список $A$;

4. Вернуть $b$ и $A$.

---

### Доказательство оптимальности.

Доказательство линейной независимости системы $\{\alpha, \alpha^q, \alpha^{q^2}, ..., \alpha^{q^{n-1}}\}$ отличается от случая базиса второго типа. Допустим противное, т.е.

$$\sum_{j=0}^{n-1} a_j \alpha^{q^j} = 0$$

для некоторого ненулевого вектора $(\alpha_j)$ с координатами из поля $GF(q)$. Подставляя равенства

$$\alpha^{q^i} = \zeta^{q^i} + \zeta^{-q^i},$$

получаем, что

$$\sum_{j=0}^{n-1} a_j(\zeta^{q^i} + \zeta^{-q^i}) = 0$$

Последнее равенство можно переписать в виде

$$\sum_{i=0}^{2n-1} b_i\zeta^i = 0$$

, если определить $b_i$ как $a_j$, где $i = \pm q^j \mod p$ (как было указано выше, последнее условие определяет число $j$ однозначно; знак плюс соответствует случаю, когда число $i$ - вычет, а знак минус - случаю, когда $i$ является квадратичным невычетом по модулю $p$), значит многочлен

$$f(X) = \sum_{i=0}^{2n-1} b_i X^i$$

имеет корни $\zeta$ и $\zeta^{-1}$ в поле $GF(q^{2n})$, а так как эти элементы имеют минимальные многочлены степени $n$ каждый, причем они взаимно просты (в указанном поле корнями первого из них являются $\zeta^{q^i}, i = 0, ..., n-1$, а корнями второго - элементы $\zeta^{-q^i}, i = 0, ..., n-1$), то многочлен $f(X)$ над полем $GF(q)$ должен делиться на их произведение, что невозможно, так как его степень меньше $2n$.

Проверка оптимальности построенного базиса почти ничем не отличается от второго случая. Достаточно проверить, что при $k < n$

$$1 + q^k = \pm q^{\sigma(k+1)} \mod p$$

и

$$1 - q^k = \pm q^{\mu(k)} \mod p$$

(в последнем случае только при $k > 0$), тогда

$$\zeta^{1+q^k} + \zeta^{-1-q^k} = \zeta^{q^{\sigma(k+1)}} + \zeta^{-q^{\sigma(k+1)}} = \alpha^{q^{\sigma(k+1)}},$$
$$\zeta^{1-q^k} + \zeta^{-1+q^k} = \zeta^{q^{\mu(k)}} + \zeta^{-q^{\mu(k)}} = \alpha^{q^{\mu(k)}}, k > 0.$$

Так же, как и в случае второго типа, при $n > k > 0$ имеем

$$\alpha\alpha^{q^k} = (\zeta + \zeta^{-1})(\zeta^{q^k} + \zeta^{-q^k}) = \zeta^{1+q^k} + \zeta^{-1-q^k} + \zeta^{1-q^k} + \zeta^{-1+q^k} = \\ = \alpha^{q^{\sigma(k+1)}} + \alpha^{q^{\mu(k)}},$$

а при $k = 0$

$$\alpha\alpha^{q^0} = \zeta^{1+q^0} + \zeta^{-1-q^0} + \zeta^{1-q^0} + \zeta^{-1+q^0} = \zeta^{1+q^0} + \zeta^{-1-q^0} + \zeta^0 + \zeta^0 = \\ = \zeta^{1+q^0} + \zeta^{-1-q^0} = \alpha^{q^{\sigma(1)}}.$$

**Вычисление матрицы $T$.**

Как и в случае второго типа, выписывая равенства

$$\alpha\alpha^{q^i} = \sum_{j=0}^{n-1} t_{i,j}\alpha^{q^j}$$

и сравнивая их с полученными равенствами

$$\alpha\alpha^{q^i} = \alpha^{q^{\sigma(i+1)}} + \alpha^{q^{\mu(i)}}, i > 0, \alpha\alpha^{q^0} = \alpha^{q^{\sigma(1)}},$$

имеем

$$t_{i,j} = \delta_{\sigma(t+1),j} + \delta_{\mu(i),j}, i \neq 0, t_{0,j} = \delta_{\sigma(1),j},$$

где $\delta_{k,s}$ - дельта-символ Кронекера.

Отметим, что в случае базиса третьего типа матрицы $T$ и $A$ идентичны.

## 2.4 Оптимизация преобразований базисов

### 2.4.1 О комбинированном использовании полиномиального и нормального базисов

Программные реализации умножения в нормальных базисах оказались медленнее алгоритмов умножения в стандартных базисах даже в полях небольших размерностей. Сравнение двух типов базисов, стандартного и нормального, наводит на мысль об ускорении арифметики в конечных полях за счет использования выгодных сторон каждого из них. Действительно, умножение быстрее производить в стандартном представлении поля $GF(2^n)$, а возведение в степень - в нормальном представлении.

Для реализации этой идеи понадобятся матрицы перехода от нормального базиса к стандартному и обратно, которые могут оказаться не разряженными, а плотными, и тогда сложность перехода от одного базиса к другому в худшем случае будет $O(n^2)$. Но в удачном случае сложность перехода может оказаться даже не выше линейной, например в случае, если число ненулевых элементов в матрицах переходов будет $O(n)$. Таким образом, возникает задача поиска нормальных базисов с "простыми"матрицами переходов к стандартным базисам и обратно.

Эта задача легко решается в случае оптимальных нормальных базисов первого типа.

**Теорема 2.2.** *Переход от стандартного базиса поля $GF(q^n)$ к соответствующему(с тем же генератором) оптимальному нормальному базису первого типа(если, конечно, он существует для данного n) и обратно можно выполнить с линейной сложностью.*

*Доказательство.* Легко видеть, что в этом случае базис $\{\zeta, ..., \zeta^n\}$(не совсем стандартный) совпадает с точностью до перестановки с оптимальным нормальным базисом

$$\{\zeta, \zeta^q, \zeta^{q^2}, \zeta^{q^{n-1}}\},$$

так как последовательность чисел $1, q, q^2, ..., q^{n-1}$, вычисленных по модулю $p$, совпадает с некоторой перестановкой

$$\{1, 2, ..., n\}$$

в силу того, что $\zeta$ есть примитивный корень $p-$ степени из единицы в поле $GF(q^n)$, а $q$ - примитивный корень $p$-й степени из единицы в поле $GF(q^n)$, а $q$ - примитивный корень по модулю $p$.

Поэтому, очевидно, переход от базиса $\{\zeta, ..., \zeta^n\}$ к нормальному базису

$$\{\zeta, \zeta^q, \zeta^{q^2}, \zeta^{q^{n-1}}\}$$

и обратно выполняется с оценкой сложности не более чем $n$.

Отметим попутно, что $\zeta$ является корнем неприводимого над полем $GF(q)$ многочлена

$$1 + X + ... + X^n = \frac{X^p - 1}{X - 1}$$

Стандартным базисом для перехода будет базис $\{1, \zeta, ..., \zeta^{n-1}\}$. Очевидно, он с линейной сложностью выражается через базис $\{\zeta, ..., \zeta^n\}$, поскольку

$$\zeta^n = 1 + \zeta + ... + \zeta^{n-1},$$

и обратно, базис $\{\zeta, ..., \zeta^n\}$, благодаря формуле

$$1 = \zeta + ... + \zeta^{n-1} + \zeta^n,$$

с линейной сложностью выражается через базис $\{1, \zeta, ..., \zeta^{n-1}\}$. В результате имеем, что переход от оптимального нормального базиса

$$\{\zeta, \zeta^q, \zeta^{q^2}, ..., \zeta^{q^{n-1}}\}$$

к стандартному базису $\{1, \zeta, ..., \zeta^{n-1}\}$ и обратно выполняется с линейной сложностью. □

### 2.4.2 Оценка сложности перехода от оптимальных нормальных базисов второго и третьего типа к стандартным и обратно

Вначале докажем несколько вспомогательных утверждений (следующая лемма выполняется и для общего случая). Пусть $x = \alpha_1$ - генератор оптимального нормального базиса

$$\{\alpha_1, ..., \alpha_n\}$$

второго типа поля $GF(q^n)$, он же генератор соответствующего стандартного базиса $\{x^0, ..., x^{n-1}\}$, т.е. $\alpha_1 = x^{q^{k-1}} = \zeta^{q^{k-1}} + \zeta^{-q^{k-1}}$, и при любом $k \leq n$

$$\alpha_k = x^{q^{k-1}} = \zeta^{q^{k-1}} + \zeta^{-q^{k-1}},$$

где $\zeta$ - примитивный корень из 1 степени $p = 2n + 1$ в поле $GF(q^{2n})$.

Рассмотрим вспомогательную последовательность $a_0, a_1, ...,$ порожденную элементом $x$. Положим $a_0 = 2$ (в случае $p = 2$, $a_0 = 0$) и, далее, $a_k = \zeta^k + \zeta^{-k}, k = 1, 2, ..., a_1 = \alpha_1$.

Элементы $a_1, a_2, ..., a_n$ этой последовательности образуют почти нормальный базис, являющийся перестановкой элментов нормального базиса.

**Лемма 2.1.** *Для любого $k \geq 1$ имеют место следующие рекуррентные соотношения:*
*1.* $a_{k+1} = a_k a_1 - a_{k-1}$
*2.* $a_{p^k} = x^{p^k}$.
*3.* $a_{p^k+i} = a_i x^{p^k} - a_{p^k-i}$.

*Доказательство.* Для доказательства первой формулы заметим, что

$$(\zeta^k + \zeta^{-k})(\zeta + \zeta^{-1}) = \zeta^{k-1} + \zeta^{-k+1} + \zeta^{k+1} + \zeta^{-k-1},$$

т.е. $a_k a_1 - a_{k-1} = a_{k+1}$.

Вторая формула очевидна.

Третья формула проверяется непосредственно:

$$a_{p^k+i} = \zeta^{p^k+i} + \zeta^{-p^k-i} = (\zeta^{p^k} + \zeta^{-p^k})(\zeta^i + \zeta^- i) - (\zeta^{p^k-i} + \zeta^{-p^k+i}) =$$
$$= a_{p^k} a_i - a_{p^k-i} = x^{p^k} a_i - a_{p^k-i}.$$

□

Рекуррентные соотношения, доказанные в лемме, позволяют записать формулы перехода от почти стандартного базиса $\{x^1, ..., x^n\}$ к почти нормальному базису.

В случае $p = 2$ вычитание в лемме можно заменить на сложение.

Из леммы следует, что для любого $i \geq 1$ элемент $a_i$ почти нормального базиса $\alpha_i$ выражается в виде значения некоторого многочлена степени $i$ над полем $GF(q)$, т.е.

$$a_i = f_i(x) = \sum_{j=1}^{i} f_{i,j} x^j.$$

Выразив таким образом все элементы $a_i$ почти нормального базиса в почти стандартном базисе, получим матрицу перехода $F_n = (f_{i,j})$ от почти стандартного базиса к почти нормальному. В случае, когда $p = 2$ плотность $S(F_n)$(количество ненулевых элементов) матрицы $F_n$ оценивается следующим образом:

**Теорема 2.3.** *Плотность матрицы перехода от почти стандартного базиса поля $GF(2^n)$ к почти нормальному базису второго или третьего типа равна $O(n^{\log_2 3})$.*

Из этой теоремы видно, что матрица перехода является разряженной, и это позволяет осуществлять быстрое умножение в этом базисе. Приведем доказательство этой теоремы.

*Доказательство.* Согласно формулам предыдущей леммы (2.1), матрица $F_{2n+1}$, построенная с их помощью, имеет при $n = 2^k - 1$ вид:

$$F_{2n+1} = \begin{pmatrix} F_n & o_n & O_n \\ 0...0 & 1 & 0...0 \\ G_n & o_n & F_n \end{pmatrix} \tag{2.1}$$

где $o_n$ - нулевой вектор-столбец высоты $n$, $O_n$ - нулевая $n \times n$ матрица, матрица $G_n$ есть симметрическое отражение матрицы $F_n$ относительно средней строки, т.е. $G_n = I_n F_n$, где $I_n = (\delta_{i,n-i+1})$ - матрица с единицами на побочной диагонали и нулями в остальных местах. Например, матрица $F_7$ выглядит так:

$$F_7 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \tag{2.2}$$

Если разбить матрицу на 4 квадратных подматрицы, удалив среднюю строку и средний столбец, получим, что левый верхний квадрат $3 \times 3$ симметричен относительно горизонтали левому нижнему и равен нижнему правому квадрату. Средняя строка и средний столбец содержат ровно одну единицу, лежащую на их пересечении. Матрица является нижнетреугольной с единицами на главной диагонали. Действительно, в общем случае согласно лемме (2.1), при $0 \le i \le 2^k$

$$\sum_{j=1}^{2^k+i} f_{2^k+i,j} x^j = a_{2^k+i} = a_i x^{2^k} + a_{2^k-i} = \sum_{j=1}^{i} f_{i,j} x^{2^k+j} + \sum_{j=1}^{2^k-i} f_{2^k-i,j} x^j,$$

откуда имеем $f_{2^k+i,j} = f_{2^k-i,j}$, при $0 \le j \le 2^k$, и $f_{2^k+i,2^k+j} = f_{i,j}$ при $1 \le j \le 2^k$.

Опираясь на это представление, можно получить следующую рекуррентную формулу для вычисления плотности последовательности матриц $F_n$

$$S(F_{2n+1}) = 3S(F_n) + 1, n \ge 3, S(F_3) = 4,$$

из которой вытекает рекуррентная формула

$$S(F_{2^k-1}) = 3S(F_{2^{k-1}-1}) + 1, k \geq 2, S(F_3) = 4.$$

Полагая $l(k) = S(F_{2^k-1})$, перейдем к линейному рекуррентному соотношению $l(k) = 3l(k-1)+1, l(2) = 4$, решением которого будет $l(k) = (3^k-1)/2$. Обозначив $n = 2^k - 1$ и выразив $3^k = (2^k)^{\log_2 3} = (n+1)^{\log_2 3}$, получим, что

$$S(F_n) = l(k) = \frac{(n+1)^{\log_2 3} - 1}{2} = O(n^{log_2 3})$$

В общем случае равенство $S(F_n) = O(n^{\log_2 3})$ сохраняется, так как выбрав $k$ таким образом, что $2^k - 1 < n \leq 2^{k+1} - 1$, можно заметить, что матрица $F_n$ является главной подматрицей матрицы $F_m$, $m = 2^k - 1$, откуда имеем

$$S(F_n) \leq S(F_m) = O(m^{log_2 3}) = O(n^{\log_2 3}).$$

Аналогично можно оценить плотность матрицы $F'_n = F_n \cdot B_n$, и непосредственно проверяется, что в матрице $B_n$ все элементы нулевые, кроме элементов $b_{i,i+1} = 1$ над диагональю и некоторых элементов нижней строки $b_{n,j}$ и справедливы следующие равенства для элементов матрицы $F'_n$ :

$$f'_{i,1} = f_{i,n}b_{n,1} = \delta_{i,n}b_{n,1}, f'_{i,j} = f_{i,j-1} + f_{i,n}b_{n,j} = f_{i,j-1} + \delta_{i,n}b_{n,j},$$

так как в силу того, что матрица $F_n$ нижнетреугольная, для ее элементов имеем $f_{i,n} = \delta_{i,n}$, где $\delta_{i,n}$ - дельта-символ Кронекера. Складывая эти равенства, получим, что

$$S(F'_n) = \sum_{i=1,j=1}^{n,n-1} f_{i,j} + \sum_{j=1}^{n} b_{n,j} = S(F_n) - 1 + \sum_{j=1}^{n} b_{n,j} \leq S(F_n) - 1 + n = O(n^{\log_2 3}).$$

□

Обозначим через $L(F_n)$ сложность линейного преобразования, определяемого матрицей $F_n$ (в данном случае - это наименьшее число операций сложения по модулю два, необходимых для вычисления этого преобразования).

**Теорема 2.4.** $L(F_n) \leq \frac{n}{2}\log_2 n + 2n = O(n \log_2 n)$

*Доказательство.* Нам будет удобно оценивать сложность преобразования, задаваемого транспонированной матрицей $F_n^T$, которая равна $L(F_n)$ согласно известной лемме о взаимосвязи сложности транспонированных матриц (см. например [1]). Впрочем, для дальнейшего нам нужно будет оценить сложность перехода от координат в нормальном базисе

$$\sum_{i=1}^{n} x_i \alpha_i$$

к координатам в почти стандартном базисе

$$\sum_{i=1}^{n} y_i x^i,$$

которая определяется как раз матрицей $F_n^T$. Пусть $2^k \le n \le 2^{k+1}$, тогда с использованием формул из леммы имеем

$$\sum_{i=1}^{n} y_i x^i = \sum_{i=1}^{n} x_i a_i = \sum_{i=1}^{2^k} x_i a_i + \sum_{i=2^k+1}^{n} x_i a_i =$$

$$= \sum_{i=1}^{2^k} x_i a_i + \sum_{i=1}^{n-2^k} x_{i+2^k} a_{i+2^k} = \sum_{i=1}^{2^k} x_i a_i + \sum_{i=1}^{n-2^k} x_{i+2^k}(a^{2^k} a_i + a_{2^k-i}) =$$

$$= x_{2^k} a_{2^k} + \sum_{i=1}^{2^{k+1}-n-1} x_i a_i + \sum_{i=2^{k+1}-n}^{2^k-1} (x_i + x_{2^{k+1}-i}) a_i + a^{2^k} \sum_{i=1}^{n-2^k} x_{i+2^k} a_i,$$

где $a_0 = 0$. Далее, определяя векторы-столбцы

$$X = \{x_1, ..., x_n\}^T,$$

$$X_1 = \{x_1, ..., x_{2^{k+1}-n-1}, x_{2^{k+1}-n} + x_n, ..., x_{2^k-1} + x_{2^k+1}, x_{2^k}\}^T,$$

где, естественно, в случае $n = 2^{k+1} - 1$

$$X_1 = \{x_1 + x_n, ..., x_{2^k-1} + x_{2^k+1}, x_{2^k}\}^T,$$

а в случае $n = 2^{k+1}$

$$X_1 = \{x_1 + x_{n-1}, ..., x_{2^k-1} + x_{2^k+1}, x_{2^k}\}^T,$$

и во всех случаях

$$X_2 = \{x_{1+2^k,...,x_n}\}^T,$$

и вектора-строки

$$Y_1 = \{y_1, ..., y_{2^k}\}, Y_2 = \{y_{1+2^k}, ..., y_n\},$$

получим

$$\sum_{i=1}^{n} y_i x^i = \sum_{i=1}^{n} x_i a_i = \sum_{i=1}^{2^k} X_{1,i} a_i + x^{2^k} \sum_{i=1}^{n-2^k} X_{2,i} a_i =$$

$$= \sum_{i=1}^{2^k} Y_{1,i} a^i + \sum_{i=1}^{n-2^k} Y_{2,i+2^k} a_{i+2^k},$$

значит

$$F_n^T \otimes X = (y'_i ... y'_n) = (Y_1, Y_2) = (F_{2^k}^T \otimes X_1, F_{n-2^k}^T \otimes X_2), \qquad (2.3)$$

где $\otimes$ - операция умножения матрицы на вектор в поле $GF(2)$. Разумеется, последнее равенство можно было получить, основываясь только на структуре матрицы $F_n$. Осталось индуктивно оценить сложность преобразования координат, определяемого матрицей $F_n^T$. Согласно последнему равенству

$$L(2^{m+1}) \leq 2^m - 1 + 2L(2^m), m \leq k-1, L(2) = 2$$

По индукции непосредственно проверяется, что

$$L(2^m) = 2^{m-1}m + 1$$

Для произвольного $n$ в пределах $2^k < n < 2^{k+1}$ получим

$$L(n) \leq 2^{k_s - 1}k_s + ... + 2^{k_1 - 1}k_1 + s - 1 + (n - 2^{k_s}) + ...$$
$$... + (n - 2^{k_s - ... - 2^{k_2}}) \leq \frac{n}{2}\log_2 n + c\frac{n}{2},$$

где

$$c = 1\frac{2}{2} + \frac{3}{4} + \frac{4}{8} + \frac{5}{16} + ... < 4$$

□

**Теорема 2.5.** *Сложность $B(n)$ перехода в поле $GF(2^n)$ от оптимального нормального базиса второго или третьего типа к соответсвтующему стандартному и наоборот удовлетворяет неравенству*

$$B(n) \leq \frac{n}{2}\log_2 n + 3n$$

*Доказательство.* Обозначим $x = \zeta + \zeta^{-1}$ генератор стандартного базиса

$$A = \{1, x^1, ..., x^{n-1}\}$$

и базиса второго или третьего типа

$$B = \{x^{2^0}, x^{2^1}, ..., x^{2^{n-1}}\}.$$

**Переход от нормального базиса к стандартному**. Переход будет осуществляться через цепочку из четырех базисов

$$B \to B' \to A' \to A$$

и преобразования координат элемента $f$ поля $GF(2^n)$ в этих базисах

$$f = \sum_{i=1}^{n} x_i x^{2^{i-1}} = \sum_{i=1}^{n} x'_i a_i = \sum_{i=1}^{n} y'_i x^i = \sum_{i=1}^{n} y_i x^{i-1},$$

где $a_i$ обозначает $\zeta^i + \zeta^{-i}$, $B' = \{a_1, ..., a_n\}$, $A' = \{x^1, ..., x^n\}$.

**Переход от базиса $B$ к базису $B'$ и преобразование координат $\tilde{x}$ к координатам $\tilde{x}'$.** Этот переход осуществляется перестановкой базисных элементов, и действительно, существует такая перестановка $\pi(i)$ чисел $\{1, ..., n\}$, что для любого $i = 1, ..., 2n$ выполняется равенство

$$2^i \mod p = \pm\pi(i) \in \{1, ..., n\}. \tag{2.4}$$

В самом деле, для базиса второго типа последовательность степеней

$$1, 2, 2^2, ..., 2^{2n-1},$$

вычисленных по модулю $p$, совпадает с некоторой перестановкой

$$\pi(1), ..., \pi(2n)$$

множества чисел $\{1, ..., 2n\}$ в силу того, что 2 - примитивный корень по модулю $p$. А в силу эквивалентности

$$2^{k+n} \equiv -2^k \mod p,$$

(по теореме Ферма $2^{2n} \equiv 1 \mod p$, значит, $2^n \equiv -1 \mod p$) окончательно получим формулу (2.4)

В случае же базиса третьего типа число 2 является квадратичным вычетом по модулю $p$, поэтому все степени $2^k \mod p, k = 1, ..., n-1$ образуют перестановку множества всех квадратичных вычетов по модулю $p$, так как их ровно $n$ штук. Добавив к этому тот факт, что $p$ равно 3 по модулю 4, и поэтому -1 является квадратичным невычетом по модулю $p$, так как в противном случае существовало бы такое число $r$, что $-1 \equiv r^2 \mod p$, а это приводило к противоречию с теоремой Ферма:

$$r^{p-1} = (r^2)^{\frac{p-1}{2}} \equiv -1 \mod p,$$

и тот факт, что произведение вычета на невычет является невычетом, получаем, что последовательность $-2^k \mod p, k = 0, ..., n-1$ образует перестановку множества всех квадратичных невычетов по модулю $p$, в итоге получаем, что и в этом случае верна формула (2.4)

Таким образом, мы получили, что базис $\{a_1, ..., a_n\}$ есть просто перестановка базиса $\{x, ..., x^{2^{n-1}}\}$. Отрицательные индексы можно заменить на соответствующие положительные благодаря равенству $a_i = \zeta^i + \zeta^{-1} = a_{-i}$ по формуле $x^{2^i} = a_{|\pi(i)|}$ (или для координат $x_i = x_{|\pi(i)|'}$. Поэтому далее везде будем считать индексы положительными, и знак модуля числа будем опускать.

Очевидно, что $B'$-тоже базис, так как он есть просто перестановка базиса $B$. Можно считать, что сложность перехода $L_{BB'}(n) = O(n)$, а если рассматривать не программную, а схемную реализацию, то даже $L_{BB'}(n) = 0$.

**Переход от базиса $B'$ к базису $A'$ и преобразование координат $\tilde{x}'$ к координатам $\tilde{y}'$** Согласно теореме (2.4) его сложность оценивается как

$$L(n) \leq \frac{n}{2}\log_2 n + 2n$$

**Переход от базиса $A'$ к базису $A$ и преобразование координат $\tilde{y}'$ к координатам $\tilde{y}$.** По рекуррентной формуле (2.3) можно в явном виде построить минимальный многочлен $m_x$ (он строится лишь однажды, до применения алгоритма перехода, поэтому сложность его построения в сложности алгоритма не учитывается), т.е.

$$m_x = f_n(x) = b_0x^0 + ... + b_nx^n = 0 \tag{2.5}$$

где $x$ - генератор этих базисов, и не все коэффициенты $b_i$ равны нулю. Сделаем явный переход от координат $\tilde{y}'$ в почти стандартном базисе $A'$ к координатам $\tilde{y}$ в стандартном $A$. Учитывая выражение (2.5),

$$\sum_{i=1}^{n} y'_i x^i = \sum_{i=1}^{n-1} y'_i x^i + y'_n \sum_{i=0}^{n-1} b_i x^i,$$

после перегруппировки коэффициентов имеем

$$\sum_{i=1}^{n} y'_i x^i = \sum_{i=1}^{n-1} x^i(y'_i + b_i y'_n) + y'_n x^0 = \sum_{i=1}^{n-1} y_i x^i + y_0 x^0,$$

откуда видно, что сложность перехода от почти стандартного базиса $A'$ к стандартному $A$ оценивается как $L_{A'A}(n) \leq n - 1$.

Итак, мы оценили сложность каждого преобразования, теперь осталось их сложить:

$$L_{BA}(n) = L_{BB'}(n) + L_{B'A'}(n) + L_{A'A}(n) \leq \frac{n}{2}\log_2 n + 3n.$$

□

**Переход от стандартного базиса к нормальному.** Теперь покажем, что теорема верна и при обратном переходе, т.е. от стандартного базиса $A$ к базису второго или третьего типа $B$. Опять определим цепочку переходов, но уже в обратном порядке:

$$A \to A' \to B' \to B.$$

**Переход от базиса $A$ к базису $A'$ и от координат $\tilde{y}$ к $\tilde{y}'$.** Опять же в силу рекуррентной формулы (2.3) у нас есть минимальный многочлен (2.5), из явного вида которого мы можем выразить $x^0$, а именно:

$$x^0 = \sum_{i=1}^{n} b_i x^i \tag{2.6}$$

Сделаем явный переход от координат $\tilde{y}$ в стандартном базисе $A$ к координатам $\tilde{y}'$ в почти стандартном $A'$. Принимая во внимание выражение (2.6), имеем

$$\sum_{i=0}^{n-1} y_i x^i = \sum_{i=1}^{n-1} y_i x^i + y_0 \sum_{i=1}^{n} b_i x^i,$$

после перегруппировки коэффициентов получим

$$\sum_{i=0}^{n-1} y_i x^i = \sum_{i=1}^{n-1} x^i (y_i + b_i y_0) + y_0 x^n = \sum_{i=1}^{n-1} y'_i x^i + y'_n x^n,$$

откуда видно, что сложность перехода от стандартного базиса $A$ к почти стандартному базису $A'$ оценивается как $L_{AA'}(n) \leq n - 1$.

**Переход от базиса $A'$ к базису $B'$ и преобразование координат $\tilde{y}'$ к координатам $\tilde{x}'$.** Вычислим преобразование

$$(x'_1, ..., x'_n) = (F_n^{-1})^T \otimes \begin{pmatrix} y''_1 \\ ... \\ y''_n \end{pmatrix} \tag{2.7}$$

умножения вектора $Y^T$ на матрицу $(F_{2^{k+1}}^{-1})^T$ (обратную к матрице из теоремы (2.4) с помощью (2.3). В этой формуле рекуррентно выполнялись преобразования с векторами $X_1$ и $X_2$, переводящие их в векторы $Y_1 = (y'_1, ..., y''_{2^k})$ и $Y_2 = (y'_{2^k+1}, ..., y'_n)$, составляющие вектор $Y = (y'_1, ..., y'_n)$. Естественно, что обратное преобразование $(F_{n-2^k}^{-1})^T \otimes Y_2^T$ приведет вектор $Y_2$ в вектор $X_2$, а чтобы получить вектор $X_1$, надо сделать преобразование $(F_{2^k}^{-1})^T \otimes Y_1^T$. Так как

$$X_1 = \{x_1, ..., x_{2^{k+1}-n-1}, x_{2^{k+1}-n} + x_n, x_{2^k-1} + x_{2^k+1}, x_{2^k}\}^T,$$
$$X_2 = \{x_{1+2^k}, ..., x_n\}^T,$$

то для восстановления вектора $X = \{x_1, ..., x_n\}$ по этим векторам достаточно к соответствующим $n - 2^k$ компонентам вектора $X_1$ прибавить (по $mod 2$) компоненты вектора $X_2$.

Индуктивно продолжая описанное преобразование, мы перейдем к координатам $\tilde{x}'$ в почти нормальном базисе $B'$ со сложностью перехода

$$L^{-1}(n) \leq n - 2^k + L^{-1}(2^k) + L^{-1}(n - 2^k),$$

и, как раньше, придем к оценке

$$L_{B'A'}(n) \leq \frac{n}{2} \log_2 n + 2n.$$

**Переход от базиса $B'$ к базису $B$ и координат $\tilde{x}'$ к $\tilde{x}$.** Переход получается посредством обратной перестановки к $\pi(i)$, а его сложность оценивается, как и раньше, т.е. $L_{B'B}(n) = 0$.

В результате имеем

$$L_{AB}(n) = L_{AA'}(n) + L_{A'B'}(n) + L_{B'B}(n) \leq \frac{n}{2}\log_2 n + 3n.$$

Заметим, что при вычислении обратного преобразования $(F_n^{-1})^T$ при $n = 2^{k+1} - 1$ мы фактически получили матричное тождество

$$(F_n^T)^{-1} = \begin{pmatrix} (F_m^T)^{-1} & o_m & G_m \\ 0...0 & 1 & 0...0 \\ O_m & o_m & (F_m^T)^{-1} \end{pmatrix} \tag{2.8}$$

где $m = (n-1)/2$, а матрица $G_m = I_m(F_m^T)^{-1}$ есть симметричное отражение матрицы $(F_m^T)^{-1}$ относительно средней строки. Это тождество, так же как и аналогичное тождество

$$F_n^{-1} = \begin{pmatrix} F_m^{-1} & o_m & O_m \\ 0...0 & 1 & 0...0 \\ G_m & o_m & F_m^{-1} \end{pmatrix} \tag{2.9}$$

(здесь $m = (n-1)/2$), а матрица $G_m = F_m^{-1}I_m$ есть симметричное отражение матрицы $F_m^{-1}$ относительно среднего столбца) можно проверить и непосредственно. С помощью этих тождеств, так же как и в теореме (2.3), можно доказать, что плотность обратной матрицы $S(F_n^{-1}) = O(n^{\log_2 3})$ и такую же плотность имеет матрица перехода от нормального базиса второго или третьего типа к стандартному базису поля $GF(2^n)$.

### 2.4.3 Алгоритм перехода от оптимального нормального базиса 2 или 3 типа к стандартному.

**Алгоритм 6** OptimalToStandard

**Вход:** степень $n$ расширения поля $GF(2)$, вектор $(b_1, ..., b_n) \in GF(2^n)$ координат элемента, представленного в ОНБ второго или третьего типа, $P$ - вектор коэффициентов минимального многочлена.

**Выход:** вектор $a = (a_0, ..., a_n - 1) \in GF(2^n)$ координат того же элемента, представленного в стандартном базисе с тем же генератором.

**Шаги:**

1. Переход от оптимального нормального базиса $B$ второго или третьего типа к почти нормальному $B'$:
   для $j = 0, ..., n-1 c_j = b_{\pi(j)}$, где

$$\pi(j) = \begin{cases} (2^j \mod p), & \text{если } (2^j \mod p) \le n \\ p - (2^j \mod p), & \text{если } (2^j \mod p) > n \end{cases}$$

2. Добавить к полученному вектору справа $2^{r-1} - n$ нулей, где $r$ определяется из неравенства $2^{r-1} < n \le 2^r$. Таким образом получить вектор $X$ длины $2^r$.

3. Переход от почти нормального базиса $B'$ к почти стандартному $S'$.
   Для $i = 1, ..., r-1$ :
   для $j = 0, ..., 2^{i-1} - 1$ :

$$X[j \cdot 2^{r-i+1}, 2^{r-1} + j \cdot 2^{r-i+1} - 1 =$$
$$= F(X[j * 2^{r-i+1}, 2^{r-1} + j \cdot 2^{r-i+1} - 1], X[2^{r-1} + j \cdot 2^{r-i+1}, (j+1) \cdot 2^{r-i+1} - 1]),$$

   где $F(X_1, X_2) = (X_1 \oplus \widetilde{\widetilde{X_2}}, X_2)$, где $\widetilde{\widetilde{Y}} = \overline{Y \gg 1}$ - сдвиг на один разряд вправо с потерей старшего разряда и обращение порядка следования бит полученного вектора.

4. Переход от почти стандартного базиса $S'$ к стандартному $S$:
   если $deg\ a = deg\ P$, то $A = X \oplus P$.

5. Вернуть $a$.

### 2.4.4 Алгоритм перехода от стандартного базиса к оптимальному нормальному базису второго типа

**Алгоритм 7** StandardToONB2

**Вход:** Степень $n$ расширения поля GF(2), вектор $a = (a_0, ..., a_{n-1}) \in GF(2^n)$ координат элемента, представленного в стандартном базисе, $P$ - вектор коэффициентов минимального многочлена.

**Выход:** вектор $(b_1, ..., b_n) \in GF(2^n)$ координат того же элемента, представленного в ОНБ 2 типа с тем же генератором, или сообщение "ОНБ второго типа не существует".

**Шаги:**

1. Вычислить $p = 2n + 1$, проверить условия ОНБ второго типа:
   a) $p$ - простое число;
   b) 2 является примитивным корнем из 1 по модулю $p$: для каждого простого делителя $\delta$ числа $p - 1$ выполняется $2^{\frac{p-1}{\delta}} \neq 1$. Если хотя бы одно из этих условий не выполняется, то вернуть сообщение ОНБ 2 типа не существует.

2. Переход от стандартного базиса $S$ к почти стандартному $S'$: если $a_0 = 1, C = a \oplus P$, иначе $C = a$

3. Добавить к полученному вектору справа $2^r - n$ нулей, где $r$ определяется из неравенства $2^{r-1} < n \leq 2^r$. Таким образом получить вектор $X$ длины $2^r$.

4. Переход от почти стандартного базиса $S'$ к почти нормальному $B'$.
   Для $i = r - 1, ..., 1$:
   для $j = 0, ..., 2^{i-1} - 1$ :

$$X[j \cdot 2^{r-i+1}, 2^{r-i} + j \cdot 2^{r-i+1} - 1] =$$
$$= F(X[j \cdot 2^{r-i+1}, 2^{r-i} + j \cdot 2^{r-i+1} - 1], X[2^{r-i} + j \cdot 2^{r-i+1}, (j+1)2^{r-i+1} - 1]).$$

   где $F(X_1, X_2) = (X_1 \oplus \widetilde{\widetilde{X_2}}, X_2)$, где $\widetilde{\widetilde{Y}} = \overline{Y \gg 1}$ - сдвиг на один разряд вправо с потерей старшего разряда и обращение порядка следования бит полученного вектора.

5. Переход от почти нормального базиса $B'$ к нормальному $B$:
   Для $j = 1, ..., n b_j = b'_{\pi^{-1}(j)}$, где $\pi^{-1}(j)$ - обратная к $\pi(j)$ (вычисленной в предыдущем алгоритме перестановка).

6. Вернуть $(b_1, ..., b_n)$.

### 2.4.5 Алгоритм перехода от стандартного базиса к оптимальному нормальному базису третьего типа

От алгоритма перехода к ОНБ 2 типа этот алгоритм отличается лишь условием существования базиса.

**Алгоритм 8** StandardToONB3

**Вход:** Степень $n$ расширения поля GF(2), вектор $a = (a_0, ..., a_{n-1}) \in GF(2^n)$ координат элемента, представленного в стандартном базисе, $P$ - вектор коэффициентов минимального многочлена.

**Выход:** вектор $(b_1, ..., b_n) \in GF(2^n)$ координат того же элемента, представленного в ОНБ 3 типа с тем же генератором, или сообщение "ОНБ третьего типа не существует".

**Шаги:**

1. Вычислить $p = 2n + 1$, проверить условия ОНБ третьего типа:
   a) $p$ - простое число;
   b) $2^n = 1 \mod p$;
   c) Для каждого простого делителя $\delta$ числа $p - 1$ выполняется $2^{\frac{n}{\delta}} \mod p \neq 1$. Если хотя бы одно из этих условий не выполняется, то вернуть сообщение ОНБ 3 типа не существует.

2. Переход от стандартного базиса $S$ к почти стандартному $S'$: если $a_0 = 1, C = a \oplus P$, иначе $C = a$

3. Добавить к полученному вектору справа $2^r - n$ нулей, где $r$ определяется из неравенства $2^{r-1} < n \leq 2^r$. Таким образом получить вектор $X$ длины $2^r$.

4. Переход от почти стандартного базиса $S'$ к почти нормальному $B'$.
   Для $i = r - 1, ..., 1$:
   для $j = 0, ..., 2^{i-1} - 1$ :

$$X[j \cdot 2^{r-i+1}, 2^{r-i} + j \cdot 2^{r-i+1} - 1] =$$
$$= F(X[j \cdot 2^{r-i+1}, 2^{r-i} + j \cdot 2^{r-i+1} - 1], X[2^{r-i} + j \cdot 2^{r-i+1}, (j+1)2^{r-i+1} - 1]).$$

   где $F(X_1, X_2) = (X_1 \oplus \widetilde{\widetilde{X_2}}, X_2)$, где $\widetilde{\widetilde{Y}} = \overline{Y \gg 1}$ - сдвиг на один разряд вправо с потерей старшего разряда и обращение порядка следования бит полученного вектора.

5. Переход от почти нормального базиса $B'$ к нормальному $B$:
   Для $j = 1, ..., n b_j = b'_{\pi^{-1}(j)}$, где $\pi^{-1}(j)$ - обратная к $\pi(j)$ (вычисленной в предыдущем алгоритме перестановка).

6. Вернуть $(b_1, ..., b_n)$.

# 3 Реализация

## 3.1 Результаты оптимизации генерации простых чисел

Был реализован алгоритм предобработки числа для теста Миллера-Рабина. Были проверены различные размеры фактор-множества. В зависимости от его, время выполнения теста было различным. Желаемая вероятность простоты была фиксирована. В итоге получили следующий результат:

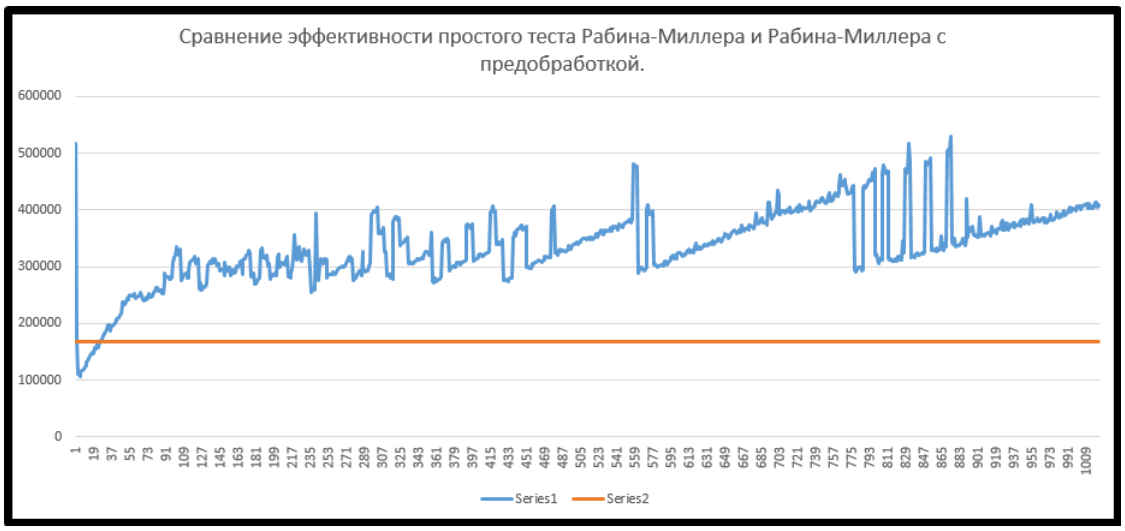


Рисунок 3.1 — График зависимости времени проверки числа на простоту в зависимости от размерности фактор-множества

На рисунке (3.1) изображен график зависимости времени проверки числа на простоту в зависимости от размерности фактор-множества. Число для генерации и последующей проверки на простоту имеет размер не более 32 бит.

На рисунке (3.2) изображен тот же график, что и на рисунке (3.1), только в большем масштабе. На нем отлично видно, что в точке 5 (соответствует мощности фактор-множества 6) время выполнения теста с предобработкой минимально. Таким образом можем заключить, что метод себя оправдывает при небольшой мощности фактор-множества.

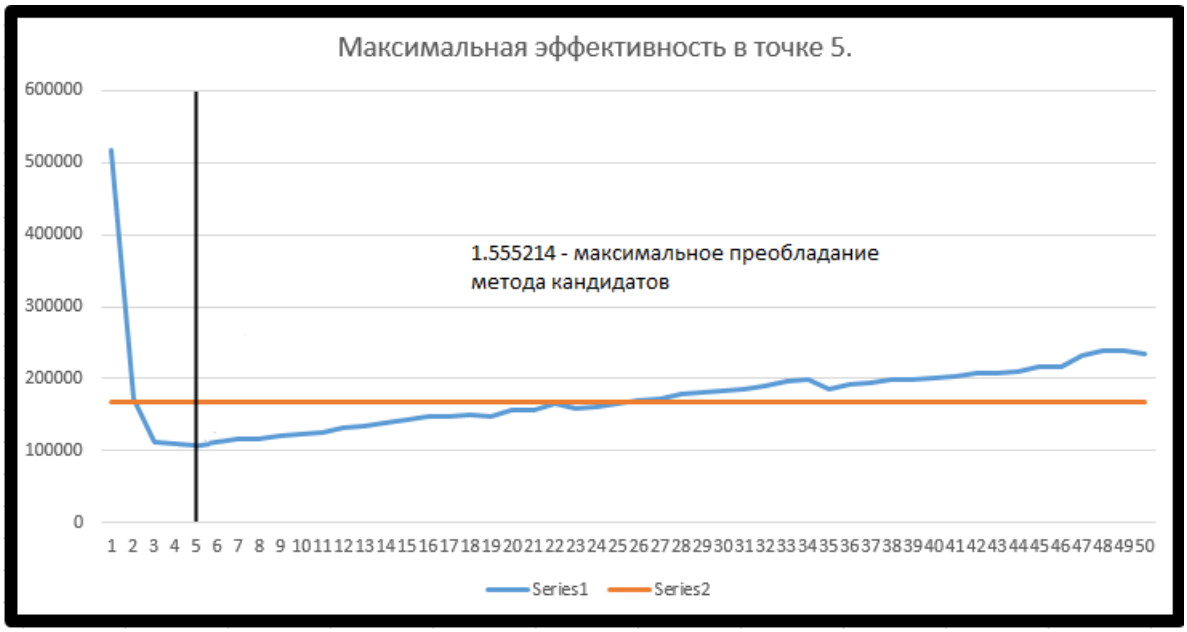


Рисунок 3.2 — График зависимости времени проверки числа на простоту в зависимости от размерности фактор-множества

При этом в алгоритме Миллера-Рабина использованы базовые числа, которые позволяют точно проверять на простоту числа вплоть до 4759123141. В результате получили, что заранее рассчитав таблицу фактор-множеств для соответствующих размерностей для генерации простого числа, при заданной вероятности можно сильно ускорить проверку генерированного числа на простоту.

## 3.2 Результаты оптимизации арифметики в ОНБ

В полях характеристики 2 типа были реализованы следующие арифметические операции в оптимальных нормальных базисах: генерация минимального многочлена, генерация таблицы умножения, умножение, поиск обратного элемента, возведение в квадрат, деление. Имплементированные алгоритмы были протестированы на процессоре Intel Core i5-3230M 2.6 GHz. Для тестирования использовались размеры полей, которые рекомендованы стандартом ДСТУ 4145-2002 для реализации арифметики в оптимальных нормальных базисах. В графиках ниже по оси $X$ будет располагаться степень расширения $n$ поля $GF(2^n)$, по оси $Y$ – среднее число тактов процессора для выполнения операции. Каждая операция была протестирована 1000 раз для получения большей точности измерения.

### 3.2.1 Генерация параметров

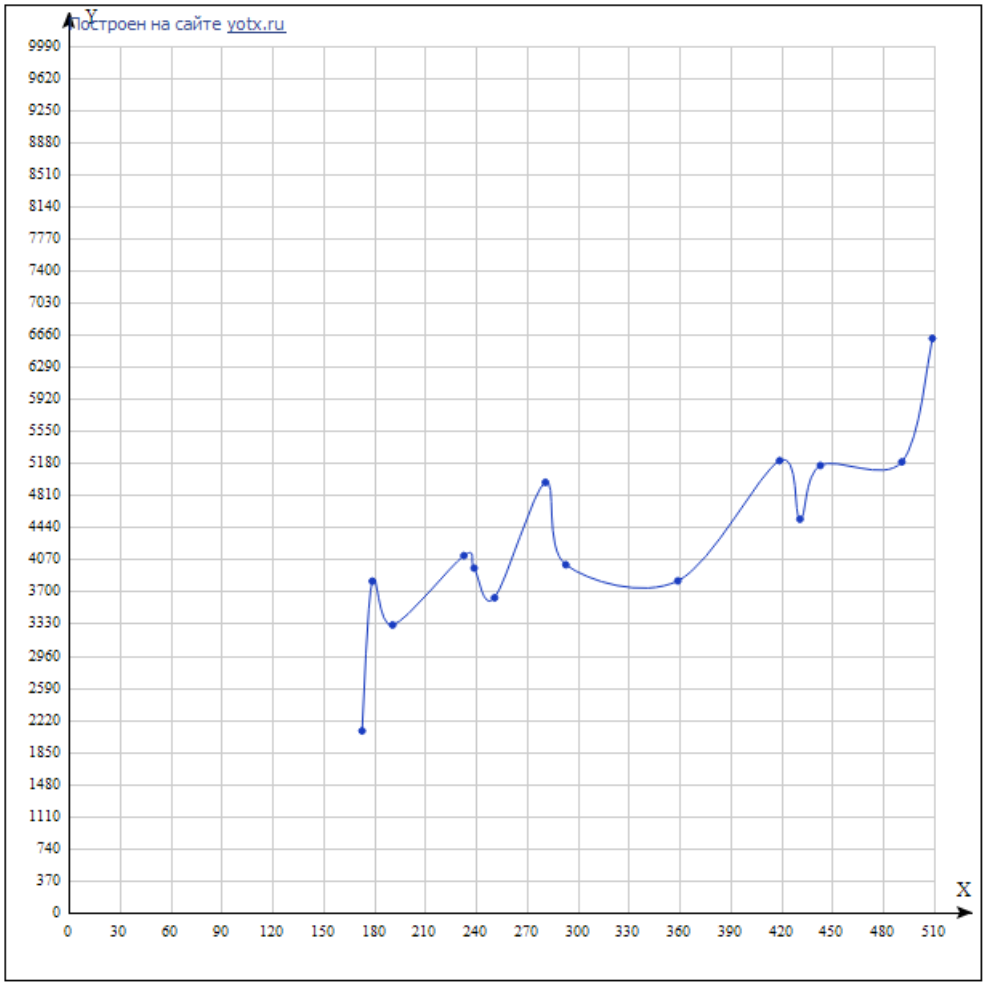


Рисунок 3.3 — График зависимости времени генерации минимального многочлена в зависимости от размера поля

На рисунке (3.3) изображен график зависимости времени генерации минимального многочлена в зависимости от размера поля. По теоретической оценке алгоритм генерации минимального многочлена имеет сложность $O(n)$.

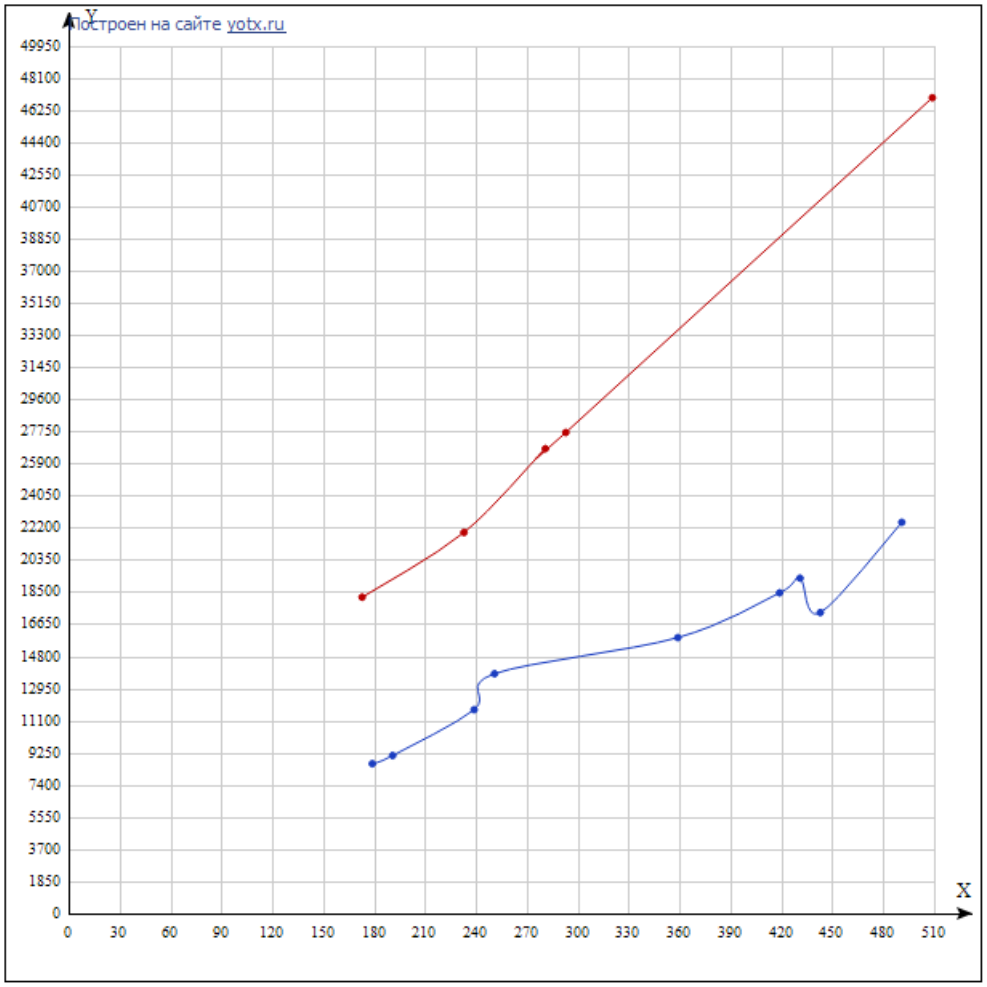


Рисунок 3.4 — График зависимости времени генерации таблицы умножения в зависимости от размера поля

На рисунке (3.4) изображен график зависимости времени генерации таблицы умножения в соответствующем ОНБ в зависимости от размера поля. По теоретической оценке алгоритм генерации таблицы умножения имеет сложность $O(n)$. Верхний график на этом рисунке соответствует генерации таблицы умножения для ОНБ 2 типа, нижний график – генерации таблицы умножения для ОНБ 3 типа. Разница в скорости работы алгоритмов заключается в некоторых дополнительных операциях при генерации базиса второго типа. Но на линейность эти дополнительные операции не влияют.

### 3.2.2 Умножение

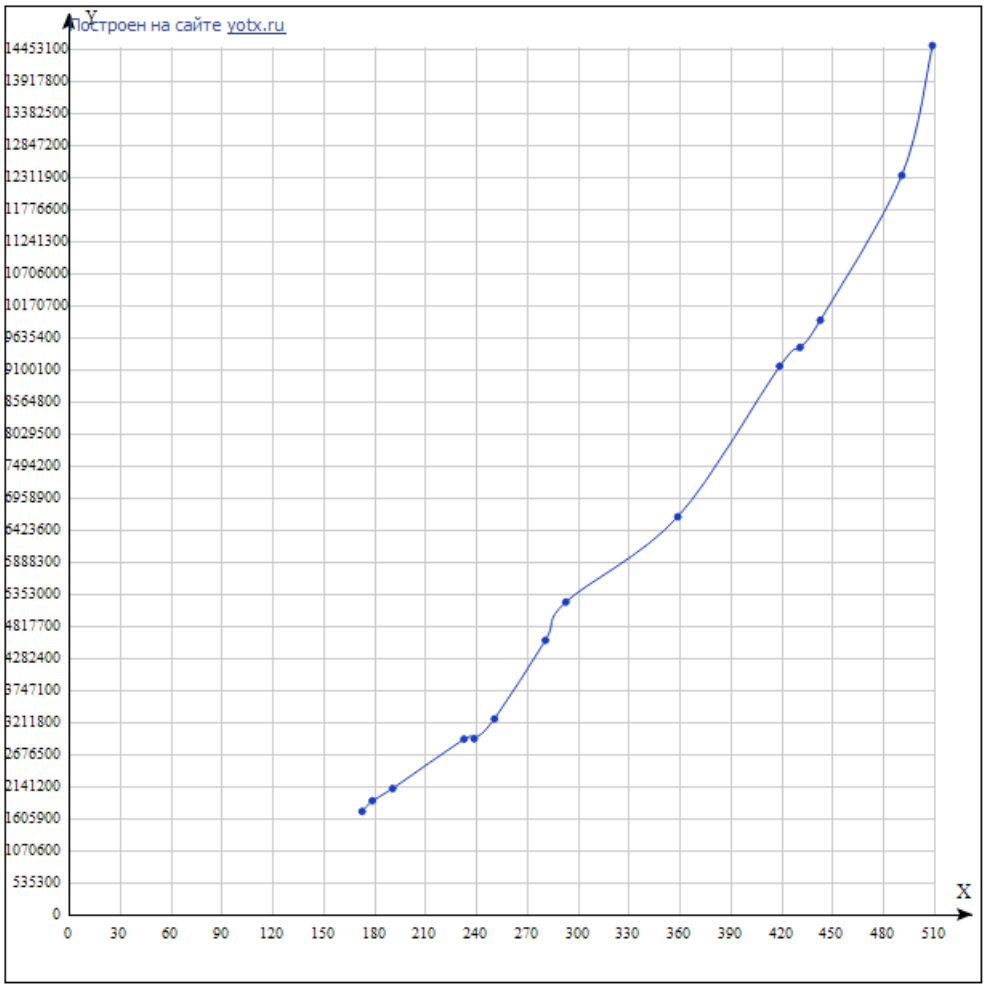


Рисунок 3.5 — График зависимости времени умножения в ОНБ в зависимости от размера поля

На рисунке (3.5) изображен график зависимости времени умножения в ОНБ в зависимости от размера поля. По теоретической оценке алгоритм генерации таблицы умножения имеет сложность $O(n^2)$.

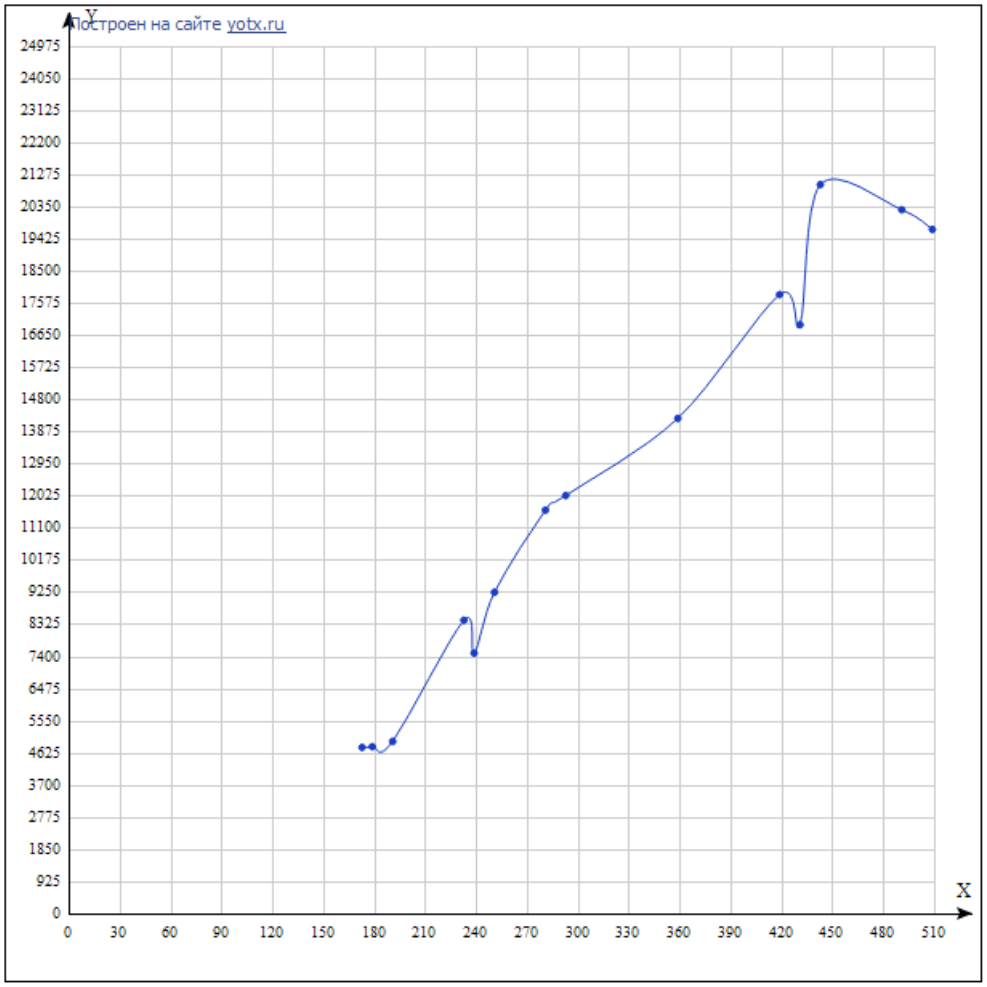


Рисунок 3.6 — График зависимости времени умножения в стандартном базисе в зависимости от размера поля

На рисунке (3.6) изображен график зависимости времени умножения в стандартном базисе в зависимости от размера поля. Этот алгоритм взят из библиотеки bee2 (polyMulMod) для сравнения времени работы с алгоритмом умножения в ОНБ. Время его работы имеет меньшую, чем $O(n^2)$ сложность, и умножение в стандартном базисе в итоге получилось гораздо более быстрым, нежели умножение в ОНБ.

### 3.2.3 Возведение в квадрат

Возведение в квадрат – важный алгоритм для реализации возведения в степень элемента поля. Так как возведение в степень в полях характеристики 2 требует наличия двух операций (возведение в квадрат, сложение по модулю 2), и так как операция сложения по модулю 2 является очень быстрой, то от скорости возведения в квадрат напрямую зависит скорость возведения в степень.

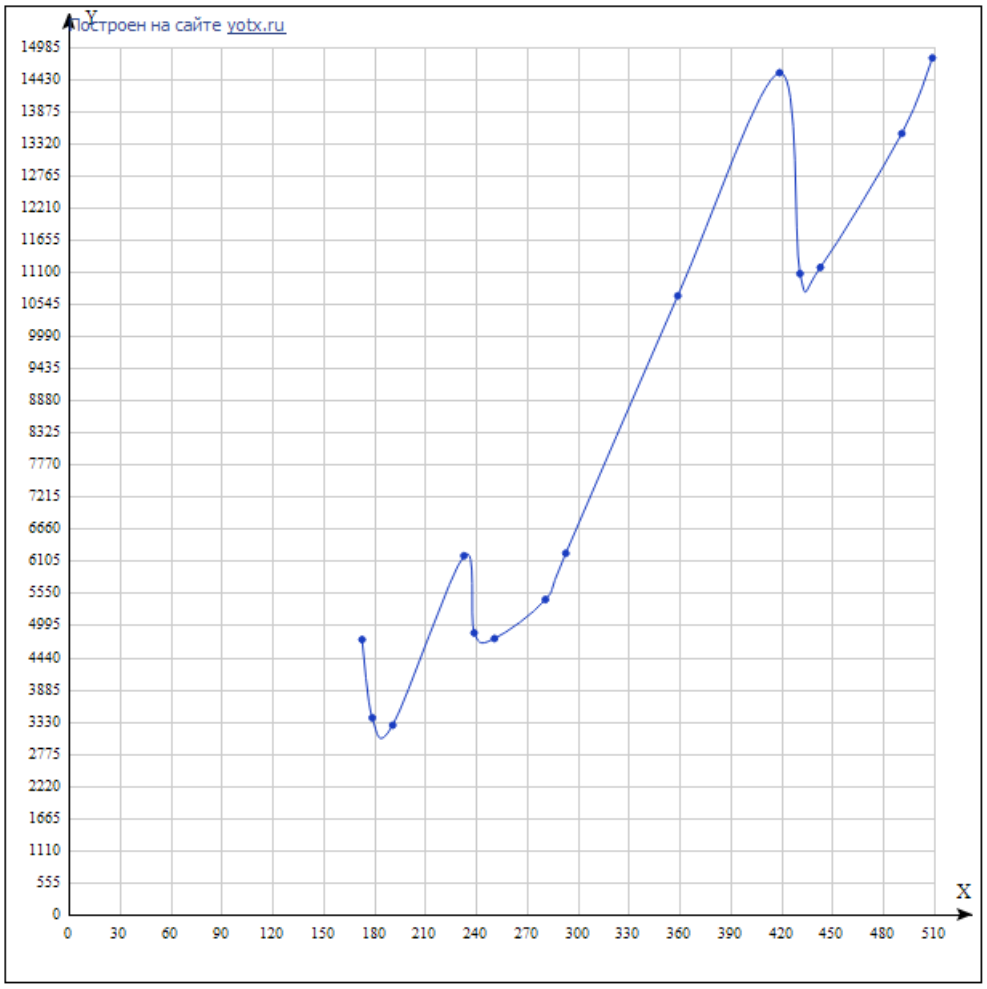


Рисунок 3.7 — График зависимости времени возведения в квадрат в стандартном базисе в зависимости от размера поля

На рисунке (3.7) изображен график зависимости времени возведения в квадрат в стандартном базисе в зависимости от размера поля Этот алгоритм взят из библиотеки bee2 (polySqrMod) для сравнения скорости работы с возведением в квадрат в ОНБ.

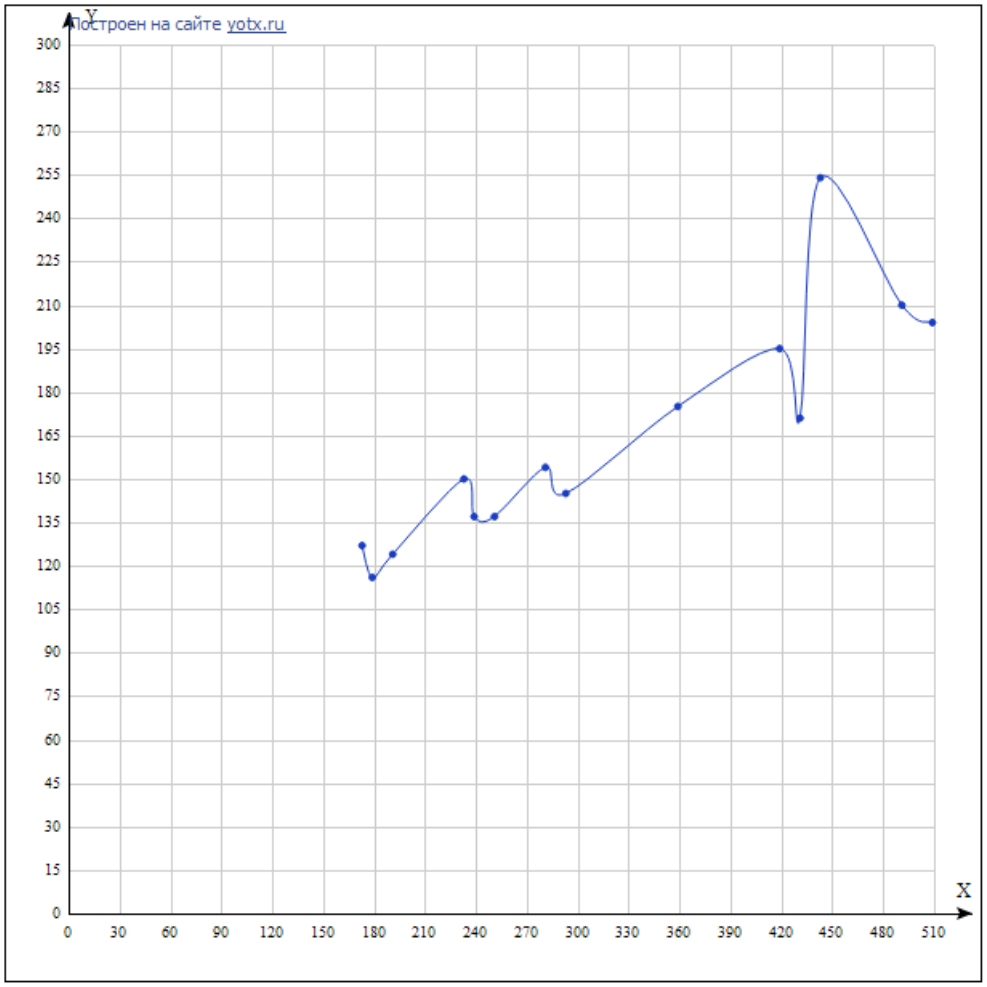


Рисунок 3.8 — График зависимости времени возведения в квадрат в оптимальном нормальном базисе в зависимости от размера поля

На рисунке (3.8) изображен график зависимости времени возведения в квадрат в ОНБ в зависимости от размера поля. Алгоритм возведения в квадрат в оптимальном нормальном базисе - это циклический сдвиг на 1 враво. Этот алгоритм оказался гораздо быстрее возведения в квадрат в стандартном базисе, из чего можно сделать вывод, что и возведение в степень в ОНБ будет также гораздо быстрее, чем в стандартном базисе.

### 3.2.4 Переходы между базисами

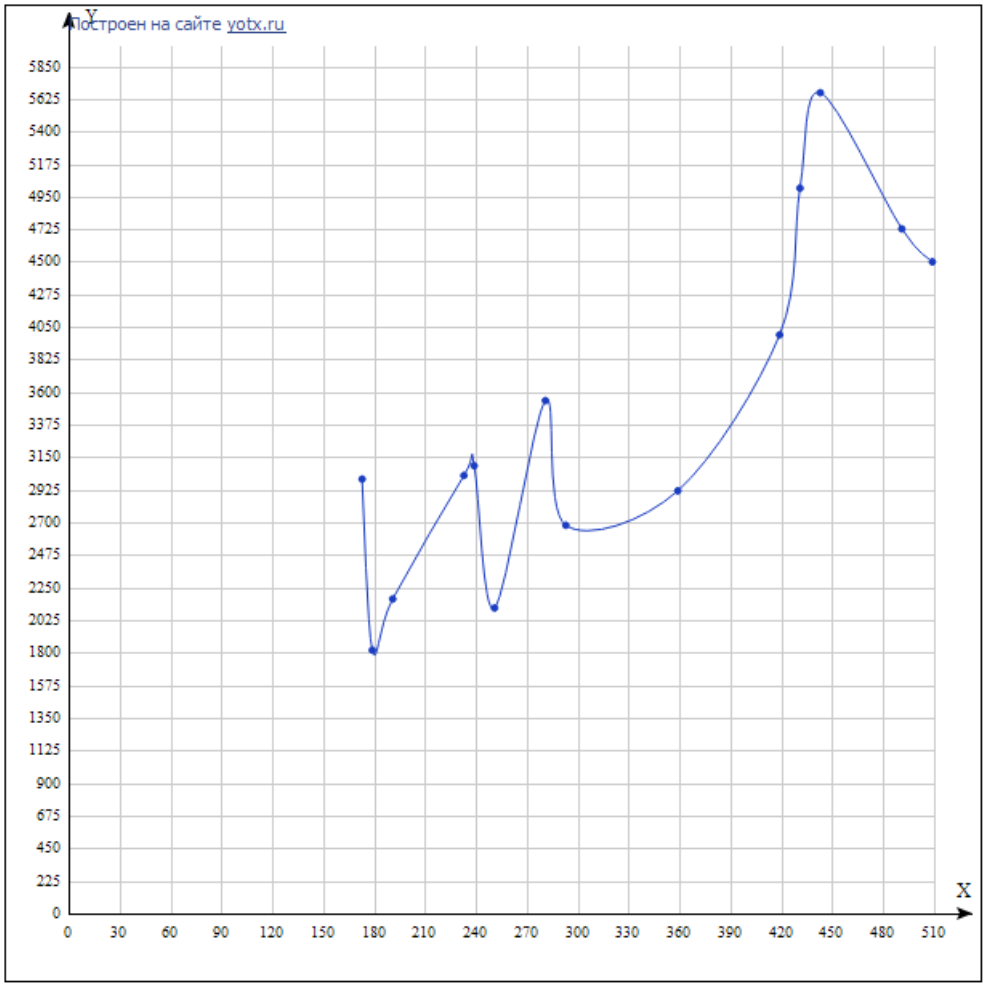


Рисунок 3.9 — График зависимости времени перехода из ОНБ в стандартный базис в зависимости от размера поля

На рисунке (3.9) изображен график зависимости времени перехода из ОНБ в стандартный базис в зависимости от размера поля. Алгоритмы перехода между ОНБ 2 и 3 типа и стандартным базисом имеют сложность $O(n \log n)$. Исходя из сложности алгоритма перехода между базисами можно сказать, а также полученных результатов измерений, можно получить, что умножение в оптимальном нормальном базисе всегда выгоднее проводить, сделав переход в стандартный базис, произведя там умножение, и вернувшись в ОНБ.

# Заключение

В работе получены следующие основные результаты:
1. В ходе работы реализованы алгоритмы арифметики в оптимальных нормальных базисах полей характеристики 2, алгоритм поиска оптимального размера фактор-множества для генерации простых чисел.
2. Проведен анализ эффективности реализованных алгоритмов для полей, заданных стандартом ДСТУ 4145-2002.
4. Эксперименты показали, что использование оптимальных нормальных базисов выгодно только вместе с использованием стандартных базисов. Для умножения, деления, обращения необходимо пользоваться стандартным базисом. Для возведения в степень необходимо использовать оптимальный нормальный базис.
5. Разработано программное обеспечение на языке программирования C, реализующее все разработанные алгоритмы.

## Литература


1. Кочергин В.В. Об аддитивных цепочках векторов, вентильных схемах и сложности вычисления степеней. / В.В. Кочергин, С.Б. Гашков. // Дискретная математика. – 1992. – № 52. – С.22–40.

2. Фролов А.Б. Элементарное введение в эллиптическую криптографию. Алгебраические и алгоритмические основы. / А.Б. Фролов, А.А. Болотов, С.Б. Гашков. – Москва: КомКнига, 2006. – 324 с.

3. Фролов А.Б. Элементарное введение в эллиптическую криптографию. Протоколы криптографии на эллиптических кривых. / А.Б. Фролов, А.А. Болотов, С.Б. Гашков. – Москва: КомКнига, 2006. – 280 с.

4. Информационные технологии. Криптографическая защита информации. Цифровая подпись, основывающаяся на эллиптических кривых. Формирование и проверка. ДСТУ 4145–2002: ГОСУДАРСТВЕННЫЙ СТАНДАРТ УКРАИНЫ – 2002. – Департамент специальных телекоммуникационных систем и защиты информации СБ Украины. – 42 с.

5. Lenstra Jr. H.W. Optimal normal bases. / H.W Lenstra Jr, S Gao. // Designs, Codes and Cryptography – 1992. – № 22. – P.149–161.

6. Mullin R.C, Optimal normal bases in gf(pn). / R.C. Mullin, I.M. Onyszchuk, S.A. Vanstone, and R.M. Wilson. // Discrete Applied Mathematics – 1988-1989. – № 22(2). – P.149 – 161.

7. Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA): – X9.62-1998 – 20.09.1998. – American National Standards Institute. – 119 p.

# ПРИЛОЖЕНИЕ А

Здесь приводится код, непосредственно реализующий рассматриваемые алгоритмы.

*Код разработанной библиотеки.*

```
#include "onb.h"
#include "utils.h"

static octet f[256] = {
  0x00, 0x01, 0x02, 0x03, 0x05, 0x04, 0x07, 0x06, 0x08, 0x09, 0x0A, 0x0B, 0x0D,
      0x0C, 0x0F, 0x0E,
  0x15, 0x14, 0x17, 0x16, 0x10, 0x11, 0x12, 0x13, 0x1D, 0x1C, 0x1F, 0x1E, 0x18,
      0x19, 0x1A, 0x1B,
  0x22, 0x23, 0x20, 0x21, 0x27, 0x26, 0x25, 0x24, 0x2A, 0x2B, 0x28, 0x29, 0x2F,
      0x2E, 0x2D, 0x2C,
  0x37, 0x36, 0x35, 0x34, 0x32, 0x33, 0x30, 0x31, 0x3F, 0x3E, 0x3D, 0x3C, 0x3A,
      0x3B, 0x38, 0x39,
  0x51, 0x50, 0x53, 0x52, 0x54, 0x55, 0x56, 0x57, 0x59, 0x58, 0x5B, 0x5A, 0x5C,
      0x5D, 0x5E, 0x5F,
  0x44, 0x45, 0x46, 0x47, 0x41, 0x40, 0x43, 0x42, 0x4C, 0x4D, 0x4E, 0x4F, 0x49,
      0x48, 0x4B, 0x4A,
  0x73, 0x72, 0x71, 0x70, 0x76, 0x77, 0x74, 0x75, 0x7B, 0x7A, 0x79, 0x78, 0x7E,
      0x7F, 0x7C, 0x7D,
  0x66, 0x67, 0x64, 0x65, 0x63, 0x62, 0x61, 0x60, 0x6E, 0x6F, 0x6C, 0x6D, 0x6B,
      0x6A, 0x69, 0x68,
  0x80, 0x81, 0x82, 0x83, 0x85, 0x84, 0x87, 0x86, 0x88, 0x89, 0x8A, 0x8B, 0x8D,
      0x8C, 0x8F, 0x8E,
  0x95, 0x94, 0x97, 0x96, 0x90, 0x91, 0x92, 0x93, 0x9D, 0x9C, 0x9F, 0x9E, 0x98,
      0x99, 0x9A, 0x9B,
  0xA2, 0xA3, 0xA0, 0xA1, 0xA7, 0xA6, 0xA5, 0xA4, 0xAA, 0xAB, 0xA8, 0xA9, 0xAF,
      0xAE, 0xAD, 0xAC,
  0xB7, 0xB6, 0xB5, 0xB4, 0xB2, 0xB3, 0xB0, 0xB1, 0xBF, 0xBE, 0xBD, 0xBC, 0xBA,
      0xBB, 0xB8, 0xB9,
  0xD1, 0xD0, 0xD3, 0xD2, 0xD4, 0xD5, 0xD6, 0xD7, 0xD9, 0xD8, 0xDB, 0xDA, 0xDC,
      0xDD, 0xDE, 0xDF,
  0xC4, 0xC5, 0xC6, 0xC7, 0xC1, 0xC0, 0xC3, 0xC2, 0xCC, 0xCD, 0xCE, 0xCF, 0xC9,
      0xC8, 0xCB, 0xCA,
  0xF3, 0xF2, 0xF1, 0xF0, 0xF6, 0xF7, 0xF4, 0xF5, 0xFB, 0xFA, 0xF9, 0xF8, 0xFE,
      0xFF, 0xFC, 0xFD,
  0xE6, 0xE7, 0xE4, 0xE5, 0xE3, 0xE2, 0xE1, 0xE0, 0xEE, 0xEF, 0xEC, 0xED, 0xEB,
      0xEA, 0xE9, 0xE8};

// Возведение в квадрат в оптимальном нормальном базисе
void sqr(word *res, word *a, size_t n, size_t m)
{
  wordCopy(res, a, n);
  shiftRight(res, n, m, 1);
}

void generate_b(word *b, size_t m)
{
  size_t index = 0, k, bit_size = bits_in_number(m);
  word M, N;
  BOOL res;
  for (index = 0; index < m + 1; ++index) {
    k = (index + m) / 2;
```

```
    M = index;
    N = k;
    M = ~M;
    M = ~(M | N);
    normalize(&M, 1, bit_size);
    res = (M == 0) ? TRUE : FALSE;
    wordSetBit(b, index, res);
  }
  normalize(b, bit_size, m + 1);
}

void generate_pi(word *pi, size_t m)
{
  size_t index = 0;
  size_t p = 2 * m + 1;
  pi[0] = 1;
  for (index = 1; index < m; ++index) {
    pi[index] = (pi[index-1] * 2) % p;
    if (pi[index] > m) {
      pi[index] = p - pi[index];
    }
    pi[index - 1] -= 1;
  }
  pi[m - 1] -= 1;
}

void applyPi(word *a, word *b, word *pi, size_t m)
{
  size_t index = 0;
  BOOL bit = FALSE;
  for (index = 0; index < m; ++index) {
    bit = wordGetBits(a, index, 1) == 1 ? TRUE : FALSE;
    wordSetBit(b, pi[index], bit);
  }
}

void applyFToWord(word *a)
{
  octet *b, *c, carry = 0;
  b = (octet *)a;
  c = (octet *)malloc(2 * sizeof(octet));

  //apply f to 4 octets
  c[0] = reverseOctet(b[3]);
  c[1] = reverseOctet(b[2]);
  c[0] <<= 1;
  carry = c[1] & 0x80;
  c[1] <<= 1;
  c[0] += carry;
  b[0] ^= reverseOctet(c[0]);
  b[1] ^= reverseOctet(c[1]);

  //apply f to 2,2 octets
  c[0] = reverseOctet(b[1]);
  c[0] <<= 1;
  b[0] ^= reverseOctet(c[0]);

  c[1] = reverseOctet(b[3]);
  c[1] <<= 1;
  b[2] ^= reverseOctet(c[1]);
```

```
  //apply f to 1,1,1,1 octets
  b[0] = f[b[0]];
  b[1] = f[b[1]];
  b[2] = f[b[2]];
  b[3] = f[b[3]];

  free(c);
}

void applyF(word *a, word *mem, size_t n)
{
  size_t index = 0, m = n / 2;
  if (m > 1) {
    wordCopy(mem, a + m, m);
    reverse(mem, mem, m);
    wordShHi(mem, m, 1);
    for (index = 0; index < m; ++index) {
      a[index] ^= mem[index];
    }
    applyF(a, mem, m);
    applyF(a + m, mem, m);
  }
  else {
    applyFToWord(a);
  }
}

void generateONB2_A(word *a, size_t m)
{
  size_t p = 2 * m + 1;
  size_t ksigma, kmu;
  word *pi = (word *)malloc(p * sizeof(word));
  word *pi_inv = (word *)malloc(p * sizeof(word));
  word *sigma = (word *)malloc((p - 1) * sizeof(word));
  word *mu = (word *)malloc((p - 1) * sizeof(word));
  size_t index;
  pi[0] = 1;
  pi[p-1] = 1;
  pi_inv[0] = WORD_MAX;
  pi_inv[1] = 0;
  for (index = 1; index < p - 1; ++index) {
    pi[index] = (2 * pi[index - 1]) % p;
    pi_inv[pi[index]] = index;
  }
  sigma[0] = 1;
  mu[0] = WORD_MAX;
  for (index = 1; index < p; ++index) {
    if (index > 0) {
      ksigma = pi_inv[index - 1];
      if (ksigma != WORD_MAX && ksigma < m) {
        sigma[ksigma] = pi_inv[1+pi[ksigma]] % m;
      }
      else if (ksigma < m) {
        sigma[ksigma] = WORD_MAX;
      }
    }
    if (index < p - 1) {
      kmu = pi_inv[index + 1];
      if (kmu != WORD_MAX && kmu < m) {
```

```
        mu[kmu] = pi_inv[-1+pi[kmu]] % m;
      }
      else if (kmu < m) {
        mu[kmu] = WORD_MAX;
      }
    }
  }
  a[0] = sigma[0];
  for (index = 1; index < m; ++index) {
    a[index * 2 - 1] = sigma[index];
    a[index * 2] = mu[index];
  }
  free(sigma);
  free(mu);
  free(pi);
  free(pi_inv);
}

void generateONB3_A(word *a, size_t m)
{
  size_t p = 2 * m + 1;
  size_t ksigma, kmu;
  word *pi = (word *)malloc(p * sizeof(word));
  word *pi_inv = (word *)malloc(p * sizeof(word));
  word *sigma = (word *)malloc((p - 1) * sizeof(word));
  word *mu = (word *)malloc((p - 1) * sizeof(word));
  size_t index;
  pi[0] = 1;
  pi[p-1] = 1;
  for (index = 0; index < p; ++index) {
    pi_inv[index] = WORD_MAX;
  }
  pi_inv[1] = 0;
  for (index = 1; index < m; ++index) {
    pi[index] = (2 * pi[index - 1]) % p;
    pi[index + m];
    pi_inv[pi[index]] = index;
  }
  sigma[0] = 1;
  mu[0] = WORD_MAX;

  for (index = 1; index < m; ++index) {
    ksigma = pi[index] + 1;
    if (pi_inv[ksigma] == WORD_MAX) {
      ksigma = p - pi[index] - 1;
    }
    sigma[index] = pi_inv[ksigma];
    kmu = pi[index] - 1;
    if (pi_inv[kmu] == WORD_MAX) {
      kmu = p - pi[index] + 1;
    }
    mu[index] = pi_inv[kmu];
  }
  a[0] = sigma[0];
  for (index = 1; index < m; ++index) {
    a[index * 2 - 1] = sigma[index];
    a[index * 2] = mu[index];
  }
  free(sigma);
  free(mu);
```

```
  free(pi);
  free(pi_inv);
}

void mul(word *res, word *a, word *b, word *A, size_t n, size_t m)
{
  size_t step = 0;
  size_t p = 2 * m - 1;
  size_t index = 0;
  size_t apos;
  size_t bpos;
  size_t aposm, bposm;
  BOOL xi, yj, sum;
  BOOL bit = FALSE;
  wordSetZero(res, n);
  for (step = 0; step < m; ++step) {
    bit = FALSE;
    for (index = 0; index < p; ++index) {
      aposm = ((index + 1) >> 1) + step;
      bposm = A[index] + step;
      apos = aposm > m ? aposm - m: aposm;
      bpos = bposm > m ? bposm - m : bposm;
      xi = wordTestBit(a, apos);
      yj = wordTestBit(b, bpos);
      bit = bit ^ (xi && yj);
    }
    wordSetBit(res, step, bit);
  }
}

void inv(word *res, word *a, word *A, size_t n, size_t m)
{
  word *stack1 = (word *)malloc(n * sizeof(word));
  word degree = m - 1;
  size_t bitSize = wordBitSize(&degree, 1);
  size_t index;
  sqr(stack1, a, n, m);
  for (index = bitSize - 1; index > 0; --index) {
    if (wordGetBits(&degree, index - 1, 1) & 1) {
      mul(res, a, stack1, A, n, m);
    }
    else {
      wordCopy(res, stack1, n);
    }
    sqr(stack1, res, n, m);
  }
  free(stack1);
}

void div_onb(word *res, word *a, word *b, word *A, size_t n, size_t m)
{
  word *b_inv = (word *)malloc(n * sizeof(word));
  inv(b_inv, b, A, n, m);
  mul(res, a, b_inv, A, n, m);
  free(b_inv);
}

void fromONB2ToStandard(word *onb2, word *st, word *b, word *pi, size_t m)
{
```

```
  size_t ext = (next_power_of_two(m) > B_PER_W) ? next_power_of_two(m) : B_PER_W
      ;
  word *mem = (word *)malloc(ext);
  wordSetZero(st, ext / B_PER_W);
  applyPi(onb2, st, pi, m);
  applyF(st, mem, ext / B_PER_W);
  if (wordGetBits(st, m - 1, 1) == 1) {
    wordShHi(st, ext / B_PER_W, 1);
    wordXor(st, st, b, ext / B_PER_W);
  }
  free(mem);
}
```

*Утилиты для работы с длинными числами.*

```
#include "utils.h"

uint64 RDTSC()
{
    __asm _emit 0x0F __asm _emit 0x31
}

void printBinaryRepresentation2(word *a, size_t arr_size, size_t m)
{
  size_t i, length = m;
  for (i = 0; i < arr_size; ++i) {
    if (length > B_PER_W) {
      length = length - B_PER_W;
      printBinaryRepresentation1(a[i], B_PER_W);
      printf(" ");
    }
    else {
      printBinaryRepresentation1(a[i], length);
      break;
    }
  }
  printf("end\n");
}

void printBinaryRepresentation1(word a, size_t m)
{
  size_t index = 0;
  if (m > B_PER_W) {
    m = B_PER_W;
  }
  for (index = m; index > 0; --index) {
    printf("%u", (a>>(index-1))&1);
  }
}

void shiftRight(word *a, size_t n, size_t m, size_t shift)
{
  word carry;
  word rest;
  word abc;
  //normalize(a, n, m);
  if (shift > B_PER_W) {
    carry = wordShHiCarry(a, n, B_PER_W, 0);
    wordSetBits(a, m - B_PER_W ,B_PER_W, carry);
    shiftRight(a, n, m, shift - B_PER_W);
```

```
  }
  else {
    abc = (m >= B_PER_W) ? m - B_PER_W : 0;
    carry = wordShHiCarry(a, n, shift, 0);
    if (m < B_PER_W) {
      wordShHi(&carry, 1, B_PER_W - m);
    }
    rest = wordGetBits(a, abc, B_PER_W);
    carry = carry | rest;
    wordSetBits(a, abc, B_PER_W, carry);
  }
}

void normalize(word *a, size_t n, size_t m)
{
  wordTrimHi(a, n, m);
}

size_t bits_in_number(size_t number)
{
  size_t index = 0;
  while (number != 0) {
    index++;
    number = number >> 1;
  }
  return index;
}

size_t next_power_of_two(size_t number)
{
  size_t index = 1;
  while (number != 0) {
    index << 1;
    number = number >> 1;
  }
  return number;
}

size_t size_in_words(size_t number)
{
  word rest = number % B_PER_W;
  size_t size_in_words = number / B_PER_W;
  return rest == 0 ? size_in_words : size_in_words + 1;
};

octet reverseOctet(octet a)
{
  a = (a & 0xF0) >> 4 | (a & 0x0F) << 4;
  a = (a & 0xCC) >> 2 | (a & 0x33) << 2;
  a = (a & 0xAA) >> 1 | (a & 0x55) << 1;
  return a;
}

word reverseWord(word a)
{
  a = (a & 0xFFFF0000) >> 16 | (a & 0x0000FFFF) << 16;
  a = (a & 0xFF00FF00) >> 8 | (a & 0x00FF00FF) << 8;
  a = (a & 0xF0F0F0F0) >> 4 | (a & 0x0F0F0F0F) << 4;
  a = (a & 0xCCCCCCCC) >> 2 | (a & 0x33333333) << 2;
  a = (a & 0xAAAAAAAA) >> 1 | (a & 0x55555555) << 1;
```

```
  return a;
}

void reverse(word *a, word *b, size_t n)
{
  size_t index = 0;
  for (index = 0; index < n / 2; ++index) {
    b[index] = reverseWord(a[n-index-1]);
    b[n-index-1] = reverseWord(a[index]);
  }
}
```

# ПРИЛОЖЕНИЕ Б

*Код программы.*

```
#include <iostream>
#include <set>
#include <vector>
#include <fstream>
#include <time.h>

using namespace std;
typedef unsigned long uint64;
uint64 RDTSC()
{
    __asm _emit 0x0F __asm _emit 0x31
}

static const int _primes[] = { 3...8167};
//остальное вырезано, так как много лишнего

int gcdex (int a, int b, int & x, int & y) {
    if (a == 0) {
        x = 0; y = 1;
        return b;
    }
    int x1, y1;
    int d = gcdex (b % a, a, x1, y1);
    x = y1 - (b / a) * x1;
    y = x1;
    return d;
}

int inverse(int a, int m) {
    int x, y;
    int g = gcdex (a, m, x, y);
    if (g != 1)
        return -1;
    else {
        x = (x % m + m) % m;
        return x;
    }
}

int inv2p(int p) {
    return (p + 1) / 2;

}

int powmod(unsigned long long a, unsigned long long pow, unsigned int mod) {
    unsigned long long ans = 1;
    int b = a % mod;
    while (pow != 0) {
        if (pow % 2 == 1) {
            ans *= b;
            ans %= mod;
        }
        b *= b;
        b %= mod;
        pow /= 2;
```

```
    }
    return ans;
}

bool miller_rabin (int n)
{
    // сначала проверяем тривиальные случаи
    if (n == 2)
        return true;
    if (n < 2 || n % 2 == 0)
        return false;

    // разлагаем n - 1 = 2^s * t
    int t = n - 1, s = 0;
    while (t % 2 == 0) {
        t /= 2;
        ++s;
    }
    int b[3] = {2, 7, 61};
    int rounds = 0;
    while (rounds < 3) {
        // вычисляем b^q mod n, если оно равно 1 или n-1, то n простое (или
            псевдопростое)
        int rem = powmod (b[rounds], t, n);
        if ((rem == 1 || rem == n - 1) && rounds == 1) {
            return true;
        }

        // теперь вычисляем b^2q, b^4q, ... , b^((n-1)/2)
        // если какое-либо из них равно n-1, то n простое (или псевдопростое)
        for (int i = 1; i < s; ++i)
        {
            rem = powmod (rem, 2, n);
            if (rem == n - 1 && rounds == 1) {
                return true;
            }
        }
        ++rounds;
    }
    return false;
}

vector<int> fast_check(int n, int step) {
    set<int> bolter;
    vector<int> answer;
    for (int i = 0; i < step; ++i) {
        int p = _primes[i];
        int r = p - n % p;
        int x = 0;
        if (r % 2 == 0)
            x += r;
        else
            x += r + p;
        if (r == p)
            x = 0;
        for (int j = x; j < _primes[step-1]; j = j + 2 * p) {
            bolter.insert(j);
        }
    }
```

```
    for (set<int>::iterator it = bolter.begin(); it != bolter.end(); ) {
        set<int>::iterator i1 = it, i2 = ++it;
        if (i2 != bolter.end()) {
            int rest = (*i2 - *i1) / 2 - 1;
            while (rest > 0) {
                answer.push_back(n + *i2 - 2 * rest);
                --rest;
            }
        }
    }

    return answer;
}

void func1(int count, int step) {
    srand(time(0));
    vector<int>* a = new vector<int>();
    do {
        do {
            int r = rand() % (INT_MAX - 65536) + 65536;
            if (r % 2 == 0)
                r += 1;
            *a = fast_check(r, step);
        } while (a->size() == 0);
        for (int i = 0; i < a->size(); ++i) {
            if (!miller_rabin(a->at(i))) {
            //  cout << "prime number = " << a->at(i) << endl;
                count--;
            }
        }
    } while (count > 0);

}

void func2(int count) {
    int r = 0;
    do {
        do {
            r = rand () % (INT_MAX - 65536) + 65536;
            if (r % 2 == 0)
                r += 1;
        } while (!miller_rabin(r));
        //cout << "prime number = " << r << endl;
    } while (--count > 0);
}

int main() {
    vector<uint64> f1, f2;
    fstream out;
    out.open("freq1.txt", fstream::out);
    uint64 start, overhead, freq1, freq2;
    for (int i = 2; i < 1024; i += 1) {
        start = RDTSC();
        overhead = RDTSC() - start;
        func1(1000, i);
        freq1 = RDTSC() - start - overhead;
        f1.push_back(freq1 / 1000);
        out << freq1 / 1000 << endl;
        cout << i << "-> OK " << endl;
    }
```

```
    start = RDTSC();
    func2(1000);
    freq2 = RDTSC() - start - overhead;
    cout << "Processor ticks, elapsed on  miller-rabin = " << freq2 / 1000 <<
        endl;
    out << endl << freq2 / 1000;
    out.close();
    return 0;
}
```